AUTOMATICALLY EVOLVING RULE INDUCTION ALGORITHMS WITH GRAMMAR-BASED GENETIC PROGRAMMING

A THESIS SUBMITTED TO
THE UNIVERSITY OF KENT AT CANTERBURY
IN THE SUBJECT OF COMPUTER SCIENCE
FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY.

By
Gisele Lobo Pappa
August 2007

To my granddad Lobo
(*in memoriam*)

Contents

List of Tables

List of Figures

List of Algorithms

Abstract

In the last 30 years, research in the field of rule induction algorithms produced a large number of algorithms. However, these algorithms are usually obtained from the combination of a basic rule induction algorithm (typically following the sequential covering approach) with new evaluation functions, pruning methods and stopping criteria for refining or producing rules, generating many "new" and more sophisticated sequential covering algorithms.

We cannot deny that these attempts to improve the basic sequential covering approach have succeeded. Hence, if manually changing these major components of rule induction algorithms can result in new, significantly better ones, why not to automate this process to make it more cost-effective? This is the core idea of this work: to automate the process of designing rule induction algorithms by means of grammar-based genetic programming.

Grammar-based Genetic Programming (GGP) is a special type of evolutionary algorithm used to automatically evolve computer programs. The most interesting feature of this type of algorithm is that it incorporates a grammar into its search mechanism, which expresses prior knowledge about the problem being solved. Since we have a lot of previous knowledge about how humans design rule induction algorithms, this type of algorithm is intuitively a suitable tool to automatically evolve rule induction algorithms.

The grammar given to the proposed GGP system includes knowledge about how humans design rule induction algorithms, and also presents some new elements which could work in rule induction algorithms, but to the best of our knowledge were not previously tested. The GGP system aims to evolve rule induction algorithms under two different frameworks, as follows. In the first framework, the GGP is used to evolve *robust* rule induction algorithms, i.e., algorithms which were designed to be applied to virtually any classification data set, like a manually-designed rule induction algorithm. In the second framework, the GGP is applied to evolve rule induction algorithms tailored to a specific application

domain, i.e., rule induction algorithms tailored to a single data set. Note that the latter framework is hardly feasible on a hard scale in the case of conventional, manually-designed algorithms, since the number of classification data sets greatly outnumbers the number of rule induction algorithms designers. However, it is clearly feasible on a large scale when using the proposed system, which automates the process of rule induction algorithm design and implementation.

Overall, extensive computational experiments with 20 UCI data sets and 5 bioinformatics data sets showed that effective rule induction algorithms can be automatically generated using the GGP in both frameworks. Moreover, the automatically evolved rule induction algorithms were shown to be competitive with (and overall slightly better than) four well-known manually designed rule induction algorithms when comparing their predictive accuracies.

The proposed GGP system was also compared to a grammar-based hill-climbing system, and experimental results showed that the GGP system is a more effective method to evolve rule induction algorithms than the grammar-based hill-climbing method.

At last, a multi-objective version of the GGP (based on the concept of Pareto dominance) was also proposed, and experiments were performed to evolve *robust* rule induction algorithms which generate both accurate and simple models. The results showed that in most of the cases the GGP system can produce rule induction algorithms which are competitive in predictive accuracy to well-known human-designed rule induction algorithms, but generate simpler classification modes – i.e., smaller rule sets, intuitively easier to be interpreted by the user.

Acknowledgements

There are two people who have to take the responsibility for me being here, today, writing these acknowledgements: Alex Freitas (my MSc co-supervisor and PhD supervisor) and Celso Kaestner (my MSc principal supervisor). If it was not for their incentive, guidance and support, I would have probably known London, Paris, Madrid, Prague and so on ... just by post cards!

After these two, I want to thank my family, whose support was essential when I decided to go for a new life. Pai, mãe, vó, Ju, Line, Jé, Xandi, Léia, Dani and Edgar, you were always here with me.

I would not have started this journey without my old friends, and would not have finished it without the new ones. I would not like to name any of you because I know I might forget somebody (you know better than me that my brain is not working properly lately... yes, before it used to!), but at the same time I can see some of you complaining your names were not in the acknowledgements, so... David (besides being my friend, he was my Latex "reference book"), Cris, Fer, Liner, Márcio, Dani, Dela, Márcia Maria, Sid, Kiran, Nick, Sam, Chris, Emma, Maurizio, Matteo, Fernando, Rogério, Alex, Cat, Alvaro, Antonio and Jó, thanks for all the support, the beers and the infinite discussions about this work.

In special, I want to thank Miguel, for all his patience, emotional and ... "statistical" support.

I also want to thank Colin Johnson and Mathieu Capcarrere, members of my PhD supervisory panel, who evaluated the progress of this work for the past three years, giving some interesting feedback; and Fred Barnes, who made available to me a computer cluster, so that the experiments of these thesis were completed on a reasonable time.

Finally, I would like to thank CAPES, the Brazilian Research Council, who gave me the full financial support for this research (grant number 165002-5).

Chapter 1

Introduction

Data mining is the process of automatically extracting accurate, comprehensible and interesting knowledge from data. It can be used to perform a variety of tasks, including data association, data regression and data classification [49, 52].

The classification task, in particular, intends to automatically separate objects belonging to different classes. When performing classification, a data mining algorithm generates a classification model from a data set, and this model can be later applied to classify objects whose class is unknown.

The classification models generated by data mining algorithms can be represented using a variety of knowledge representations [13, 145]. Broadly speaking, classification models can be divided into two categories, depending mainly on the type of knowledge representation being used: human-comprehensible and black-box models. Examples of human-comprehensible models include classification rules, decision trees, and bayesian networks, whereas instance-based representations, artificial neural networks and support vector machines represent black-box models.

In some application domains, where an understanding of the classification model is unnecessary, black-box representations are very successful, and can generate very accurate models. However, in domains where the discovered knowledge is only useful if it can be interpreted, like in medical domains, human-comprehensible models are essential [117, 49]. For instance, in principle a medical doctor should not recommend a patient for surgery based only on the prediction made by a classification algorithm; it is important that the doctor interprets the model in the context of his/her own knowledge background. As another example, most users would hesitate in investing a large amount of money in a financial application based on the prediction of a black-box model [39].

In this thesis, we are interested in classification models which are both accurate and comprehensible. More precisely, we focus on knowledge represented as classification rules. A classification rule has the format "IF *cond1* AND *cond2* $\cdots$ THEN *conseq*", where the conditions in the antecedent are described by associations between attributes and their values (e.g. salary $>$ £30,000), or two different attributes (e.g. salary $>$ mortgage). The consequent represents the predicted value for the class attribute.

There are several different types of algorithms available to produce classification rules (see Chapter 2). In this thesis we are interested on one of them: rule induction algorithms, which in general lend themselves naturally to the discovery of comprehensible classification rules. For the past 30 years, a great variety of rule induction algorithms were *manually* designed. However, to the best of our knowledge, rule induction algorithms, like CN2 [26] or Ripper [32], were never *automatically* generated.

The automation of any task can be achieved by programming a machine to follow, step by step, the process a trained human would follow to execute it, obtaining, at the end, the same or similar results. There has been significant progress in the automation of data analysis tasks. A relevant example consists of methods to automatically select the most suitable classification algorithm for a data set being mined, out of several candidate algorithms [119]. Another interesting example is the "scientist robot" described in King et *al.* [83]. This system automatically generates hypotheses to explain observations in biological data, plans and physically executes experiments to test the hypotheses – using a laboratory robot, and interprets the results to falsify hypotheses inconsistent with the data.

The human desire to automatically create computer programs for machine learning tasks dates back to the pioneering work of Samuel in the sixties, when the term machine learning was first coined meaning "computers programming themselves". As years went by, machine learning gained a new definition, related to the system's capability of learning from experience [97]. However, the idea of automatically generating computer programs persisted. In the early nineties, a whole new area dedicated to the study of this idea started to be widely disseminated: genetic programming [84].

The main idea behind genetic programming (GP) is to automatically evolve computer programs capable of producing good solutions (hopefully better than manually-produced solutions) for the target problem.

Regarding the classification task, GP has been used to evolve classification

rules for a specific data set [52], to combine classifiers [87], to evolve neural networks [150], and even to evolve other evolutionary algorithms [108] (which can then be used as classification algorithms), but not, to the best of our knowledge, to evolve a complete rule induction algorithm. This is the main goal of this work: to create a new GP system to automatically evolve complete rule induction algorithms.

Automatically evolving a rule induction algorithm "from scratch" would certainly be an extremely hard task for a GP system. However, if we provide it with some background knowledge about the basic structure of rule induction algorithms, the task becomes more feasible. This is exactly the basic principle of Grammar-based GP, and this is the approach followed in this thesis.

Grammar-based GP (GGP) [143] is a special type of GP that incorporates in its search mechanism prior knowledge (expressed in the form of a grammar) about the problem being solved. Intuitively, GGP is an appropriate method for automatically evolving rule induction algorithms for two main reasons. First, it makes use of what we already know about the design of rule induction algorithms (background knowledge accumulated through several decades of research). Second, it provides an automatic way of performing a global search that evaluates, in parallel, many combinations of elements of rule induction algorithms, which can find new, potentially more effective rule induction algorithms.

1.1 Motivation

There are three motivations for automatically evolving rule induction algorithms. First, it represents a further step towards the automation of the data analysis task. Instead of automatically evolve *rule sets* for specific data sets, as many systems already do, it would automatically evolve *complete rule induction algorithms.* These automatically evolved rule induction algorithms could then be applied to generate rule sets for virtually any classification data set.

Second, all current rule induction algorithms were manually developed by a human being, and so they inevitably incorporate some human preconceptions and biases. In particular, the vast majority of rule induction algorithms select one attribute-value-at-a-time, in a greedy fashion, ignoring attribute interactions. A machine-developed algorithm could completely change this kind of algorithm preconceptions, since "its bias" would be different from the kind of algorithm bias imposed by a human designer.

A good example of a "free of preconceptions" solution created by an Evolutionary Algorithm (EA) is the design of a satellite dish holder boom, which connects the satellite's body with the dish needed for communication [46]. The human-designed structure has a symmetric ladder-like shape. The structure found by the EA looks like a random drawing, and it is not intuitive at all. However, it is approximately 20,000% better than the traditional human-designed structure.

The third motivation to automatically design rule induction algorithms is as follows. It has already been shown that no classification algorithm is the best to solve all kinds of classification problems [90]. The impact the choice of a suitable algorithm has in the classification model generated from the data is so big that meta-learning [139] emerged as a whole new research area dedicated to study this problem.

In particular, the STATLOG [98] and METAL [118] projects put together the efforts of many researchers to learn how to characterize data sets via "meta-attributes", i.e., attributes describing an entire data set, rather than describing an individual example (data instance, or record). Then they used these meta-attributes to create a classification "meta-model" capable of selecting the most suitable classification algorithm for each data set.

Despite the progress obtained with these efforts, the choice of which classification algorithm to apply to a specific data set is still an open problem in general. This is because there are two major limitations in almost all of the meta-learning techniques. First, they have to identify the best meta-attributes which characterize the data. Choosing the right set of meta-attributes can be an extremely difficult task, given the huge diversity of classification data sets. Second, they perform "algorithm selection". They try to select the best algorithm out of a small pre-defined set of algorithms, but there is no guarantee that the set of candidate algorithms will include a very good classification algorithm for a specific data set.

In order to bypass these two limitations associated with meta-learning approaches, this work focuses on automated "algorithm construction". This approach avoids the limitations of conventional meta-learning, and presents two main advantages over the latter: (a) it can produce rule induction algorithms potentially better than the available ones, and, as a related point, the system can always be updated to produce more new rule induction algorithms by simply modifying the grammar the GGP works with; (b) it presents a much cheaper alternative to the manual design of rule induction algorithms, specially because it can produce a rule induction algorithm tailored to a specific data set. Note

that the manual design of a rule induction algorithm tailored to a specific data set would be much more time consuming and is not feasible on a large scale.

The previously presented motivations are in general valid regardless of the type of classification algorithms being evolved. Nevertheless, we had two additional motivations to evolve rule induction algorithms instead of any others. The first one, as explained before, concerns the type of human-comprehensible classification models they generate.

Besides, research in the rule induction field has being carried out for more than 30 years and certainly produced a large number of algorithms. However, these algorithms are usually obtained from the combination of a basic rule induction algorithm (typically following the sequential covering approach) with new evaluation functions, pruning methods and stopping criteria for refining or producing rules, generating many "new" and more sophisticated sequential covering algorithms. Hence, in some sense, we can say there was a "human-driven evolution" of rule induction algorithms in the past decades. We propose to take advantage of this "human-driven evolution" and extend it to a new type of evolution, by automating the design of new rule induction algorithms by means of an evolutionary algorithm – more precisely, a grammar-based genetic programming system.

1.2 Aims and Objectives

The overall aim of this work is to create a new Grammar-based Genetic Programming (GGP) system able to automatically evolve rule induction algorithms. In order to achieve this aim, our first objective is to design a grammar which reflects not only the main structure of human-designed rule induction algorithms, but also includes some other ideas that we think might work in rule induction algorithms, even though they were not previously tested – to the best of our knowledge. This grammar can then be used by the GGP to generate not only well-known rule induction algorithms but also new, potentially better rule induction algorithms.

Once the grammar is defined, the next objective is to design a GGP system which uses the grammar to produce rule induction algorithms according to one of the following two goals:

1. To evolve a *robust* rule induction algorithm from multiple data sets, so that the evolved rule induction algorithm should obtain a high predictive accuracy across a range of very different data sets (from very different application domains); or

2. To evolve a rule induction algorithm tailored to *one specific application domain* (a specific data set).

In order to evaluate the proposed GGP system, another objective of this thesis is to compare the rule induction algorithms produced by the GGP system with well-known human-designed rule induction algorithms. This comparison will be based mainly on the predictive accuracy of the rule induction algorithms produced, but it will also take into account the degree of "innovation" (or "originality") of the automatically evolved rule induction algorithms, by comparison with well-known manually-designed algorithms.

Furthermore, the proposed GGP system will also be compared to a much simpler grammar-based hill climbing approach. The motivation for this comparison is to justify the use of a more sophisticated system, such as the GGP, to automatically evolve rule induction algorithms, instead of a simple greedy search algorithm as the hill climbing.

Finally, in order to consider not only the predictive accuracy but also the complexity of the rule models generated when evolving rule induction algorithms, a multi-objective fitness function will be introduced to the initial proposed system.

1.3 Contributions

This thesis shows that genetic programming can be used together with previous knowledge about human-designed rule induction algorithms (incorporated into a grammar) to automatically evolve new rule induction algorithms. More precisely, based on the objectives of this thesis, its main contributions are:

- A grammar, which represents the basic structure and the major components (or "building blocks") of many human-designed rule induction algorithms.

- A new grammar-based genetic programming system, which automatically evolves rule induction algorithms. This system can be applied to generate either *robust* or data set-tailored rule induction algorithms. This new GP system is the major contribution of this thesis, and it can be considered a pioneering contribution, since – to the best of our knowledge – it is the first GP system for automatically creating a fully-fledged rule induction algorithm.

- New automatically evolved rule induction algorithms – in particular, new algorithms that are quite innovative and competitive with respect to well-known manually-designed algorithms.

1.4 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 discusses the three main methods used to extract classification rules from data: sequential covering algorithms, evolutionary algorithms, and methods for extracting classification rules from other types of knowledge representation. It specially describes in detail the main components of sequential covering algorithms, which will be used to create the grammar which will guide the proposed Grammar-based Genetic Programming (GGP) system.

Chapter 3 describes the main ideas behind a GP system, and shows how one of its drawbacks, i.e. the closure problem, led to the development of one of its variations: GGP. In order to give a clear view of GGP systems, we first introduce the basic concepts of grammars, and then describe two types of GGPs: solution-encoding individual and production-rule-sequence-encoding individual. In the first of them, an individual directly encodes a candidate solution (rule induction algorithm), whilst in the other one an individual encodes a sequence of production rules from the grammar, whose applications generate a rule induction algorithm.

Chapter 4 introduces the proposed GGP system to automatically evolve rule induction algorithms. It starts by giving a detailed description of the grammar, followed by a discussion of design and implementation issues concerning the main components of the system. These main components include the individual representation, the population initialization procedure, the individual evaluation process and the crossover and mutation operators. Finally, it describes related work and discusses the differences among related systems and the proposed system.

Chapters 5 and 6 report the extensive results of a large set of computational experiments using the GGP system into two different frameworks. In Chapter 5, the evolution of robust rule induction algorithms, which were evolved to be effectively applied to any classification data set, is studied. It also presents a multi-objective version of the GGP system proposed. In Chapter 6, the evolution of rule induction algorithms tailored to a specific application domain (a specific data set) is described. In general, these chapters show how the GGP parameters were set, how the rule induction algorithms produced by the GGP compare to well-known

human-designed rule induction algorithms and what is different in their design elements. They also compare the rule induction algorithms evolved by the GGP with the rule induction algorithms generated by a grammar-based hill climbing search method.

Chapter 7 presents the conclusions and describes future research directions.

Chapter 2

Rule Induction Algorithms

2.1 Introduction

For a long time researchers have been dreaming about machines able to learn and program themselves. The field of machine learning first emerged with the ambitious goal of creating these dreamed machines. However, as the area developed, researchers realized we barely understand how the human learning process works, so simulating it on a machine was not a simple task. At this point, machine learning was redefined as the "machines' ability to improve through experience" [99].

Using this new definition of machine learning, one of the first tasks researchers programmed machines to learn was classification. The classification task aims to separate objects belonging to different classes, where a class is defined as a group of objects sharing common features. A classification system learns how to distinguish examples from classes C_1 and C_2 (or more classes) by "studying" a set of examples containing objects from both classes, named the training set. Each training example is represented by a set of predictor attributes and a goal (or class) attribute. The aim of the classification system is to produce a classification model by exploring relationships between the predictor and the goal attributes.

In order to evaluate the quality of the classification model created by the classification system, the model is tested in a new set of examples (records) from the same application domain of the training set, named the test set. The examples presented in the test set are different from the ones in the training set, and are used to evaluate the generalization ability of the classification model.

The model generated by a classification system can be described using a variety of knowledge representations, such as decision trees, decision rules, artificial

neural networks, instance-based representations, Bayesian networks, support vector machines, etc [13, 145]. In this thesis we focus on rule induction algorithms, i.e., methods which produce classification models represented by rules. The motivation for focusing on rules is that this kind of knowledge representation tends to be intuitively comprehensible to the user, as discussed in Chapter 1.

A classification rule has the format "IF *cond1* AND *cond2* $\cdots$ THEN *conseq*", where the conditions in the antecedent are described by associations between attributes and their values (e.g. salary $>$ £10,000), or two different attributes (e.g. salary $>$ mortgage). The consequent represents the predicted value for the goal attribute. An example of a classification rule is "IF ((sex = female) AND (salary $>$ £30,000) AND (age $>$ 45)) THEN marital-status = divorced". This rule states that women older than 45 and who earn more than £30,000 belong to the class of divorced people.

Classification models represented by rule sets can be generated using many different methods. Here we explore three commonly used strategies to generate rule sets [99], namely:

1. The sequential covering (or separate and conquer) strategy [145].

2. The use of evolutionary algorithms, like genetic algorithms and genetic programming, to extract rules from data [80, 52, 148].

3. The generation of a classification model based on a knowledge representation different from rule sets (like a neural network or a decision tree) followed by a rule extraction method [124, 77].

The next sections describe these three strategies used to generate rule sets from data, and show examples of how rule induction algorithms implement them.

2.2 The Sequential Covering Strategy

The sequential covering strategy (also known as separate and conquer) is certainly the most explored and most used strategy to induce rules from data. It was first employed by the algorithms of the AQ family [96] in the late sixties, and over the years was applied again and again as the basic algorithm in rule induction systems.

The separate and conquer strategy works as follows. It learns a rule from a training set, removes from it the examples covered by the rule, and recursively

learns another rule which covers the remaining examples. A rule is said to cover an example e when all the conditions in the antecedent of the rule are satisfied by the example e. For instance, the rule "IF (salary > £ 100,000) THEN rich" covers all the examples in the training set in which the value of salary is greater than £100,000, regardless of the current value of the class attribute of an example.

The learning process goes on until a pre-defined criterion is satisfied. This criterion usually requires that all or almost all examples in the training set are covered by a rule. Alg. 2.1 shows the basic pseudo-code for sequential covering algorithms producing rule sets (a somewhat different algorithm for generating ordered lists of rules is presented in Section 2.2.1), and Alg. 2.2 describes the procedure *LearnOneRule* used by Alg. 2.1.

Algorithm 2.1: CreateRuleSet

```
RuleSet = ∅
for each class C_i do
    Set training set T as the entire set of training examples
    while Rules Stopping Criterion not satisfied do
        Create an Initial Rule R
        Set class predicted by new rule R as C_i
        R' = LearnOneRule(R)
        RuleSet = RuleSet ∪ R' (i.e, Add R'to RuleSet)
        Remove from T all class C_i examples covered by R'
Post-process RuleSet
return RuleSet
```

In Alg. 2.1 and Alg. 2.2, elements in italic represent a set of building blocks which can be instantiated to create different types of sequential covering algorithms. The block "*Create an Initial Rule R*" in Alg. 2.1, for example, can be replaced by "*Create an empty Rule R*" or "*Create a Rule R using a random example from the training set*". The block "*Evaluate CR*" in Alg. 2.2 could be replaced by "*Evaluate CR using accuracy*" or "*Evaluate CR using information gain*".

Replacing building blocks in this basic algorithm by a specific method can create almost any of the existing sequential covering rule induction algorithms. This is possible because algorithms following the sequential covering approach usually differ from each other in four main elements: the representation of the candidate rules, the search mechanisms used to explore the space of candidate rules, the way the candidate rules are evaluated and the pruning method, although the last one can be absent [57, 145].

Before going into detail about these four elements, let us describe a couple of

Algorithm 2.2: LearnOneRule(R)

```
bestRule = R
candidateRules = ∅
candidateRules = candidateRules ∪ bestRule
while candidateRules ≠ ∅ do
    newCandidateRules = ∅
    for each candidateRule CR do
        Refine CR
        Evaluate CR
        if Refine Rule Stopping Criterion not satisfied then
            newCandidateRules = newCandidateRules ∪ CR
            if CR is better than bestRule then
                bestRule = CR
    candidateRules = Select b best rules in newCandidateRules
return bestRule
```

sequential covering algorithms that do not adopt exactly the pseudo-code defined in Alg. 2.1. This is appropriate since it shows that attempts to improve this basic algorithm were made. Note that even though these new algorithms proved to be competitive with the traditional algorithms, currently the most used and accurate algorithms stick to the simple and basic approach described by the pseudo-code in Alg. 2.1.

PN-Rules [3] and LERILS [24] are rule induction algorithms that slightly changed the dynamics of the basic algorithm showed in Alg. 2.1. PN-Rules, for instance, is based on the concept that over-fitting can be avoided by adjusting the trade-off between support (number of examples covered by a rule) and accuracy during the process of building a rule. The main difference between PN-Rules and the traditional algorithms is that the former finds two sets of rules: a set of P-rules and a set of N-rules. P-rules are learned first, favor high coverage, and are expected to cover most of the positive examples (i.e. examples belonging to the class predicted by the rule) in the training set. In a second step, a set of N-rules is created using only the set of examples covered by the P-rules. The idea is that the N-rules can exclude the false positives (examples covered by a P-rule but which belong to a different class from the predicted one) of the covered examples. This 2-phase process is repeated for all the classes. During the classification of examples in the test set, a scoring method assesses the decisions of each binary classifier, and chooses the final classification.

In the case of LERILS, maybe it is not appropriated to call it a sequential

covering algorithm, since it does not remove examples from the training set after learning a rule. All the rules are learned using the whole set of training examples. However, apart from not removing examples from the training set after creating a rule, all its other elements are based on conventional instances of the components found in the basic algorithm representing the sequential covering approach. LERILS works in 2 phases. First it uses a bottom-up search combined with a random walk to produce a pool of k rules, where k is a parameter defined by the user (since the examples are not removed from the training set after creating a rule, a fixed number of rules is defined as the stopping criterion). In a second phase, it uses again a random procedure together with the minimum description length heuristic to combine the rules into a final rule set.

The literature in rule induction and specially sequential covering algorithms is very rich. There are several surveys - e.g. [57, 122] - and papers comparing a variety of algorithms [56, 6, 98, 90]. The next sections summarize the main points to be considered when creating a sequential covering algorithm, and in particular the specific methods that can replace the building blocks presented in Algs. 2.1 and 2.2. For a more detailed description the reader is referred to the original papers describing the methods.

2.2.1 Representation of the Candidate Rules

The rule representation has a significant influence in the learning process, since some concepts can be easily expressed in one representation but hardly expressed in others. In particular, rules can be represented using propositional or first order logic.

Propositional rules are composed by selectors, which are associations between pairs of attribute-values, like $age > 10$, $salary < 2000$ or $sex = male$. Besides the operators ">", "<" and "=", "$\leq$", "$\geq$" and "$\neq$" are also available in some methods. "=" and "$\neq$" are used for nominal attributes, whilst the others are used with continuous attributes. The majority of the algorithms build a rule as a conjunction of selectors, although some algorithms of the AQ family, for example, also allow internal disjunctions and intervals. CN2 [27], C4.5 rules [124], and Ripper [33] are examples of propositional rule induction algorithms.

First-order rules are more sophisticated, and can express relations between two attributes, generating rules with conditions such as $x > y$. When using a first-order representation, the concepts are usually represented as Prolog relations, like

father(x,y). Methods that use this Prolog representation are classified as Inductive Logic Programming (ILP) systems [88, 89].

ILP uses the same principles of rule induction algorithms, essentially replacing the concepts of conditions and rules by literals and clauses. In addition, ILP techniques allow the user to incorporate into the algorithm background knowledge about the problem, which helps to focus the search in promising areas of the search space. FOIL [123] and REP [17] use this representation.

Apart from these two main representations, a few algorithms use some different representations. BEXA [135], for example, uses the multiple-valued extension to propositional logic to represent rules, while systems like FuzzConRI [152], which use fuzzy logic, are becoming more common.

Besides different rule representations, there are also different types of classification models which can be created when combining single rules into a collection of rules. The rule models generated by a rule induction algorithm can be ordered (also known as rule lists or decision lists) or unordered (rule sets).

In rule lists the order in which the rules are learned is important because rules will be applied in order when classifying new examples in the test set. In other words, the first rule in the ordered list that covers the new example will classify it, whereas subsequent rules will be ignored. In contrast, in rule sets the order in which the rules are produced is not important, since all the rules in the model are considered when classifying a new example. In the latter case, when more than one rule covers a new example, and the class predicted by them is not the same, a tie-break criterion can be applied to decide which rule gives the best classification. Examples of these criteria are selecting the rule with higher coverage or heuristics like the Laplace estimation [41]. Alg. 2.1, introduced in the beginning of Section 2.2, represents the algorithm employed to generate a set of unordered rules. A few changes have been introduced to it in order to generate an ordered decision list. Alg. 2.3 describes the new pseudo-code to generate a decision list following the sequential covering approach.

Comparing Algs. 2.1 and 2.3, the outer *for* loop from Alg. 2.1 is absent in Alg. 2.3 because decision list algorithms do not learn rules for each class in turn. Instead they learn rules for all classes at the same time. The set of examples removed from the training set after a rule is learned also changes. While in Alg. 2.1 only the covered examples belonging to the current class C_i are removed, in rule list algorithms all the covered examples (no matter their class) are removed. This happens because as rule lists apply rules in order, if one rule covers an example,

Algorithm 2.3: CreateDecisionList

```
RuleList = ∅
repeat
    Create an Initial Rule R
    R' = LearnOneRule(R)
    Set consequent of learned rule R' as the most frequent class found in
    the set of examples covered by R'
    Add R' to RuleList
    Remove from T all the examples covered by R'
until Rules Stopping Criterion not satisfied
return RuleList
```

no other rules will have the chance to classify it.

Rule lists are considered more difficult to understand than rule sets. This is because in order to comprehend the last rule of a list all the previous rules must also be taken in consideration [26]. Since the knowledge generated by rule induction algorithms is supposed to be analyzed and validated by an expert, rules at the end of the list become very difficult to understand, particularly in very long lists. Hence, unordered rules are often favored over ordered ones.

2.2.2 Search Mechanism

A rule induction algorithm acts like a search algorithm exploring a space of candidate rules. Its search mechanism has two components: a search strategy and a search method. The search strategy determines the region of the search space where the search starts and its direction, while the search method specifies which specializations/generalizations should be considered. The building blocks "*Create an Initial Rule R*" (in Alg. 2.1) and "*Refine CR*" (in Alg. 2.2) determine the search strategy of a sequential covering algorithm. "*Select b best rules in newCandidateRules*", in Alg. 2.2, implements its search method.

Broadly speaking, there are three kinds of search strategies: bottom-up, top-down and bi-directional. A bottom-up strategy starts the search with a very specific rule, and iteratively generalizes it. This specific rule is usually an example from the training set, chosen at random (like in LERILS [24]) or using some heuristic. RISE [41], for example, starts considering all the examples in the training set as rules. Then, for each rule in turn, it searches for its nearest example of the same class according to a distance metric, and attempts to generalize the rule (example) to cover that nearest example.

A top-down strategy, in contrast, starts the search with the most general rule and iteratively specializes it. The most general rule is the one that covers all examples in the training set (because it has an empty antecedent, which is always satisfied for any example). The top-down strategy is more frequently used by sequential covering algorithms but its main drawback is that as induction goes on, the amount of data available to evaluate a candidate rule decreases drastically, reducing the statistical reliability of the rules discovered. This usually leads to data over-fitting and the small disjunct problem [19].

However, there are ways to prevent over-fitting in top-down searches. One of them is to stop the rules' production after the data set size falls bellow some threshold (although this approach can miss some rare but important concepts). Pruning methods, discussed in Section 2.2.4, are also an attempt to avoid this problem.

At last, a bi-directional search is allowed to generalize or specialize the candidate rules. It is not a common approach but it can be found in the SWAP-1 [142] and *Reconsider and Conquer* [15] algorithms. When looking for rules, SWAP-1 first tries to delete or replace the current rules' conditions before adding a new one. *Reconsider and Conquer*, in turn, starts the search with an empty rule, but after inserting the first best rule to the rule set, it backs up and carries on the search process using the candidate rules previously created.

After selecting the search strategy, a search method has to be implemented. The search method is a very important part of a rule induction algorithm since it determines which specializations/generalizations will be considered at each specialization/generalizations step. Too many specializations/generalizations are not allowed due to computational time, but too few may disregard good conditions and reduce the chances of finding a good rule. Among the available search methods are the greedy search and the beam search.

The greedy and the beam searches are the most popular methods. Greedy algorithms create an initial rule, generalize/specialize it, evaluate the extended rules created by the generalization/specialization operation, and keep just the best extended rule. This process is repeated until a stopping criterion is satisfied. PRISM [22] is just one among many algorithms that use greedy search.

Although they are fast and easy to implement, greedy algorithms have the well-known myopia problem: at each rule extension step, they make the best local choice, and cannot backtrack if later in the search the chosen path is not good enough to discriminate examples belonging to different classes. As a result, they

do not cope well with attribute interaction [39, 52].

Beam search methods try to eliminate the drawbacks of greedy algorithms selecting, instead of 1, the b best extended rules at each iteration, where b is the width of the beam. Hence, they explore a larger portion of the search space than greedy methods, coping better with attribute interaction. Nevertheless, learning problems involving very complex attribute interactions are still a very difficult problem for beam search algorithms, and for rule induction/decision tree algorithms in general [131]. CN2, AQ and BEXA algorithms implement a beam search.

Apart from these two conventional search methods, some algorithms try to innovate in the search method in order to better explore the space of rules. ITRULE [133], for instance, uses a depth-first search, while LERILS [24] applies a randomized local search. Furthermore, stochastic search methods use evolutionary approaches, such as genetic algorithms and genetic programming, to explore the search space. Examples of systems using this approach will be discussed in Section 2.3.

In conclusion, the main problem with the search mechanism of sequential covering algorithms nowadays is that, regardless of performing a top-down or a bottom-up search, most of them use a greedy, hill-climbing procedure to look for rules.

A way to make these algorithms less greedy is to use a n-step look-ahead hill-climbing procedure. Hence, instead of adding/removing one attribute-at-a-time from a rule, the algorithm would add/remove n conditions-at-a-time. This approach was attempted by some decision-tree algorithms in the past, but there is no strong evidence of whether look-ahead improves or harms the performance of the algorithm. While Dong and Kothari [42] concluded that look-ahead produces better trees (using a nonparametric look-ahead method), Murthy and Salzberg [102] argued it can produce larger and less accurate trees (using a one-step look-ahead method). A more recent study by Esmeir and Markovich [47] used look-ahead for anytime induction decision trees, and found that look-ahead produces better trees and higher accuracies, as long as a large amount of time is available.

Look-ahead methods for rule induction were tested by [58] in a bottom-up algorithm. One and two step look-ahead were used, and they sightly improved the accuracy of the algorithm in the data sets used in the experiments, but at an algorithm quadratic cost. Nevertheless, further studies analyzing the impact of deeper look-ahead in the bottom-up and top-down approach are needed to reach

stronger conclusions about their effectiveness. It is believed that computational time is one of the reasons which prevents the use of look-ahead in rule induction. However, in domains in which time can be sacrificed for better classification models, it is an idea worth trying.

2.2.3 Rule Evaluation

The regions of the search space being explored by a rule induction algorithm can drastically change according to the evaluation heuristic used to assess a rule while it is being built. This section describes some of the heuristics used to estimate rule quality. In all the formulas presented, P represents the total number of positive examples in the training set, N represents the total number of negative examples in the training set, p represents the number of positive examples covered by a rule R and n the number of negative examples covered by a rule R. In Alg. 2.2, the building block "*Evaluate CR*" is responsible for implementing rule evaluation heuristics.

When searching for rules, the first aim of most of the sequential covering algorithms is to find rules that maximize the number of positive covered examples and, at the same time, minimize the number of negative covered examples. It is important to note that these two objectives are conflicting because as the number of covered examples increases, the tendency is that the number of negative covered examples will also increase. Examples of heuristics used by these algorithms are confidence, Laplace estimation, m-estimate and ls-content.

Confidence (also known as precision or purity) is the simplest rule evaluation function and is described as in Eq. (2.1).

$$confidence(R) = \frac{p}{p+n} \tag{2.1}$$

It is used by SWAP-1, and its main drawback is that, in the case of, say, a rule R_1 covering 95 positive examples and 5 negative examples (confidence = 0.95), and a rule R_2 covering 3 positive examples and no negative examples (confidence = 1), it will prefer R_2. This is undesirable, because R_2 is not a statistically reliable rule, being based on such a small number of covered examples.

In order to overcome this problem with the confidence measure, the Laplace estimation (or "correction") measure was introduced, and it is defined in Eq. (2.2).

In Eq. (2.2), *nClasses* is the number of classes available in the training set. Using this heuristic, rules with apparent high confidence but very small statistical

support are penalized. Consider the previously mentioned rules R_1 and R_2 in a 2-class problem. Now the Laplace estimation values are 0.94 for R_1 and 0.8 for R_2. Now R_1 would be preferred over R_2, as it should be. Laplace estimation is used by the CN2 [26] and BEXA [135] algorithms.

$$laplaceEstim(R) = \frac{p+1}{p+n+nClasses} \tag{2.2}$$

m-estimate [44] is a generalization of the Laplace estimation, where a rule with 0-coverage is evaluated considering the classes' *a priori* probabilities, instead of $1/nClasses$. More precisely, m-estimate is computed by Eq. (2.3), where m is a parameter. Eq. (2.3) corresponds to adding m virtual examples to the current training set, distributed according to the prior probabilities of the classes. Hence, higher values of m give more importance to the prior probabilities of classes, and their use is appropriate in data sets with high level of noise.

$$m-estimate(R) = \frac{p+m\frac{P}{P+N}}{p+n+m} \tag{2.3}$$

The ls-content measure, shown in Eq. (2.4), divides the proportion of positive examples covered by the proportion of negative examples covered by the rule, both estimated using the Laplace estimation. $P+nClasses$ and $N+nClasses$ can be omitted because they are constant during the rule refinement process. The ls-content is used by the HYDRA [5] algorithm.

$$ls-content(R) = \frac{\frac{p+1}{P+nClasses}}{\frac{n+1}{N+nClasses}} \cong \frac{p+1}{n+1} \tag{2.4}$$

From these four described heuristics, the Laplace estimation and the m-estimate are the most successfully used, mainly because of their small sensitivity to noise.

A second desired feature in rules is simplicity. Rule size is the most straightforward measure of simplicity, but more complicated heuristics such as the minimum description length [123] can also be applied. Nevertheless, heuristics to measure simplicity are most of the time combined with other rule evaluation criteria.

The minimum description length provides a trade-off between size and accuracy using concepts of information theory. It first estimates the size of a "theory" (a rule set, in the context of rule induction algorithms) in number of bits, and then add to it the number of bits necessary to encode the exceptions relative to

the theory, using an informational loss function [145]. The aim is to minimize the description length of the theory, so the trick is to find the best way to code the theory using a minimal number of bits.

Within the group of heuristics that measure coverage/size are the ones based on gain, which compute the difference in the value of some heuristic function measured between the current rule and its predecessor. Information gain is the most popular of these heuristics, and it is defined as in Eq. (2.5), where R' represents a specialization/generalization of the rule R. In Eq. (2.5), the logarithm of the rule confidence is also known as the information content of the rule, and it can also be used as a heuristic function by itself. The information gain measure is used by the PRISM [22] and Ripper [33] algorithms.

$$infoGain(R) = -log(confidence(R)) - log(confidence(R')) \tag{2.5}$$

In [60, 59], Furnkranz and Flach analyzed the effect of commonly used heuristic functions in the PN-space, and found that many of them are equivalent. They concluded that there are two basic prototypes of heuristics, namely precision and the cost-weighted difference between positive and negative examples. Precision is equivalent to the confidence measure described in Eq. (2.1), and the cost-weighted difference is defined in Eq. (2.6), where d is an arbitrary cost.

$$costWeigth = p - dn \tag{2.6}$$

They also interpreted the Laplace estimation and the m-estimate as a trade-off between these two basic heuristics, and again recognized their success due to their smaller sensitivity to noise.

There is a last property desired in discovered rules but not often considered: interestingness, in the sense that a rule should also be novel and surprising to the user. This is very difficult to measure, but as shown in Tsumoto [138], it is very desirable in practice when the rules will be analyzed by the user. He demonstrated that from 29,050 rules found by a rule induction algorithm only 220 were considered interesting by a user. When measuring rule interestingness, two approaches can be followed: a user-driven approach or a data-driven approach.

The user-driven approach uses the user's background knowledge about the problem to evaluate rules. In work done by [127], for example, the authors used the general impressions of the users about the problem in the form of *if-then* rules.

The impressions were matched with the discovered rules in order to find, for example, rules with the same antecedent and different consequents from the general impressions, and therefore surprising rules (in the sense that they contradict some general impressions of the user).

In contrast, data-driven approaches measure interestingness based on statistical properties of the rules, in principle without using the user's background knowledge. A review of data-driven approaches can be found in [69]. Measuring the interestingness of rules to the user in an effective way without the need for the user background knowledge sounds appealing at first glance. Nonetheless, it is important to note that a couple of studies tried to calculate the correlation between the value of these data-driven rule interestingness measures and the real subjective user interest on the rules, and they suggest this correlation is relatively low [20, 107]. In any case, the topic of rule interestingness is an active research area by itself, and it is out of the scope of this thesis.

At last, it is interesting to point out that, intuitively, complete and incomplete rules should be evaluated using different heuristics. The reason is that, while in incomplete rules there is a strong need to cover as many positive examples as possible, a major goal of complete rules is also to cover as few negative examples as possible. Most algorithms use the same heuristic to evaluate both complete and incomplete rules. Maybe it is time to evaluate the effects of using different measures to evaluate rules in different stages of the generalization/specialization processes.

2.2.4 Pruning Methods

The first algorithms developed using the sequential covering approach searched the data for complete and consistent rule sets. It means they were looking for rules that covered all the examples in the training set (complete) and that covered no negative examples (consistent). However, real-world data sets are rarely complete and usually noisy.

Pruning methods were introduced to sequential covering algorithms to avoid over-fitting and to handle noisy data, and are divided in two categories: pre- and post-pruning. Pre-pruning methods stop the refinement of the rules before they become too specific or over-fit the data, while post-pruning methods find a complete and consistent rule or rule set, and later try to simplify it. Pre-pruning methods are implemented through the building blocks "*Rules Stopping Criterion*"

in Alg. 2.1 and "*Refine Rule Stopping Criterion*" in Alg. 2.2. Post-pruning uses the building block "*Post-process RuleSet*" in Alg. 2.1.

Pre-pruning methods include stopping a rule's refinement process when some pre-defined condition is satisfied, allowing it to cover a few negative examples. They also apply the same sort of pre-defined criterion to stop adding rules to the classification model, leaving some examples in the training set uncovered.

Along with the most common criteria applied for pre-pruning are the use of a statistical significance test (used by CN2); requiring a rule to have a minimum accuracy (or confidence, such as in IREP, where rule accuracy has to be at least equal to the accuracy of the empty rule) or associating a cutoff stopping criterion to some other heuristics.

The statistical significance test used by CN2 compares the observed class distribution among examples satisfying the rule with the expected distribution that would result if the rule had selected examples randomly. It provides a measure of distance between these two distributions. The smaller the distance, the more likely that the distribution created by the rule is due to chance.

Post-pruning methods aim to improve the learned model (rule or rule set) after it has been built. They work by removing rules or rules' conditions from the model, while preserving or improving the predictive accuracy in the training set. Among the most well-known post-pruning techniques are reduced pruning error (REP) [17] and GROW [32]. These two techniques follow the same principles. They divide the training data in two parts (grow and prune sets), learn a model using the grow set and then prune it using the prune set. Nonetheless, REP prune rules using a bottom-up approach whilst GROW uses a top-down approach. Hence, instead of removing the worst condition from the rules while the accuracy of the model remains unchanged (like REP does), GROW adds to a new empty model the best generalization of the current rules.

When comparing pre- and post-pruning techniques, each of them has its advantages and pitfalls. Though pre-pruning techniques are faster, post-pruning techniques usually produce simpler and more accurate models (with the cost of inefficiency, since some rules are learned and then simply discarded from the model). Intuitively, this is due to the fact that post-pruning has more information (the complete learned model) available to make decisions, and so it tends to be less "shortsighted" than pre-pruning. However, methods which learn very specific rules and later prune them in a different set of data often have a problem related to the size of the data. If the amount of training data is limited, dividing it in

two subsets can have a negative effect since the rule induction algorithm may not get statistical support from the data when finding or pruning rules.

In any case, pruning complete rule sets is not as straightforward as pruning decision trees. Considering that most of the rule pruning literature was borrowed from the tree pruning literature [124], it is necessary to keep in mind that pruning a subtree always keeps full coverage of the data set, while pruning rules can leave currently covered examples uncovered, and the algorithm may have no resources to reverse this situation.

In an attempt to solve the problems caused by pre- and post-pruning techniques, some methods combine or integrate them to get the best of both worlds. Cohen [32], for example, combined the minimum description length criterion (to produce an initial simpler model) with the GROW algorithm.

I-REP [61] and its improved version, Ripper [33], are good examples of integrated approaches. Their rule pruning techniques follow the same principles of REP [17], but they prune each rule right after it is created, instead of waiting for the complete model to be generated. After one rule is produced, the covered examples are excluded from both the grow and prune sets, and the remaining examples are re-divided in two new grow and prune sets.

The main differences between I-REP and Ripper lie on Ripper's optimization process –which is absent in I-REP, and on the heuristics used for pruning rules and stopping adding rules to the rule set. The optimization process considers each rule in the current rule set in turn, and creates two alternative rules from them: a replacement rule and a revision rule [33]. After that, a decision is made on whether the model should keep the original rule, the replacement or the revision rule based on the minimum description length criterion.

If at the end of the optimization process there are still positive examples in the training set which are not covered by any of the rules, the algorithm can be applied again to find new rules which will cover the remaining positive uncovered examples.

2.3 Evolving Rules with Evolutionary Algorithms

The global search and problem independency of evolution techniques made their application successful in many diverse domains, including rule induction [39, 52,

148]. In this section a brief introduction to evolutionary algorithms (particularly genetic algorithms (GA) and genetic programming (GP)) is given, followed by the description of some GAs and GPs developed for rule induction.

2.3.1 Introduction to Evolutionary Algorithms

Evolutionary Algorithms (EAs) are stochastic search methods inspired by the Darwinian concepts of evolution and survival of the fittest. They became very popular in many kinds of problems, like function optimization and several machine learning tasks, due to their domain-independent nature and their robust global search mechanism - with its associated implicit parallelism and noise tolerance [10, 63].

In essence, an EA evolves a population of individuals, where each individual is a candidate solution for the target problem. At each generation, the individuals are evaluated according to a fitness function. The best individuals are selected to reproduce, and undergo crossover and mutation procedures in order to produce new offspring (new candidate solutions) that inherit some features from their parents. The evolutionary process is iteratively performed until a stopping criterion is satisfied, such as a maximum number of generations is reached or an optimal solution is found.

According to the representation of their individuals and the evolutionary operators they use, EAs can be generally classified as genetic algorithms, genetic programming, evolutionary strategies and evolutionary programming [10]. In this work we are particularly interested in genetic algorithms (GA) and genetic programming (GP), which are the kinds of EA most used for rule induction. Explaining the difference between these two approaches is not an easy task.

When the GP era started, GPs were said to be a variation of GAs where individuals represented computer programs. Hence, while GAs find a solution for a specific instance of a problem (e.g., finding a (near) optimal tour for an instance of the Traveling Salesman Problem - TSP), GPs are supposed to find a general solution for the problem (e.g, finding a good algorithm for solving any instance of the TSP). In addition, while GAs used a fixed-length string to encode individuals, GPs' individuals were represented by a variable sized tree. This definition was accepted by some years, but as research developed the terms GA and GP became almost indistinguishable. This happened because most GPs did not really evolve programs (for instance, to the best of our knowledge there is no GP that can

evolve a generic algorithm for solving any instance of the TSP problem), and they can also use a fixed-length string to encode their individuals.

In [149], Woodward argued that indeed the representation used by GAs and GPs cannot be pointed out as the main difference between them. Instead he concluded that the difference lies in the interpretation of the representation. In a GA the mapping between the description and the object being described is one to one or many to one. In contrast, in a GP the mapping is always a many to one (e.g., in a regression the same function can be expressed in many ways depending on the function and terminals sets) and non uniform.

The next section presents some examples of systems that used GAs and GPs for rule induction. Discerning GA from GP in the context of rule induction is even more complicated then in the context of conventional optimization tasks. Both GAs and GPs have been used to evolve sets of rules for specific data sets (current GPs do not generate a *generic* rule induction *algorithm*), and an analysis of the interpretation of the representation, as suggested by Woodward [149], is also not valid. This is because in the rule induction problem both GAs and GPs use a many-to-one mapping, since a concept (class) C can be described by different predictor attributes, combined in different ways and associated to distinct rule sets. Hence, they will be contrasted using their actual individual representation.

2.3.2 GAs and GPs for Rule Induction

There is a huge literature dedicated to GAs and GPs designed to solve the rule induction task. Here we will concentrate on how GAs and GPs implement the same four basic components present in sequential covering algorithms, previously pointed out as: (1) the representation of the candidate rules, (2) the search mechanisms used to explore the space of candidate rules, (3) the way the candidate rules are evaluated and (4) rule pruning.

However, before studying these basic components, we will explore the differences between the individual representations used by GAs and GPs for rule induction, and show how they indirectly determine the representation used by the candidate rules. Next, we will present how the EA's operators and the fitness function control the search and evaluation of rules, and how pruning methods can be introduced.

GA *versus* GP representation

GA methods for rule induction can be broadly classified in two approaches, according to the way the individuals encode the rules: the Pittsburgh and the Michigan approaches [52]. In the Pittsburgh approach, a single GA individual represents an entire set of rules. In the Michigan approach, each individual represents a single rule. In the latter case, in general either the GA has to be run many times in order to generate a rule set, or many rules in the population have to be combined to create a rule set. Regardless of the approach followed, the individuals are represented using a fixed or variable length string, which in turn can be encoded using a low (binary) or high level representation, as explained later.

GPs' individuals are typically represented by a tree (although other representations such as a linear representation can also be used). In the majority of the methods, each tree (individual) represents a set of rules for a target class C, and the GP is run as many times as the number of classes present in the data [35] - since each run discovers a rule set for a distinct class. Some methods also propose a multi-tree representation of individuals [101] - where each tree encodes the rule set for a distinct class, so that rules for all classes are discovered in just one algorithm run.

In the GA representation, an individual usually represents only the antecedent of the rule, while the consequent is set to the class of the majority of the covered examples. Hence, the class predicted by a rule is (re)-set every time a rule is created or modified. Including the class in the individuals' genotype is not recommended because the fitness of the rule is totally dependent on the consequent value, and a rule with a good antecedent but the wrong consequent would lead to a low fitness and consequently a low chance of surviving the selection process.

In GAs, the rule antecedent encoding strongly depends on the type of data we are producing rules for. Nominal (symbolic) and numerical (continuous) attributes can be better encoded in different ways. When using a low level encoding, for instance, symbolic attributes are assigned a bit for each of the values they can take on, as used in [80]. Thus an attribute nationality which can take on the values "American, European, Asian, African" is represented by four bits, and the string 1001 represents the antecedent "nationality = American or African". Using this same representation, numerical attributes could have their current values translated into binary code. However, for data sets with many attributes this representation can lead to very long strings, making the GA less efficient. In contrast, when using a high level encoding, attributes have their values directly

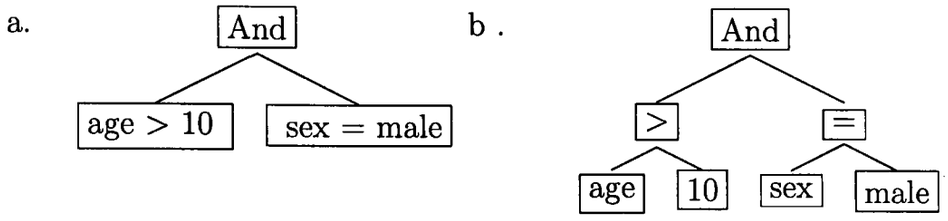

Figure 2.1: GP tree using (a) booleanized attributes and (b) non-booleanized attributes

encoded in the genome, such as in [10].

Besides using low or high level encoding, a rule can be represented by a fixed or a variable number of conditions. Fixed length representations usually have size a (where a is the number of attributes in the data set), and an active bit setting a particular rule condition as active (used by the rule) or not. It facilitates the use of crossover and mutation operators since the conditions in a given position mutated or swapped during crossover are guaranteed to represent the same attributes in both the parents and the offspring. Variable-length representations have sizes varying from 1 to a, and parent individuals using this kind of representation often need to be aligned before evolutionary operators are applied.

Regarding GPs, rules are created using a set of functions (like AND, OR, >, <, =) and terminals (represented by the data set attributes and values). Non-terminals represent the root and internal nodes in the tree, while terminals represent the leaf nodes. GPs' major problem when looking for rules is related to what is called the closure property [12]. The closure property states that the output of a node in the tree has to be valid as an input for the next node in the tree. However, nominal and continuous attributes use different operators (a rule condition like "IF nationality > American" is not valid), and the algorithm has to address this problem.

There are two conventional ways of dealing with the closure property in the context of rule induction. The first one is to "booleanize" the attributes [52], in other words to transform all the attributes into boolean ones, which allows the use of logical connectives (e.g. AND, OR) as functions. In this case all the GP's terminals are represented by complete conditions such as "*age* > *10*", in contrast with having the symbols "*age*" and "10" in the terminal set and the symbol ">" in the function set. Figure 2.1 illustrates these differences.

The second way of addressing the closure property problem is to use a strongly-typed GP (STGP) [84] or a Grammar-based GP (GGP) [148]. Strongly-typed

GP and Grammar-based GP are special kinds of algorithms in which the closure problem is tackled by imposing constraints when creating and/or modifying the individuals. In the basic STGP [100], a modified version of the original STGP, the user associates a type with each terminal, and then specifies a data type for each of the arguments and the output of the functions. For example, we associate the type nominal to the attribute *sex* and its values, an the type real to the attribute *age* and its values. We then define that the function "=" accepts nominal values as its two inputs and outputs a boolean value, while ">" accepts two real inputs and also outputs a boolean value. This solves the problem of creating invalid rule conditions like "nationality > American", because the function ">" would not accept nominal values as inputs.

In grammar-based GP, the individuals are created following a grammar, which describes the valid operations. A more detailed description of methods for rule induction following this approach can be found in Section 3.3.

According to the individual representations given above, we can conclude that in the case of GAs the rules use a traditional representation. The user specifies which kinds of relational operators are accepted, internal disjunctions can be easily represented by the lower level encoding and the class predicted by the rule can change dynamically according to the number of positive examples covered. In the case of GP, the rules' representation is determined by the function and terminal sets, which can include a large variety of symbols and operators. Thus GP can generate rules using very expressive representations, associating attributes using relational and arithmetic operators, such as "IF ((wage - monthlySpents < 2 * acountBalance) AND (age > 45)) THEN giveLoan". Although most of the EAs developed for rule induction uses propositional-logic [19], there are examples of EAs creating first-order rules (such as in [67]) and fuzzy rules (like in [43]).

EA Operators and Fitness Function

The previous section described how GAs and GPs represent a set of rules. In this section we explain how EA operators search for and prune rules, and how the fitness function evaluates them.

EAs' search is guided by evolutionary operators. Crossover and mutation are the most common ones, and can be used to generalize or specialize a rule. Crossover selects two individuals and exchanges parts of them, while mutation replaces part of an individual by a randomly generated part. When crossover and

mutation generalize or specialize a rule, the generalization operation is equivalent to an OR logical operator, and the specialization to an AND operator. For instance, consider two individuals Ind_1 and Ind_2, represented by the conditions "$age > 25$" and "$age < 50$". Applying a specialization crossover to these conditions would restrict the *age* to a range varying, say, from 30 to 50, creating the condition "$30 < age < 50$". A generalization crossover, in turn, could set *age* to any value, it means, the *age* would not matter anymore in the new rule.

There are also other operators specially designed for generalizing/specializing rules, such as the drop condition operator. It randomly selects a condition to be excluded from the rule and removes it, as shown in [80]. It is important to notice that, when choosing which operator to apply to a rule, we have to consider how general or specific the rule is. It is not worth, for example, to specialize even more a rule that covers just one example. EAs usually adjust the probabilities of applying generalization or specialization operators according to the total number of positive and negative examples covered by a rule.

In EAs, the population initialization methods establish the regions of the space where the search starts. The majority of EAs for rule induction do not initialize their population entirely at random because this can create rules that do not cover any examples in the training set. Initialization methods include seeding (choosing an example from the training set to represent a rule) followed by specialization steps, such as in [93], or creating the first population just with very general rules, composed by one or two conditions.

During EAs' search, a fitness function assesses the quality of the individuals in the population, and it is used to select the individuals which will form the new population. When an individual represents a complete rule set (as when using the Pittsburgh approach), the entire rule set is evaluated, so heuristics which take into account the performance of the entire set, like accuracy or sensitivity/specificity, are used. When an individual represents a single rule, we can use pretty much the same evaluation heuristics used by sequential covering rule induction algorithms. They are based on a combination of the information obtained from the number of covered (positive and negative) examples in relation to the total number of examples in the data set [141], and some of them also incorporate a measure of rule complexity (usually obtained from the size or depth of the tree which represents an individual), like in [35].

Pruning, the last element of sequential covering algorithms, is performed in

EAs using special operators, like the previously introduced drop condition. Although we can consider that drop condition performs a kind of pruning, its main drawback is to choose at random the condition to be excluded from the rule. Some more sophisticated operators which try to remove from the rule irrelevant conditions were implemented and are used in [93, 19].

2.4 Extracting Rules from other Knowledge Representations

Rules are in general considered an appealing knowledge representation because of their simplicity and interpretability. However, in certain domains the accuracy obtained by methods using a less comprehensible knowledge representation outperforms the accuracy obtained by models based on rules. But just producing models with high accuracy and no interpretability is not enough from a data mining point of view [49], so methods to extract rules from neural networks [77], support vector machines [106] and decision trees [124] have been proposed.

Artificial Neural networks (ANN) are models inspired by the biological neural system and its capabilities of processing information and learning. They are composed by a number of highly interconnected processing elements (neurons) which work in parallel and learn from experience. Their success is explained by their robust nature - which makes them noise tolerant, and their distributed processing approach - which makes them fault tolerant. Nevertheless, their main disadvantage lies in the black-box model they produce. In general an ANN is not able to explain the decision process used to classify an example as belonging to class C_1 or C_2. This problem prevents their application in safety-critical tasks and other kinds of task (e.g. medical diagnosis) where an understanding of the decision making process is crucial.

In order to overcome this main drawback of ANN, in the early nineties a variety of methods to extract rules from ANN were proposed. It is very difficult to categorize and/or generalize these methods because for different ANN architectures different rule extracting algorithms might be applied. Since there are so many different algorithms, and since ANNs are not the focus of this thesis, we are not going into detail about them. Instead, we are going to concentrate on the broad differences among them.

In one of the first survey papers about rule extraction algorithms (focusing only

on feed-forward multilayered ANN), a taxonomy of such algorithms was proposed [7]. It is based on five criteria:

1. Expressiveness power of the extracted rules: This criterion is related to the rule representation element present in sequential covering algorithms and EAs. It classifies a rule as being represented by propositional, first-order, fuzzy or nonconventional logic.

2. Translucency: This one refers to how the rule extraction algorithm "sees" the ANN when extracting rules from it, and is classified in three approaches. The first approach is called decompositional, and it produces rules at the level of single ANN units (neurons) and then combines them in a rule set. The second approach is called pedagogical. In this case, the ANN is considered a black-box, and the algorithm extracts rules by mapping the inputs to the outputs using a symbolic learning algorithm. The last approach is the eclectic, and combines the two previously described ones. In a later paper [136], a fourth approach was introduced to incorporate algorithms that extract rules from recurrent neural nets, and it is called compositional. It addresses algorithms which analyze the ANN units all together.

3. Portability: It describes how well the rule extraction algorithm can be migrated from one ANN architecture to another.

4. Quality of the extracted rules: It is evaluated using the following criteria: (a) rule accuracy; (b) rule fidelity - how well the rules mimic the behavior of the ANN; (c) rule consistency, which means, how similar rules extracted from different networks trained to perform the same task are; (d) rule comprehensibility. Criteria (a) and (d) can be used to evaluate rules generated by any rule induction algorithm, and were previously discussed in the sections about sequential covering and EAs algorithms.

5. Algorithm complexity.

Although these criteria were proposed more than 10 years ago, and current methods keep using them, they are not suitable for all ANNs architectures. A recent survey done by Jacobson [77], considering this time recurrent ANNs, revised these criteria for that specific architecture. The revised criteria exclude translucency, for example, which is the same for all recurrent ANNs, and introduced some architecture-related ones, such as how the recurrent ANNs continuous state space is mapped into a finite set of discrete states.

Research in the last 15 years proved to be possible to extract accurate and comprehensible rules from ANNs, although this is a hard task and it is not successful in all application domains. It also showed how difficult it is to categorize these algorithms, once they usually include some architecture-driven properties. The main new challenge now is not extracting rules from ANNs, but rather extracting them from a relatively new type of classification models: support vector machines.

Support Vector Machines (SVMs) [132] are methods that build classifiers by constructing hyper-planes in a n-dimensional space, it means, by drawing "lines" in the n-dimensional space which are able to separate examples from different classes. Drawing these lines is not complicated when we are faced with linear problems. However, most of the real world problems deals with non-linear class boundaries.

In the case of non-linear problems, SVMs create a mapping between a set of input values (examples) and a feature space, where these initially non-linear class boundaries are made linearly separable via a transformation (or mapping) of the feature space. This mapping is done by a set of mathematical functions called kernels. After performing this mapping, SVMs use a iterative training algorithm to minimize an error function.

Methods to extract rules from SVMs are far from being as developed as the ones for ANNs. Some examples of recently proposed approaches are the works of Nunez *et al.* [106] and Fung *et al.*[55]. Nunez *et al.* use a clustering algorithm to create a prototype vector for each of the possible classes of a problem, and combine them with the support vectors using geometric methods. These geometric methods create ellipsoids in the input space, which are converted in IF-THEN rules. Fung *et al.*, in turn, use a method that generates hyperplane classifiers and later defines rules as hypercubes in a n-dimensional space.

Rule extraction from SVM is a new, promising and under-explored research area. It is very likely that, in the near future, attempts to improve these pioneering works and come up with better and more efficient methods will be made.

Until now we described the efforts of the ANN and SVM communities to extract rules from black-box approaches. The motivation for creating these algorithms was to have access to more comprehensible and simpler knowledge. The last approach we will present in this section shows a method to extract rules from decision trees. But what is the motivation to do so, if trees are also considered comprehensible models?

When comparing rules with trees, rules are in general said to have the following advantages:

- Trees may have a lot of irrelevant information because of the replication problem [61]. This problem arises because decision tree learning algorithms cannot represent overlapping rules. Consequently, in some cases, the same subtree has to be learnt many times in different points of the tree. This problem also tends to make decision trees less comprehensible and more complex than sets of rules.
- There are certain rules that cannot be easily represented in a tree structure
- Systems based on rules are easier to update

Taking these points in consideration, after creating a decision tree following the divide and conquer strategy, the tree can be transformed into a set of rules. The divide and conquer strategy [124] constructs a decision tree using a top-down, typically greedy search. It evaluates all the predictor attributes to verify how well each of them, individually, classifies the examples in the training set. The best attribute is selected as the root of the tree. For each possible value of the chosen attribute a sub-tree is generated. For each sub-tree, the next attribute is chosen considering only the examples of the training set whose attributes values satisfy the condition associated with the corresponding branch of the tree. The process is recursively repeated until a stopping criterion is satisfied, e.g., until all examples in a leaf node belong to the same class or until the number of examples in a leaf node is smaller than a user-defined threshold. The class predicted by each leaf is determined by the most common class value found among the examples at that leaf.

After the tree is generated, it is mapped into a set of rules, creating a new rule for each of the tree paths from the root to a leaf. But this mapping does not make the set of rules simpler than the complete tree. In order to simplify the set of rules, each rule is typically pruned using a greedy process. We summarize here the rule pruning process used by C4.5 [124], which is probably the most used method for extracting rules from a decision tree.

Following the individual rules' simplification process, a subset of rules is chosen to represent a specific class C. The minimum description length principle is used to guide the construction of these subsets. During this process, if the number of simplified rules is not too big, all the possible subset combinations are tried.

Otherwise a simulated annealing algorithm is used to find a good rule subset for each class C.

These rules subsets are then ordered because in the case of rule conflict (more than one rule classifying the same example in different classes) the rule that appears first in the rule set is chosen to classify it. Classes with many false positives are left to the end of the rule set, to give other rules the opportunity to correctly classify them. A default rule is also inserted to the set, and its class is set to the class which has more examples not covered by any rule. A last step then optimizes the rule set as a whole. It tries to delete rules from the rule set, and those rules whose omission reduces the number of classification errors are excluded from the set.

2.5 Summary

Rule induction is a large research area. This chapter presented some concepts which are relevant to this thesis, including the three most used approaches to induce rules from data, namely sequential covering algorithms, evolutionary algorithms and algorithms to extract rules from other knowledge representations.

In this thesis, we are particularly interested in sequential covering algorithms, and we showed how most of the algorithms proposed in the literature following this approach use the same basic algorithm to produce rule sets or rule lists. We presented an overview of the methods which can be used for searching, evaluating and pruning rules during the induction process. This will be helpful when introducing our approach to automatically evolve rule induction algorithms.

We also showed how GAs and GPs are being used to extract *rules* from specific data sets, and later we will contrast these approaches with the proposed one, where a GP produces complete rule induction *algorithms* which can be applied to any classification data set.

An overview of rule extraction methods to generate rule models from neural networks and support vector machines was also presented, and emphasized the importance of having classification models which can explain their decision making process and be understandable by humans.

Chapter 3

Genetic Programming

3.1 Introduction

Section 2.3 introduced the basic concepts of evolutionary algorithms and showed how genetic algorithms and genetic programming were previously used to automatically evolve rule sets for a particular data set. This chapter explains in more details the concepts of genetic programming, and presents one of its variations: grammar-based genetic programming [143, 147, 110].

Genetic Programming (GP) [84] is an area of evolutionary computation which aims to automatically evolve computer programs. It first became popular for evolving a population of trees representing Lisp S-expressions, where each individual of the population was a candidate solution for problems varying from symbolic regression to the invention and reinvention of circuits and controllers [85]. As research in the area of GP developed, many other representations for GP individuals were also developed, including GPs with GA-like genomes (linear genomes), individuals represented by graphs or based on grammars. Programs in a variety of languages other than Lisp were also created (including Prolog, C and Java programs), but there is still a great deal of research to be done in this area.

GPs can be easily adapted to solve different problems just by redefining its sets of terminals and functions, and by finding an appropriated fitness function. Its success is backed up by a list of 36 human-competitive solutions (see [85] for the definition of the eight criteria which make a GP-evolved solution human-competitive), where two created patentable new inventions [62].

A standard GP system has some elementary components which include the individual representation, the population initialization procedure, the calculation of the fitness function, a selection mechanism and evolutionary operators. These

elements will be discussed in Section 3.2.

Despite its success in finding good solutions for a variety of problems, one of the main drawbacks of the standard GP form is the need to satisfy a property called *closure*. The closure property requires that all the GP terminals and GP functions produce a value which can be used as a valid input by another GP function. Because of this constraint, many conventional GP systems can work with only one kind of data type. In Section 2.3.2 we showed how GPs dealt with the closure problem when creating rule induction models for specific data sets: booleanizing attributes, using a strongly typed genetic programming (STGP)[100] or a grammar-based genetic programming (GGP) [144]. Booleanizing attributes is a problem-dependent solution to avoid closure, but both STGP and GGP can be used in any other problem domain.

As explained before, the STGP created by Montana associates a data type to each GP terminal and GP function, and the population initialization process and crossover operation are restricted (a node has to return the data type expected by its parent node). This mechanism overcomes the problem of dealing with only one data type, but at the same time, as pointed out in [82], makes some zones in the search space inaccessible, due to crossover restrictions. Nonetheless, one of the most interesting observations showed in Montana's work is the fact that only 20 out of 50,000 individuals generated in the standard GP (i.e., not STGP) initial population of a multidimensional least squares problem were type consistent. This observation emphasizes the importance of a strongly-typed system.

Following the STGP, grammar-based GPs (GGPs) were also created in order to effectively cope with the constraints imposed by the closure property. As its name indicates, the main difference between a standard GP and a grammar-based GP lies in the use of a grammar. In GGPs, the sets of terminals and functions are replaced by a grammar. The grammar enforces the generation of syntactically correct individuals and, although it also restricts the explorable zones of the search space, it most importantly allows the inclusion of prior knowledge about the problem to guide the search. Section 3.3 introduces the concept of grammars and discusses the two main approaches used when combining GP and grammars: GGP with Solution-Encoding Individual and GGP with Production-Rule-Sequence-Encoding Individual. It also describes some examples of GGP systems used to create rule sets for specific data sets, and contrasts them with the systems using standard GP techniques.

3.2 Standard Genetic Programming

This section discusses the six essential components considered in the design of a standard GP system:

1. A set of functions and a set of terminals, used to create the GP first population.
2. A representation for the individuals.
3. A population initialization method.
4. A fitness function, used to measure the quality of the individuals (candidate solutions).
5. A selection method.
6. Crossover and mutation operators, which use the selected individuals to generate new offspring.

The function set and the terminal set define the primitives with which a program (an individual) in GP is built. Terminals provide a value to the system, while functions process a value already in the system [12]. The terminal set is usually composed by constants, variables and/or zero-argument functions. In turn, the function set may include boolean and arithmetical functions, conditional and/or loop statements and subroutines, among many others.

While choosing the elements that will compose the terminal and function sets, the designer has to keep in mind that they are responsible for determining the size of the GP search space. Hence, too many terminals and functions might create an unnecessarily large search space of solutions, making it difficult to find a good solution in that space. At the same time, the terminal and function sets should have enough elements to generate a good candidate solution for the problem [84].

As pointed out before, although the designer can freely choose the terminal and function sets used by the GP, these terminals and functions have to respect the closure property. Recall that the closure property states that every function in the function set has to be able to handle the values it receives as input (which can be a terminal or the output of another function). For instance, a division operator has to be modified to cope with division by zero (this is often implemented by making the operator return a given value, rather than an error, in case of division by zero).

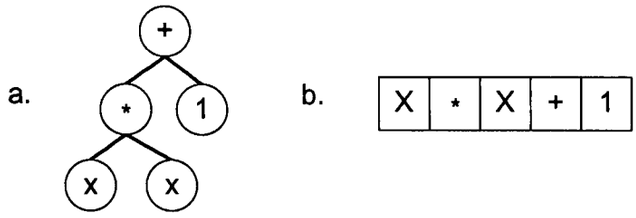

Figure 3.1: GP individual using (a) tree representation and (b) linear representation

The constraints imposed by the closure property allowed the first GP systems to deal with only one data type. Nevertheless, new GP systems were developed in order to overcome this problem, as will be explained in Section 3.3.

Once the set of functions and terminals are defined, they can be used to generate the first GP population. This population is formed by a set of individuals, where each individual represents a solution to the target problem. Hence, if we are developing a GP for discovering classification rules in a specific data set, for instance, each GP individual will represent a set of candidate rules for that data set, as explained in Section 2.3 (see Figure 2.1).

There are two main types of individual representations used in the GP literature: the first one represents an individual as a tree, and the second as a linear structure (other types of representations, such as graphs are also possible [12]). Figure 3.1 shows an example of a tree-represented and a linear-represented GP individual. Both of them encode the function $x^2 + 1$. In this example, $*$ and $+$ are functions, and x and *1* are terminals.

As observed in Figure 3.1, in the tree representation the execution of the tree is usually made in postfix order (reading the leftmost node of the tree first), while a linear representation is simply a sequence of commands that are executed from left to right. Nonetheless, these conventions can be changed depending on the functions included in the function set.

After choosing the set of terminals and functions, and an individual representation, the next step into the implementation of a GP system is to create the first population. In the context of the very popular tree representation, there are two methods commonly used [84], named grow and full. The grow method creates individuals (with a pre-defined maximum depth) by selecting elements belonging to both the function and the terminal sets (except by the root node, which is selected from the function set). This selection scheme end up producing trees with very diverse shapes and depths.

The full method, in turn, generates trees by selecting only elements from the function set, until the maximum depth of the tree is reached. At this point, it selects elements only from the terminal set to finalize the tree generation process. As a result of this initialization process, the trees produced have all the same depth.

Nevertheless, in order to maintain diversity in the population of individuals generated, a method combining the two just described methods was invented. It is called ramped-half-and-half, and works as follows. Given the maximum depth M of a tree, the population is divided into equally distributed groups of individuals which will be initialized with a maximum depth varying from 2 to M. Within each of the created groups, half of the individuals is generated using the grow and half using the full initialization technique.

At the end of the population initialization process, the individuals are evaluated according to a fitness function, and the ones representing the best candidate solutions for the problem are more likely to be selected to undergo reproduction, crossover and mutation operations. Each of these operations is applied to the selected individuals according to a user-defined rate. Section 3.2.1 describes possible fitness evaluation scenarios, while Section 3.2.2 reviews selection methods and GGP operators.

After the design of the main components of the GP is completed, we still have to choose values for standard parameters like population size, number of generations, crossover and mutation rates, maximum individual size, etc. In general these parameters are very problem related, and are usually selected during preliminary experiments.

3.2.1 Fitness Function

In a GP, the fitness function is used to evaluate how well an individual solves the target problem. It is responsible for determining which individuals will reproduce and have parts of their genetic material (i.e., parts of their candidate solution) passed onto the next generation. The better the fitness of an individual, the higher the probability of that individual being selected for reproduction, crossover and mutation operations.

Together with the sets of terminals and functions, the fitness function is one of the problem-dependent components of a GP. It is well known that, in most of the real-world problems addressed by GP or other search method, the quality of

a candidate solution cannot (or at least should not) effectively be evaluated using a single measure (objective). Consider the case of evolving a set of classification rules for a given data set, for instance.

In Section 2.3.2 we mentioned that the GP fitness function usually involves a measure related to the classification accuracy of an individual (rule set) when evaluated in a given set of training examples, but this is not the only important characteristic of a good rule set. As the set of rules should be comprehensible for a human reader, comprehensibility should also be taking into account and, in an ideal world, a measure of rule interestingness should also be considered (see Section 2.2.3).

Unfortunately, many projects that should involve the simultaneous optimization of multiple objectives avoid the complexities of such optimization, and adopt the simpler approach of just weighing and combining the objectives into a single function. However, this simpler approach is not very effective in many cases due to two main reasons. First, the objectives being optimized are often conflicting with each other. Second, the objectives often represent different and non-commensurate aspects of a candidate solution's quality, so that mixing them into a single formula is not semantically meaningful.

The use of multi-objective fitness based on the Pareto optimality is becoming more and more common in evolutionary algorithms [38, 31, 30]. At the same time, the use of multi-objective optimization based on Pareto optimality is becoming more popular in machine learning in general too [53, 79]. The next section describes the concepts of multi-objective optimization and Pareto optimality, and show how they can be applied in order to calculate the fitness function of GP individuals.

The Pareto Optimality Concept

According to the Pareto's multi-objective optimization concept, when many objectives are simultaneously optimized, there is no single optimal solution. Rather, there is a set of optimal solutions, each one considering a certain trade-off among the objectives [31]. In this way, a system developed to solve this kind of problem returns a set of optimal solutions, and can be left to the user to choose the one that best solves his/her specific problem. This means that the user has the opportunity of choosing the solution that represents the best trade-off among the conflicting objectives after examining several high-quality solutions. Intuitively, this is better than forcing the user to define a single trade-off before the search is performed.

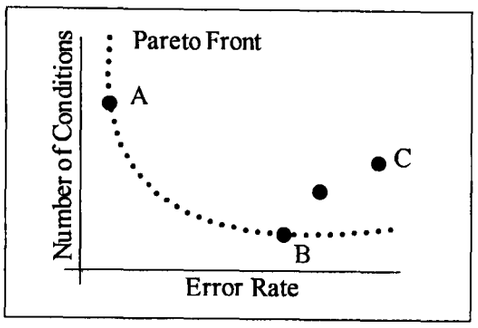

Figure 3.2: Pareto front obtained when optimizing both the error rate and the total number of rule conditions produced by a rule induction algorithm

This is what happens when the multi-objective problem is transformed in a single-objective one by assigning weights to each objective and combining all objectives into a single weighted formula. The Pareto's multi-objective optimization concept is used to find this set of optimal solutions. According to this concept, a solution S_1 dominates a solution S_2 if and only if [38]:

- Solution S_1 is not worse than solution S_2 in any of the objectives;
- Solution S_1 is strictly better than solution S_2 in at least one of the objectives.

Figure 3.2 shows an example of possible solutions found by an evolutionary algorithm when solving a rule induction problem where there are two objectives: to minimize both the error rate and the number of total conditions present in a rule set. The solutions that are not dominated by any other solutions are considered Pareto-optimal solutions, and they are represented by the dotted line in Figure 3.2.

Note that Solution A has a small error rate but a large number of rule conditions. Solution B has a large error but a small number of rule conditions. Assuming that minimizing both objectives is important, one cannot say that solution A is better than B, nor vice-versa. On the other hand, solution C is clearly not a good solution, since it is dominated, for instance, by B.

In GPs, the use of a multi-objective fitness function allows the system to return to the user the Pareto front found in the last generation of the GP or stored in an archive separated from the population. The user then chooses the best solution based on a desired trade-off between the objectives. In cases where the system does not have a direct user, an automated decision making process can choose one solution over the others. For instance, in [114], we suggested the use of a decision making criteria based on the number of individuals in the population

which are dominated by a solution in the Pareto Front. The more solutions in the population a solution in the Pareto Front dominates, the larger of the chance of it being selected as the final solution. A review of other methods for decision making in multi-objective problems can be found in [31].

3.2.2 Selection Methods and Evolutionary Operators

Following the evaluation of the GP individuals, a subset of them is selected to undergo reproduction, crossover and mutation operations. There are many selection methods used in a GP, including fitness-proportional selection, ranking selection and tournament selection [10]. Here we describe the tournament selection scheme, which will be used by the proposed GP described in Chapter 4.

The tournament selection method works by randomly obtaining k individuals from the population. These k individuals will compete against each other in a tournament. The individual with the best fitness value defeats the other individuals. The bigger the value of k, the higher the selective pressure, i.e., the faster the stronger individuals will dominate the population.

The individuals selected during the GP selection phase go through crossover, mutation or reproduction operations, according to user-defined rates. The reproduction operator simply copies the selected individual to next generation, without any alteration. Crossover swaps genetic material (parts of candidate solutions) between two individuals, whereas mutation replaces some part of the genetic material of an individual with new randomly-generated genetic material. The basic idea of crossover and mutation is as follows.

Crossover re-combines the genetic material of two parent individuals in order to produce two new children. If the individuals are represented by trees, randomly selected subtrees are swapped between the two parents. In the case of linear genomes, randomly selected linear segments of code are swapped.

Unlike crossover, mutation acts on a single parent individual of the population. It randomly selects a subtree of the tree-based genome or a gene in a linear genome and replaces it by a new randomly-generated subtree or gene.

Both crossover and mutation operations can be implemented in many ways [63]. Regardless of how they are implemented, they always have to respect the closure property, guaranteing that all the new generated individuals are valid.

In the GP community, a wide discussion about the effects of crossover operator has taken place in the literature [8, 12]. While in Genetic Algorithms (GAs)

crossover was considered a powerful tool to preserve building blocks (good segments of code), in GP its effectiveness has not been proved yet. Many researchers defend the idea that the crossover operator in GP systems is nothing more than a macro-mutation operator. According to empirical studies, crossover really reduces the fitness of the offspring relative to their parents in almost every GP system.

One of the reasons for the bad results obtained with crossover operators is related to the fact that they are context insensitive. While in GAs crossover respects homology - because each gene has a specific function - in GP crossover does not take into account the context when choosing crossover points. Consider a tree-representation GP individual, for example. A crossover point is randomly selected for each of the two individuals, and their genetic material swapped. However, a subtree that is good in the context in which it appears in individual 1 can be bad in the context in which it will appear in individual 2. This fact is not taken into account when GP crossover is performed.

In addition, researchers believe that the destructive effect of crossover is one of the factors responsible for the phenomenon of bloating [8]. Bloating is the exponential growth of introns in the individuals. An intron is defined in biology as a base sequence, found in a gene, which is not used to create proteins. With respect to artificial systems, an intron is a sequence of code that does not directly affect the survivability of the individual [12], like $x = x + 0$ or $y = y * 1$. In the first generations of the evolutionary process, introns can help building blocks to protect themselves from the destructive power of crossover. But after the first generations, they start growing exponentially (bloating), sometimes making the evolutionary process finish prematurely.

Nevertheless, current crossover operators can be improved by adding to them some "intelligence", as summarized by Banzhaf *et al* [12]. Intelligent crossover operators tend to improve the constructive power of traditional crossover over the generations. Even though the improvements obtained from intelligent crossover operators are visible, their development is still a challenge for researchers.

3.3 Grammar-based Genetic Programming

This section introduces Grammar-based GP (GGP), a variation of the standard GP described in the previous section. As the name suggests, the main difference between the standard GP approach and a grammar-based one is the presence of a grammar. In GGP systems, the set of terminals and functions is replaced by a

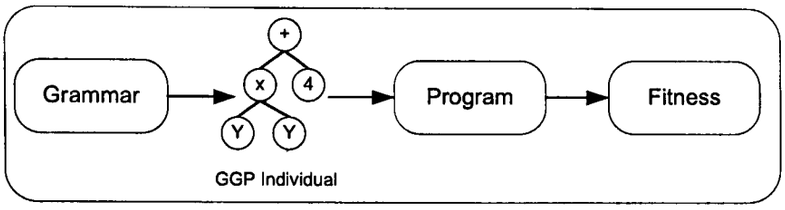

Figure 3.3: GGP scheme with solution-encoding individual representation

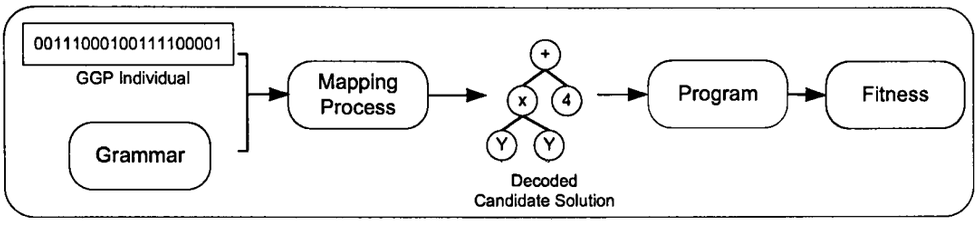

Figure 3.4: GGP scheme with production-rule-sequence-encoding solution representation

grammar. The grammar guarantees that all the individuals are syntactically correct. Note that in GGP we do not use the terms GP functions and GP terminals, but rather terminals and non-terminals, where terminals and non-terminals refer to the symbols of the grammar.

The motivation for combining grammars and GP is two-fold [110]. First, it allows the user to incorporate prior knowledge about the problem domain to help guiding the GP search. Second, it guarantees the closure property through the definition of grammar production rules.

Grammar-based genetic programming has been used in a variety of application domains. One of its first domains of application was to develop the topology of neural networks [65, 76]. It was also used in symbolic function regression [82, 110, 95], to take into account the dimension of variables when evolving physical laws [125], to perform the clustering task [36], and to evolve rule sets [147, 109, 137].

GGPs can be divided into different classes according to two different criteria:

1. The representation used by the GGP individuals.

2. The type of grammar the GGP is based on.

Considering the representation of the GGP individuals, grammar-based systems follow two different approaches: solution-encoding individual and production-rule-sequence-encoding individual, represented in Figures 3.3 and 3.4, respectively. In the first approach (Figure 3.3), there is no difference between the individuals' genotype and phenotype. An individual is represented by a tree,

which corresponds to a derivation tree produced when following the production rules of the grammar. The main characteristics of this approach and some GGP systems based on it will be presented in Section 3.3.2.

The second GGP approach (Figure 3.4) differs from the first because it uses a mapping between the individual genotype and phenotype (the search and solution space). In this approach, the individuals are represented by a linear genome (usually a binary string or an array of integers), which is generated independent from the grammar. When evaluating the individuals, a genotype/phenotype mapping is made, and the genetic material is used to select appropriate production rules from the grammar, as detailed in Section 3.3.3.

Regarding the types of grammar used to guide the GP, the most popular are the context-free grammars [4]. However, after the popularization of systems combining grammars and GP, works have been done using logic grammars [147], attribute grammars [64], tree-adjunct and tree-adjoining [81] grammars. While context-free grammars are used to restrict the syntax of the programs generated, logic grammars, attribute grammars and tree-adjoining/tree-adjunct grammars also consider context-information while generating trees (programs), and can express more complex representations. The next section briefly introduces grammars.

3.3.1 Grammars

Grammars [4] are simple mechanisms capable of representing very complex structures. Their formal definition was first given by Chomsky in 1950. According to Chomsky, a grammar can be represented by a four-tuple $\{N, T, P, S\}$, where N is a set of non-terminals, T is a set of terminals, P is a set of production rules, and S (a member of N) is the start symbol. The production rules define the language which the grammar represents by combining the grammar symbols.

Chomsky classified the grammars in four main categories, named regular (or type 3) grammars, context-free (or type-2) grammars, context-sensitive (or type 1) grammars and phrase-structure (or type 0) grammars [29]. Regular grammars are the most restrictive and also the simpler grammars, while phrase-structure grammars are the most general grammars, and also very complex. The differences between these classes of grammars is determined by the structure of the production rules they might have. For example, in regular grammars, the production rules can be presented in only two forms: a single non-terminal symbol produces a single

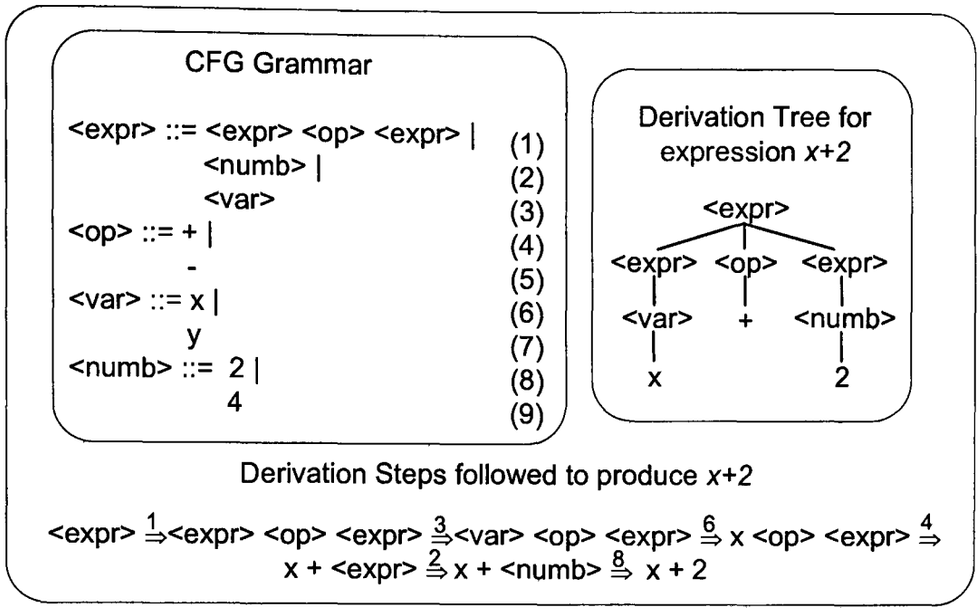

Figure 3.5: Context-free grammar to create a simple expression, showing an example of a derivation tree and the list of derivation steps followed to generate the expression $x+2$

terminal symbol; or a single non-terminal symbol produces a single non-terminal symbol followed by a terminal symbol. In contrast, in context-free grammars, a production rule is defined by a non-terminal followed by a string of terminals and/or non-terminals. Grammars belonging to the types 0 and 1 are rarely used in real applications, mainly because their implementation and parse processes are very complicated.

In this section we are specially interested in context-free grammars (CFG), which are the focus of this work. CFGs are the most commonly class of grammars used with genetic programming. Figure 3.5 shows an example of a CFG which produces simple expressions by combining the operators $+$ and -, the variables x and y and the numbers *2* and *4*.

The grammar in Figure 3.5 is described using the Backus Naur Form (BNF) [103]. When using the BNF notation, production rules have the form <expr> ::= <expr><op><expr>, and symbols wrapped in "<>" represent the non-terminals of the grammar. Three special symbols might be used for writing the production rules in BNF: "|","[ ]" and "( )". "|" represents a choice, like in <var> ::=$x|y$, where <*var*> generates the symbol x or y. "[ ]" wraps an optional symbol which may or may not be generated when applying the rule. "( )"is used to group a set of choices together, like in $x ::= k(y|z)$, where x generates k followed by y or z.

The application of a production rule from $p \in P$ to some non-terminal $n \in$

N is called a derivation step, and it is represented by the symbol $\Rightarrow$. Figure 3.5 shows the derivation steps necessary to produce the expression $x+2$. These derivation steps can be graphically represented by the derivation tree also presented in Figure 3.5.

Most of the GP systems described in the next sections are based on a CFG. This is because CFGs are very good methods to enforce the production of syntactically correct solutions. However, in some problems, information about context is essential, and this kind of information cannot be provided by a CFG. Consider, for example, a grammar which describes a language of strings having the same numbers of characters "a", "b" and "c", i.e, L = $a^n b^n c^n$. This language is context sensitive, because we need to have information about how many characters "a" have been produced in order to produce the same number of "b" and "c" characters.

The language L = $a^n b^n c^n$ cannot be generated by a CFG, but it can be produced by a context-sensitive grammar. As pointed out before, a context-sensitive grammar is complex to implement and difficult to parse. Thus, in cases like this, instead of opting for a context-sensitive grammar, we can adopt extensions for CFGs which can deal with context. In the case of GGPs, these extensions include the use of logic grammars (based on definite clause grammars) [147] and attribute grammars [76].

Logic grammars are generalizations of a CFG, where the symbols of the grammar (terminals or non-terminals) can include arguments. Arguments can be any term in the grammar, where terms can be a logical variable, a function or a constant, and are used to enforce context-dependency [148].

Attribute grammars are an extension of CFG where each symbol of the CFG can be associated with one or more attributes. Hence, when generating a derivation tree, each node in the tree is associated with a set of attributes. The values of the attributes in the derivation tree are determined through two kinds of evaluations. In the first case, the value of an attribute is inherited from its parent node (inherited attribute). In the second case, the value of an attribute is determined by the values of the attributes of its child node's attribute values (synthesized attributes). Examples of GP systems using attribute grammars can be found in [76, 28].

Finally, some GP systems were developed using tree-adjoining grammar (TAG) [81] and tree-adjunct grammars (TAGs implemented with only the adjoining operator). A TAG is similar to a CFG. However, each symbol (terminal/non-terminal)

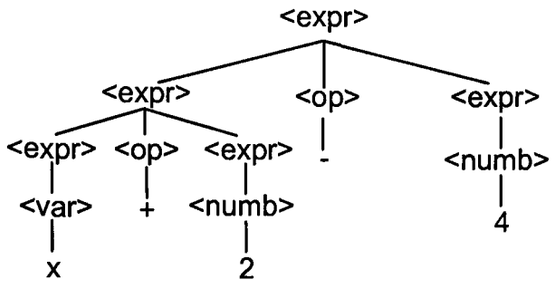

Figure 3.6: GGP tree representing the expression *x -2 (x+2-4)*

of the grammar is a tree. TAGs are classified as mildly context-sensitive grammars. According to [70], the main advantage of using TAGs and tree-adjunct grammars over CFGs is that the former implements a natural way of preserving building blocks, as each symbol in the grammar is actually a sub-tree. A TAG is represented by the quintuple $\{T, NT, I, A, S\}$, where T, NT and S stand for the terminals, non-terminals and start-symbol, as in CFGs. I and A represent the set of initial and auxiliary trees. Note that TAGs do not have production rules, and the derivation trees are created by two other trees (belonging to I and A) put together using adjoining and substitution operators. A more detailed explanation of TAGs can be found in [81], and examples of GPs using them can be found in [71, 70].

3.3.2 GGP with Solution-Encoding Individual

This section presents GGP systems following the approach described in Figure 3.3, named solution-encoding individual. All the GGP systems described in this section use a GGP individual which directly encodes the solution for the target problem, and do not require any mapping from the search (genotype) to the solution space (phenotype). This type of individual representation requires a special procedure to initialize the first population of individuals, and to control crossover and mutation operations.

Figure 3.6 shows an example of a solution-encoding individual using the context-free grammar described in Figure 3.5. The individual is built through a set of derivation steps, and production rules are applied to the tree until all the leaf nodes are represented by terminals. The solution represented by the GGP individual shown in Figure 3.6 is obtained by reading the leaf nodes of the tree, from left to right (*x+2-4*). This same procedure is used to generate all the individuals in the population. In order to guarantee a variety of individuals in the

initial population, the initial population of individuals can be initialized using the traditional ramped-half-and-half procedure, explained in Section 3.2, where trees of different shapes and sizes are produced.

Note that, when choosing a production rule from the grammar to form the individual's derivation tree, the initialization algorithm needs to check whether that particular production rule will be able to reach a terminal symbol in a number of derivation steps smaller than the maximum tree depth permitted.

Crossover and mutation operations are restricted to non-terminals, and different non-terminals might be assigned different crossover/mutation rates. In the case of crossover, a non-terminal N_x is randomly selected from the tree of the first individual I_1. After that, the system searches for the same non-terminal N_x in the tree of individual I_2. If N_x is present in I_2, the subtrees rooted at N_x in individuals I_1 and I_2 are swapped (respecting the maximum individual size parameter). If N_x is not present in I_2, the operation is not performed.

Regarding the mutation operator, a random non-terminal N_x is selected from the derivation tree of the individual, the subtree rooted at N_x is deleted, and a new subtree is created by following the productions of the grammar (starting from N_x).

The population initialization, individual representation and crossover and mutation operations just described were introduced by Whigham [143]. In his work, he describes grammars to solve the 6-multiplexer problem, and a classification problem named "greater glider density" [144]. For the latter problem, Whigham defines a set of grammars which supports *if-then-else* statements, and builds programs that can be read as a set of rules.

Furthermore, one of the most interesting features of the system created by Whigham is that the grammar is adaptable. At each generation, an analysis of the fittest individuals in the population is performed, and the grammar's production rules are modified according to the results of this evaluation.

Following Whigham, many other researchers applied GGPs with the solution-encoding individual representation. Horner [74] implemented a C++ GP library able to deal with solution-encoding individuals, which he named GPK (Genetic Programming Kernel). GPK's implementation presents similar features to the system proposed in [143]. However, its initialization process is much more complex. It requires the calculation of the search-space size, and the definition of probabilities of individuals of a specific size s (s varying from 0 to the maximum tree depth) to appear in an equally distributed population. These probabilities

are later used to set the number of individuals in the initial population which should have size s.

Indeed the population initialization process used in [74] was too complicated, as pointed out by [116], but the technique used by Whigham [143] is simple and elegant, and successfully maintain diversity in the population.

Wong and Leung [147, 148] combined GGP with ILP (Inductive Logic Programming) to produce a data mining classification system called LOGENPRO (The Logical grammar based Genetic Programming system). However, instead of a CFG, they worked with a logic grammar to create individuals. LOGENPRO can produce both decision trees and rule sets. Here we are going to describe how it was used to evolve classification rule sets.

When evolving rule sets, LOGENPRO individuals (derivation trees extracted from a logic grammar) represent classification rules (LOGENPRO follows the Michigan approach, where each individual represents a single candidate rule - see Section 2.3.2). The first population of the GGP implemented by LOGENPRO is formed by randomly generated individuals, or by individuals extracted from other learning systems. The first population can also contain individuals created by the user and given to the system.

LOGENPRO uses crossover and mutation operators in order to produce new valid individuals. The trees representing individuals might present some "frozen" nodes, which cannot be swapped during crossover operations. Crossover swaps subtrees rooted at two non-terminals to produce a single individual, and mutation recreates a subtree rooted at a random non-terminal according to the grammar. Reproduction is not used, but a third operator called drop condition (which generalizes rules) was implemented. The fitness function is based on the support-confidence framework, where each rule is associated with a confidence factor similar to the significance test used by CN2 (see Section 2.2.4).

LOGENPRO also has a mechanism called token competition, which aims to maintain population diversity. It considers each instance in the training set as a token, and the individuals compete for them. The strongest individuals in the population get to collect tokens first. After token competition, the fitness of the individuals is modified according to the number of tokens it collected. Individuals which do not collect any tokens are considered "'redundant", and are replaced by new individuals. The individuals (rules) with the best fitness are passed to the next generation without any modification, by elitism. The evaluation of LOGENPRO in UCI [104] data sets and in medical domains showed

results competitive with other rule induction algorithms.

With the same purposes of Wong, Tsakonas *et al.* [137] proposed to evolve rule sets for two medical domains using steady-state GPs and CFGs. They proposed two systems: one to evolve crisp rules and another to evolve fuzzy rules. Again, the population initialization and evolution operations are the conventional ones in GGPs with solution-encoding individual, as proposed by Whigham. In this system, each GGP individual is a rule list for a specific class. Hence, in a system with C classes, the GGP is executed C times, in each of them trying to find a rule list which separates the *i-th* class ($i = 1,...,C$) from the other *C-1* classes.

The fitness of the system to evolve crisp rules is a weighted function, which takes into account a correlation measure based on the statistics gathered in the confusion matrix and the size of the GGP trees (the size of the rule sets). In the case of the system with fuzzy rules, the simplicity of the rule sets was ignored, as the authors concluded that in the way the grammar was defined, large individuals could still be easily interpreted.

The system was compared with C4.5 [124], with a boosting algorithm, and with a standard GP for symbolic regression in two medical domains. The results showed that the proposed methods found very comprehensible rules (according to the opinion of some experts in the areas), with a classification accuracy competitive with the other methods.

Falco *et al.*[36] also implemented a GGP system to comply with a data mining task, but instead of classification they worked with clustering. They defined a new way of representing clusters via logical formulas, and used a GGP to search for these formulas. Comparisons with well-know clustering algorithms showed that GGP is a successful method to create clusters.

In a more recent work, McConaghy and Gielen [95] proposed CAFFEINE (CAnonical Form Function Expressions IN Evolution). CAFFEINE makes use of canonical form functions to evolve human-interpretable expressions for symbolic regression problems. These functions are implemented via a CFG. In [95], CAFFEINE was tested using both a GGP using a solution-encoding individual and a production-rule-sequence-encoding individual. Both implementations were evaluated in an application of knowledge extraction in analog design, and in this particular problem the GGP using the solution-encoding performed better than the GGP using the production-rule-sequence-encoding individual.

From all the systems described so far, Wong and Leung [148] is the only one which does not use CFGs. However, other attempts to work with grammars

different from the context-free ones were made. Hussain and Browse [76], for instance, proposed the use of an attribute grammar to evolve the topology of neural networks. They used attribute grammars to represent the topology of neural networks in many different levels, in a system they called NGAGE (Network Generating Attribute Grammar Encoding). In NGAGE, the GGP individuals are represented by derivation trees, where each of the leaf nodes represents a neuron in the neural net. The attributes associated with a leaf node define which of the other leaf nodes will be used to form its inputs and outputs. The internal nodes of the derivation tree define the structure of the neural net.

3.3.3 GGP with Production-Rule-Sequence-Encoding Individual

This section describes GGP systems following the production-rule-sequence-encoding individual approach illustrated in Figure 3.4. Note that, as the evolutionary algorithms described in this section use a linear genome, some authors named them as genetic algorithms, despite the fact that they are being used to evolve programs. Regardless of the name these systems receive, here we focus on the role of the grammar in them, and how the genotype-phenotype mapping process is implemented.

All the GGPs based on this approach present two common elements which distinguish them from the algorithms described in Section 3.3.2:

1. Their individuals are represented by a linear chromosome, which can be fixed or variable in length, and represented by a string of bits or integers.

2. The search space is distinct from the solution space, and each gene in a chromosome usually points to a production rule in the grammar. Hence, a mapping process is needed to generate solutions by reading the individuals' chromosomes.

As this approach works with linear genomes, no restrictions are needed in the way the initial population is created, or crossover and mutation operators are applied.

Banzhaf and Keller [11, 82] described one of the first systems using linear genomes and grammars. In their system, the fixed-length binary genotype was divided in codons, composed by a predetermined number of bits. Each codon was mapped to a symbol of the output language, and different codons could lead to

the same symbol (redundant genetic code). They argued that redundant genetic code leads to more diverse populations, as in natural systems. However, the role of the grammar in this system was to act as a correction mechanism, in the case of invalid outputs being generated when combining the output symbols.

Following the same principles of [11], Paterson and Livesey [115] proposed GADS (Genetic Algorithm for Deriving Software). GADS uses a fixed-length integer array to represent an individual, where each gene in the individual chromosome (an integer number) points to a production rule in the grammar. The main difference between the works presented in [11] and [115] is the role of the grammar in the system. While in the former the grammar is just a correction mechanism, in the latter the grammar is actually responsible for the generation of solutions.

In [115], after the population of linear individuals is initialized, a mapping process between the genotype and the phenotype of an individual starts by automatically inserting the start symbol in the root of the individual's tree, and reading the genes in the chromosome from left to right. The production rule indicated by a gene might be used or not. It is used if the non-terminal in the left-handed side of the production rule is present in the individual tree. Otherwise, it is ignored, and the next gene in the chromosome is read. The mapping process finishes when all the genes in the chromosome are read. At this point, if there are still non-terminals in the derivation tree, they are replaced by their default values. Note that each non-terminal of the grammar is associated with a default production rule.

After the works of Keller and Banzhaf [11, 82] and Paterson and Livesey [115], some other authors proposed small modifications to mitigate the drawbacks of the previous approaches (for instance, Freeman [51] improved the control of the proliferation of introns in [115] by introducing simple modifications into the mapping process). However, in 1998, Ryan *et al* [130] introduced what they called grammatical evolution (GE) [110]. In contrast with the previous approaches, GE uses a variable-length binary string genome divided in codons of 8 bits. The codons are used to select appropriate rules from the grammar, and the system has mechanisms to prevent the selection of invalid rules, avoiding the appearance of introns.

The mapping process works by translating a codon into an integer value, and then dividing this value by the number of available production rules for the non-terminal being extended. The remainder of this division is then used to select a

rule from the set of available rules. For each non-terminal, its production rules are numbered from 0 to n. Consider for example the grammar presented in Figure 3.5. The non-terminal <*op*> produces the terminals + or -. Assume the algorithm is reading the coding 00010101. This codon is translated into the integer 21, which is divided by 2 (number of production rules available having <*op*> in their antecedent). The remainder of this division is 1, and so the production rule which generates the "-" terminal is chosen. In the case that the individual runs out of codons and there are still non-terminals to be expanded, an operation called wrap starts reading the genotype of the individual again and reuse its codons.

In terms of evolution, GE uses a steady-state approach, rather than a generational approach, and the offspring replace individuals in the same population. The genetic operators are the simple ones used in a standard genetic algorithm, including one-point crossover and point mutation. A third operator, named codon duplication operator, is also introduced. In [109] we can find an example of a GE system used to evolve market index trading rules. The system evolved a set of fuzzy *if-then* rules which outperformed the baseline buy and hold strategy.

The GE approach has been applied to a variety of problems in the past years, and new variations of this system have emerged. CHORUS [9], for instance, implements a new version of the system with a position independent representation of individuals. The main difference between CHORUS and GE lies on the mapping process. In CHORUS, instead of numbering the production rules of each non-terminal independently, all the productions of the grammar are numbered together (similar to the system used in [115]). As a result, during the mapping process, the integer extracted from the chromosome is divided by the total number of rules in the grammar. In this way, one gene will always represent the same rule in the grammar, regardless of its position in the chromosome.

CHORUS also works with a concentration table, which helps selecting rules during the mapping process. The concentration table has a column for each rule in the grammar, and is used to keep track of the count (or concentration) of the rules. Every time a gene is read, it increases the concentration of the rule it represents. The algorithm stops reading the genes when the concentration of one of the applicable rules is greater than the others, and marks the position of the gene to be read later.

Apart from CHORUS, another work extends the standard GE approach to work with attribute grammars [28]. Attribute grammars, as explained before, take into account context-sensitive information, and are much more powerful than

CFGs. This particular work implements a GE system with attribute grammars to solve the knapsack problem. For more details the reader is referred to [28].

3.4 Summary

This chapter introduced the concepts of standard genetic programming and grammar-based genetic programming (GGP). GGP offers two advantages over the standard GP approach: (1) it overcomes the problems of the closure property, and (2) it allows the user to insert into the GP previous knowledge about the target problem to guide the search.

This chapter also described the two main approaches followed when using GGP systems: solution-encoding individual and production-rule-sequence-encoding individual. We presented the main differences between these two approaches, which lie mainly in the fact that while in the former there is no difference between the genotype and phenotype of the individuals, in the latter a mapping process is required to transform an individual's genotype into its phenotype.

An overview of the types of grammars commonly used in GGP systems was presented, and some examples of GGP systems which were created to evolve rules were described. In contrast with the standard GP systems described in Section 2.3.2, which usually use attributes with a single data type to deal with the problems of closure, the GGPs in this chapter use a grammar to define the types of valid rules.

Chapter 4

Automatically Evolving Rule Induction Algorithms

4.1 Introduction

Chapters 2 and 3 introduced the main concepts of rule induction algorithms and grammar-based genetic programming (GGP), describing examples of how conventional GPs and GGPs evolve rule sets for specific data sets. This chapter proposes the use of GGPs to automatically evolve rule induction *algorithms* instead of rule sets. As it will be shown later, the proposed GGP can be used to generate very robust rule induction algorithms or algorithms with performance tailored to a specific application domain (data set). The "nature" of the rule induction algorithms generated depends only on the data the GGP is trained with [112, 113].

The proposed GGP is based on the solution-encoding individual approach (see Section 3.3.2), where the grammar is used to create the individuals in the GGP initial population, instead of taking part in a genotype/phenotype mapping process (production-rule-sequence-encoding individual approach). To the best of our knowledge, there is no significant research which indicates that either the solution-encoding individual approach or the production-rule-sequence-encoding individual approach is superior to the other. Hence, in the absence of significant evidence in favor of any of these two approaches with respect to their effectiveness, we chose to use the solution-encoding-individual approach. This choice was based on the fact that, as this approach lacks a genotype-phenotype mapping process, there is no need to worry about how effective the mapping is or how large is the degree of epistasis (interaction among genes) at the genotype level.

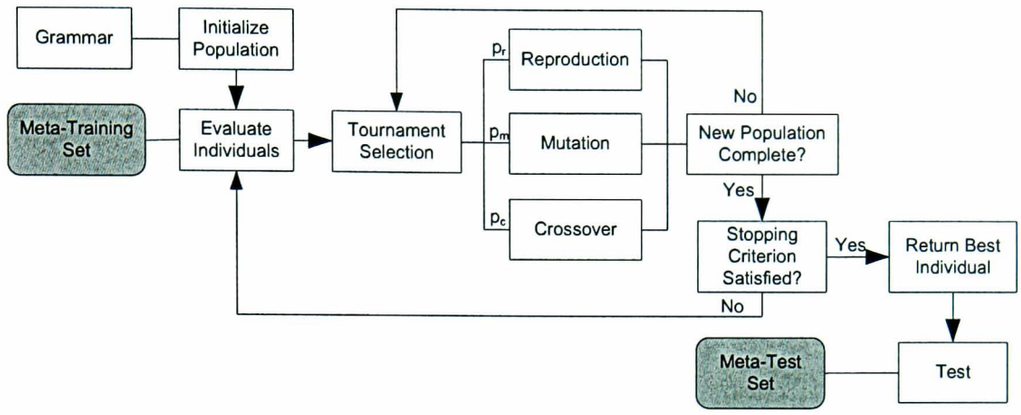

Figure 4.1: Scheme of the Grammar-based GP for Rule Induction

However, we make no claim that the solution-encoding individual approach is superior for our problem domain. Running computational experiments comparing the relative effectiveness of this approach and the production-rule-sequence-encoding approach is a topic left for further research.

Figure 4.1 shows an scheme of the proposed GGP method. The grammar contains background knowledge about the basic structure of rule induction algorithms following the sequential covering approach, and is described in Section 4.2.

Each individual in the GGP population represents a new rule induction algorithm, potentially as complex as well-known algorithms, such as CN2 [26] or PRISM [22], as explained in Section 4.3. These individuals are built by following a set of derivation steps of the grammar, as detailed in Section 4.4. The individuals (rule induction algorithms) are evaluated using a set of data sets, named meta-training set. The classification accuracies obtained from the runs of the rule induction algorithms in the meta-training set are used to generate the values of the fitness function for the individuals, as described in Section 4.5.

Following evaluation, an elitist strategy [63] selects the individual with the best fitness value, and passes it onto the new population without any modifications. Next, a tournament selection scheme with $k = 2$ (for details, see Section 3.2.2) is used to select the individuals which will produce the new population. After selection, the winners of the tournaments undergo either reproduction, mutation, or crossover operations, depending on user-defined rates, as described in Section 4.6.

The evolution process is conducted until a maximum number of generations is reached. At the end of the evolutionary process, the best individual (highest fitness produced along the run of the algorithm) is returned as the solution for the problem. The chosen rule induction algorithm is then evaluated in a new set of

data sets, named meta-test set, which contains data sets different from the data sets in the meta-training set.

4.2 The Grammar : Specifying the Building Blocks of Sequential Covering Rule Induction Algorithms

The main role of the grammar inside the GGP solution-encoding individual framework is to guarantee that all the individuals in the GGP population are valid. In order to guarantee that, the grammar is used to generate the initial population of individuals, and also controls crossover and mutation operations. In this work, we aim to automatically evolve a sequential covering rule induction algorithm. Therefore, our grammar presents all the elements which we find appropriated to use while building rule induction algorithms.

The proposed grammar is presented in Table 4.1. It uses the BNF terminology introduced in Section 3.3.1, and its *Start* symbol is represented by the non-terminal with the same name. Recall that non-terminals are wrapped into "<>" symbols, and each of them originates one or more production rules. Grammar symbols not presented between "<>" are terminals. In the context of rule induction algorithms, the set of non-terminals and terminals are divided into two subsets. The first subset includes general programming elements, like *if* statements and *for/while* loops, while the second subset includes components directly related to rule induction algorithms, such as *RefineRule* or *PruneRule*.

The non-terminals in the grammar represent high-level operations, like a while loop (*whileLoop*) or the procedure performed to refine a rule (*RefineRule*). The terminals, in turn, represent a very specific operation, like *Add1*, which adds one condition-at-a-time to a candidate rule during the rule refinement process (*RefineRule*). Terminals are always associated with a building block. A building block represents an "atomic operation" (from the grammar's viewpoint) which does not need any more refinements. Building blocks will be very useful during the phase of rule induction code generation, and are explained in Section 4.5.1.

As observed in Table 4.1, the grammar contains 26 non-terminals (NT), which originate 83 production rules. The grammar was carefully created by doing a comprehensive study of the main elements of the pseudo-codes of basic rule induction algorithms (Algorithms 2.1, 2.2 and 2.3 introduced in Chapter 2).

The first NT, *Start*, generates one of the two rule models described by Algs. 2.1 and 2.3: a rule set (*CreateRuleSet*) or a rule list (*CreateRuleList*). It also determines if the rule models generated will be post-processed or not.

NT 2 and NT 3 describe the non-terminals *CreateRuleSet* and *CreateRuleList*. They represent the outer loops described in Algs. 2.1 and 2.3. The *forEachClass* terminal allows rules to be built for each class in turn. The *whileLoop* (NT 4) keeps adding rules to the rule set/list until *condWhile* is satisfied.

condWhile (NT 5) is satisfied under one out of the two conditions: 1) When all the examples in the training set are covered by the current set of rules; or 2) When a percentage of examples in the training set or a fixed number of them is covered by the current set of rules. We used 90%, 95%, 97% and 99% when working with percentages, and 10 or 20 examples when working with the absolute numbers. While the first condition requires that rules are produced for all the examples in the training set, the second condition gives the algorithm some flexibility, and helps avoiding over-fitting.

The non-terminals *RuleSetTest* (NT 6) and *RuleListTest* (NT 7) define how the rules will be applied when classifying new instances. As explained in Section 2.2.1, the rules in a decision list can be only applied in order. However, during the creation of the list, the rules can be appended or prepended to it. The standard approach is to append rules to the list, but Webb and Brkic [140] suggested that pre-pending rules to lists generates simpler models. They argue that there are usually simple rules that cover many of the positive examples, but also a few negative examples. Leaving these simpler rules at the end of the model would allow more specific/complicated rules to handle these exceptions before the more general rules are applied.

In the case of rule sets, NT 6 defines which tie-breaking criterion will be applied in cases where two or more rules classify a test example (unseen during training) in two different classes. Among the options are the use of the *ls*-content of a rule (see Eq. (2.4)) or the Laplace estimation (see Eq. (2.2)). Note that both these measures can also be used to evaluate rules when creating them, although *ls*-content is not used for this purpose in the current version of the grammar.

The body of the *while* loop described by NT 4 implements the non-terminal *CreateOneRule* (NT 8), which represents the pseudo-code of Alg. 2.2 discussed in Chapter 2. Rules are built following three basic steps: initialization, refinement and selection – the last two steps being iterative.

The first step generates an initial rule. As shown in NT 9, a rule can be

Table 4.1: The grammar used by the GGP

```
1-  <Start> ::= (<CreateRuleSet>|<CreateRuleList>) [<PostProcess>].
2-  <CreateRuleSet> ::= forEachClass <whileLoop> endFor
                          <RuleSetTest>.
3-  <CreateRuleList> ::= <whileLoop> <RuleListTest>.
4-  <whileLoop>::= while <condWhile> <CreateOneRule> endWhile.
5-  <condWhile>::= uncoveredNotEmpty |uncoveredGreater
                  (10| 20| 90%| 95%| 97%| 99%) trainEx.
6-  <RuleSetTest> ::= lsContent |confidenceLaplace.
7-  <RuleListTest>::= appendRule | prependRule.
8-  <CreateOneRule>::= <InitializeRule> <innerWhile> [<PrePruneRule>]
                        [<RuleStoppingCriterion>].
9-  <InitializeRule> ::= emptyRule| randomExample| typicalExample |
                          <MakeFirstRule>.
10- <MakeFirstRule> ::= NumCond1| NumCond2| NumCond3| NumCond4.
11- <innerWhile> ::= while (candNotEmpty| negNotCovered)
                      <FindRule> endWhile.
12- <FindRule> ::= (<RefineRule>|<innerIf>) <EvaluateRule>
                     [<StoppingCriterion>] <SelectCandidateRules>.
13- <innerIf> ::= if <condIf> then <RefineRule> else <RefineRule>.
14- <condIf> ::=  <condIfExamples> | <condIfRule>.
15- <condIfRule> ::= ruleSizeSmaller (2| 3| 5| 7).
16- <condIfExamples> ::= numCovExp ( >| <)(90%| 95%| 99%).
17- <RefineRule> ::= <AddCond>| <RemoveCond>.
18- <AddCond> ::= Add1| Add2.
19- <RemoveCond>::= Remove1| Remove2.
20- <EvaluateRule>::= confidence | Laplace| infoContent| infoGain.
21- <StoppingCriterion> ::= MinAccuracy (0.6| 0.7| 0.8)|
                             SignificanceTest (0.1| 0.05| 0.025| 0.01).
22- <SelectCandidateRules> ::= 1CR| 2CR| 3CR| 4CR| 5CR| 8CR| 10CR.
23- <PrePruneRule> ::= (1Cond| LastCond| FinalSeqCond) <EvaluateRule>.
24- <RuleStoppingCriterion> ::= accuracyStop (0.5| 0.6| 0.7).
25- <PostProcess> ::= RemoveRule EvaluateModel| <RemoveCondRule>.
26- <RemoveCondRule> ::= (1Cond| 2Cond| FinalSeq) <EvaluateRule>.
```

initialized in four different ways: (1) with an empty antecedent (represented by the terminal *emptyRule*), (2) from a seed example (picked randomly from the training set, and represented by the terminal *randomExample*), (3) from a typical [151] training example (represented by the terminal *typicalExample*) or (4) according to the frequency of the attribute-value pairs in the data set.

The concept of typical examples is borrowed from the instance-learning based literature [151]. An example is said to be typical if it is very similar to the other examples belonging to the same class it belongs to, and not similar to the other examples belonging to other classes. In other words, a typical example has high intra-class similarity and low inter-class similarity. Eq. (4.1) shows how the typicality of an example is calculated. It is the ratio of the intra-class and inter-class similarities, where the similarity between the examples e_1 and e_2 is calculated as the complement of the distance between the examples e_1 and e_2, and P and N represent the number of positive and negative examples in the training set, respectively. The distance between the examples e_1 and e_2 is calculated as a simple Euclidean distance, as shown in Eq. (4.2). In Eq. (4.2), m represents the number of attributes in the data set, and max_i and min_i correspond to the maximum and minimum values assumed by the attribute i. In the case of nominal (or categorical) attributes, the distance with respect to a single attribute can only be 0 (if attribute values have the same value) or 1 (if attribute values have different values). If one of the attribute values is a missing value, the difference between it and any other attribute value is set to 0.5.

$$typicality(e) = \frac{(\sum_{i=1}^{P} 1 - distance(e,i))/P}{(\sum_{j=1}^{N} 1 - distance(e,j))/N} \tag{4.1}$$

where

$$distance(e_1, e_2) = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(\frac{e_{1i} - e_{2i}}{max_i - min_i})^2} \tag{4.2}$$

In the case of initializing a rule according to the frequency of the attribute-value pairs, the non-terminal *MakeFirstRule* (NT 10) creates a rule with 1, 2, 3 or 4 conditions in the rule antecedent. The pairs of attribute-values (conditions) inserted in the rule antecedent are selected using a probabilistic selection scheme, in which the probability of selecting a given attribute-value pair is proportional to the frequency of that attribute-value pair in the training set.

After the initial rule is created using one of the methods described above, it is set as the current best rule, and then the rule refinement process starts. It

is an iterative process that occurs inside the *innerWhile* loop (NT 11), which follows the non-terminal *InitializeRule* in NT 8. As described in the conditions of the *innerWhile* loop, rules can be refined until they do not cover any negative examples, as stated by the terminal *negNotCovered*, or until a set of candidate rules is not empty (*candNotEmpty*). This set of candidate rules refers to the rules which are undergoing the refinement process. At the first iteration, the only candidate rule is the one created in the initialization process. In the remaining iterations, they are the rules selected by the non-terminal *SelectCandidateRules* (NT 22), as will be explained later.

At each iteration of the *innerWhile*, the non-terminal *FindRule* (NT 12) creates a rule by: (a) finding all the possible refinements of the initial rule (*RefineRule*); (b) evaluating the rules generated through the refinements (*EvaluateRule*) and (c) selecting a fixed number of created rules to keep performing the refinement process. Moreover, *FindRule* also defines alternative ways to refine the rules (*innerIf*) and allows the definition of an alternative criterion to stop the rule refinement process (*StoppingCriterion*).

RefineRule (NT 17) changes the current candidate rules by adding (*AddCond* - NT 18) or removing (*RemoveCond* - NT 19) conditions (attribute-value pairs) to/from them. According to the production rules generated by the NT 18 and NT 19, either one or two conditions-at-a-time can be inserted in/removed from a rule.

Most of the current rule induction algorithms, as described in Section 2.2.2, use a top-down or a bottom-up search while looking for rules. Only a few of them, like SWAP-1 [142], implement a bi-directional search strategy. However, a bi-directional search might allow us to fix up rules by removing or adding conditions from/to it (specially in the case of greedy searches, where only one candidate rule goes through the refinement process). In the grammar presented in Table 4.1, the non-terminal *innerIf* (NT 13) changes the way the rules are refined according to their size (*condIfRule* – NT 15) or the number of examples covered by the current rule set/list (*condIfExamples* – NT 16). In the former case, rules having size (number of conditions) smaller than 2, 3, 5 or 7 might be refined in a different way than rules having size greater than or equal to 2, 3, 5 or 7. In the latter case, a decision is made based on the percentage of examples of the training set covered by the current rule set/list. It considers if the number of covered training examples is smaller or greater than 90%, 95% or 99% of the total number of examples in the training set.

The motivation to include this *innerIf* statement in the grammar is that the algorithm can choose, for example, to add two conditions-at-a-time to the rule while its size is small (i.e. the number of examples covered by the rule is hopefully big enough to detect attribute interaction). However, as the rule size grows, and the number of examples covered by the rule shrinks, it might be easier to improve its predictive power just by adding one condition-at-a-time instead of two. The same argument is valid for the number of examples in the training set covered by the rule set/list. The fewer the examples left uncovered, intuitively the more difficult it is to find a combination of conditions which would improve the current rule. However, even though our intuition says this might be the most appropriated way to use the *innerIf* included into the grammar, the GGP might come up with other counter-intuitive but more effective way of using it.

The rules created through the refinement process are evaluated using one of the measures defined in NT 20, namely confidence, Laplace estimation, information content or information gain. These measures take into account the number of positive and negative examples the rule covers, and they will be used, in the last phase of the rule generation process, to select a subset of rules to go through further refinements. A more detailed description of these measures can be found in Section 2.2.3.

During the rule selection phase, the rule evaluation function can be combined with some other criterion in order to select the best rules found so far. Hence, in some cases, a rule has also to fulfill a refinement stop criterion imposed by the optional non-terminal *StoppingCriterion* (NT 21). *StoppingCriterion* requires a rule to have a minimum accuracy or to be significant according to a statistical significance test during the rule selection process.

The minimum accuracy criterion calculates the accuracy (or confidence, as described in Eq. (2.1)) of the rule and compares it with a threshold. The statistical significance test – a χ^2 (chi squared) test – calculates the distance between the distribution of the classes of the examples covered by the rule and the expected distribution (given by the frequencies of examples in each class for the entire training set). The lower the value of the distance, the higher the probability that the concept represented by the rule is due to chance [27]. The numbers 0.6, 0.7 and 0.8 following the terminal *MinAccuracy* represent the thresholds used for accuracy, and the figures 0.1, 0.05, 0.025 and 0.01 after the terminal *SignificanceTest* are confidence levels for the significance test.

The rule selection process, based on the values of the evaluation function and

stopping criterion, selects a number of candidate rules to enter the next iteration of the rule refinement process. As specified in NT 22, the number of selected rules can vary from 1 to 5, or it can be 8 or 10. Regardless of the number of selected rules, the current best rule is replaced only if its evaluation function value is worse than the value of the best selected rule.

When this single rule building process terminates, a new rule is added to the rule set/list being produced. But before this happens, a last operation can be performed as a result of applying NT 8: rule pre-pruning. *PrePruneRule* (NT 23) implements a pre-pruning method that tries to simplify a rule by removing one condition or a set of conditions from its antecedent. Rules can be simplified in three ways: (1) Removing one condition-at-a-time (*1Cond*) from its antecedent, as long as the new rule is better than the original rule according to an evaluation criterion; (2) Removing the last added condition (*LastCond*) from its antecedent; or (3) Removing a sequence of conditions from the end of the rule antecedent (*FinalSeqCond*), as long as the new rule is better than the original rule according to an evaluation criterion. During the pre-pruning phase, rules are evaluated in a set of data different from the one used to build them.

Recall that rules are created until all or a large part of the examples in the training set are covered by the generated rules. However, the process of rule creation can also be halted when a condition defined by the non-terminal *RuleStoppingCriterion* (NT 24) is not satisfied. This condition is usually based on some property of the last rule found, and includes verifying if the accuracy (confidence) of the just-produced rule is greater than a threshold. We use threshold values of 0.5, 0.6 and 0.7[1].

Once the rule set/list is completed, the rule induction algorithm can still perform a last operation: post-process the rule model. The presence of a post-process step in the algorithm is determined by the application of NT 1. Post-processing methods (NT 25) can apply the same techniques used to pre-prune a single rule to all the rules in the rule model. Hence, *RemoveCondRule* (NT 26) describes similar methods to the ones described in NT 23. After the model is completed, it can be simplified by removing one (*1Cond*) or two (*2Cond*) conditions-at-a-time from the rule antecedents. In the case of rule sets, the model is simplified as long as the new rule is better than the original one according to an evaluation criterion. In the case of rule lists, this process goes on while the accuracy of the entire model

[1]The accuracy is normalized to return a number in [0...1].

is not reduced. The option of removing a final sequence of conditions is also available while post-processing rules. Besides, more compact models can be tried out by removing one by one entire rules from the current model, as implemented by the *RemoveRule* terminal in the right-hand side of the production rule generated by the NT 25. After each rule is removed from the rule set the whole model has to be re-evaluated, as indicated by the terminal *EvaluateModel*. Again, as in the pre-pruning phase, the rules/rule sets being post-processed are evaluated in a set of data different from the one used to build them.

By applying the production rules defined by the grammar, we can generate up to approximately 5 billion different rule induction algorithms (see Appendix A for a detailed calculation of the size of the search space). Each of these rule induction algorithms can be represented by an individual in the GGP population.

4.2.1 The New Rule Induction Algorithmic Components in the Grammar

As explained before, the grammar presented in Table 4.1 contains knowledge about how humans design rule induction algorithms, but it also presents some new components which, to the best of our knowledge, were not used in rule induction algorithms before.

The major new components inserted to the grammar are:

- The terminal *typicalExample*, which creates a new rule using the concept of typicality, borrowed from the instance-based learning literature.
- The non-terminal *MakeFirstRule*, which allows the first rule to be initialized with one, two, three or four attribute-value pairs, selected probabilistically from the training data in proportion to their frequency. Attribute-value pairs are selected subject to the restriction that they involve different attributes (to prevent inconsistent rules such as "sex = male AND sex = female").
- The non-terminal *innerIf*, which allows rules to be refined in different ways (e.g. adding or removing one or two conditions-at-a-time to/from the rule) according to the number of conditions they have, or the number of examples the rule list/set covers.
- Although some methods do use rule look-ahead, i.e., they do insert more than one condition-at-a-time to a set of candidate rules, we did not find in

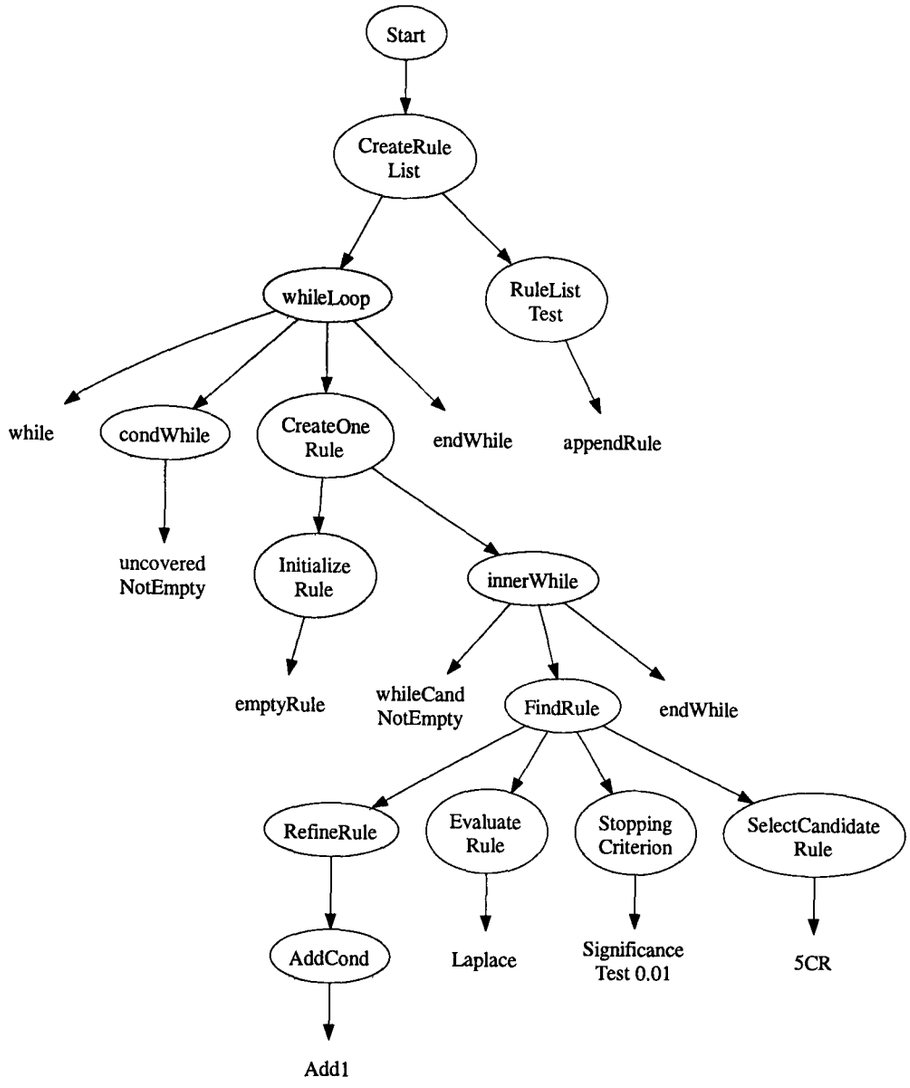

Figure 4.2: Example of a GGP Individual

the literature any rule induction algorithm which removes two conditions-at-a-time from a rule. This is implemented by the terminal *Remove2*. See [58] for a more detailed discussion on bottom-up look-ahead algorithms.

Note that the list above shows a set of single components which are new "building blocks" of rule induction algorithms. These components increase the diversity of the candidate rule induction algorithms considerably, but it is the combination of the "standard" and new components which will potentially contribute to the creation of a new rule induction algorithm different from the conventional algorithms.

4.3 Individual Representation

As pointed out before, the GGP system described in this chapter follows the solution-encoding individual approach, where each individual is represented by a derivation tree created by applying a set of production rules from the grammar.

Figure 4.2 shows an example of an GGP individual. The root of the tree is the non-terminal *Start*. The tree is then derived by the application of production rules for each non-terminal. For example, *Start* (NT 1) generates the non-terminal *CreateRuleList* (NT 3), which in turn produces the non-terminals *whileLoop* and *RuleListTest*. This process is repeated until all the leaf nodes of the tree are terminals.

Every time the application of a production rule involves an option between two or more symbols in the right-hand side of a production rule, each of the candidate symbols has the same probability of being chosen. Hence, the generation of the derivation tree representing an individual is a non-deterministic process, as usual in GP.

Recall that an individual represents a complete rule induction algorithm. Thus, in order to extract from the tree the pseudo-code of the corresponding rule induction algorithm, we read all the terminals (leaf-nodes) in the tree from left to right. The tree in Figure 4.2, for example, represents the pseudo-code described in Alg. 4.1, expressed in a high level of abstraction. This algorithm actually represents an instance of the well-known CN2 algorithm [26] producing an ordered list of rules, with the beam-width (or star, using the CN2 terminology) parameter set to 5 and the statistical significant test threshold set to 0.01.

4.4 Population Initialization

The GGP system proposed in this work generates the initial population using a method similar to the one suggested by Whigham [143]. It starts the population generation procedure by calculating the minimum tree depths for all the production rules of the grammar.

The minimum tree depth is used to guide the selection of production rules in the grammar in a way that programs with different depths are generated. Production rules which generate only terminals have minimum tree depth of 1. The minimum depth of other production rules is calculated in a bottom-up fashion. Consider the production rule <A> ::= <B> <C>. If the minimum depth of <B>

Algorithm 4.1: Pseudo-code of the rule induction algorithm represented by the derivation tree in Figure 4.2

```
RuleList = ∅
repeat
    bestRule = an empty rule
    candidateRules = ∅
    candidateRules = candidateRules ∪ bestRule
    while candidateRules ≠ ∅ do
        newCandidateRules = ∅
        for each candidateRule CR do
            Add 1 condition-at-a-time to CR
            Evaluate CR using the Laplace estimation
            if CR is significant at the 0.01 significance level then
                newCandidateRules = newCandidateRules ∪ CR
                if CR is better than bestRule then
                    bestRule = CR
        candidateRules = 5 best rules selected from newCandidateRules
    RuleList = RuleList ∪ bestRule
until all examples in the training set are covered
```

and <C> are known, the minimum depth of <A> is set as the maximum of the values of the minimum depths of <B> and <C> + 1. If the values of <B> and/or <C> are unknown, their minimum tree depths are calculated first, in a recursive manner.

In the current grammar presented in Table 4.1 the depth of the GGP trees does not vary a lot. The minimum depth of a tree is 9, and the maximum depth is 10. This is because the current grammar does not have any recursive production rules (i.e., productions like <A> ::= <B> <A> | c). Very different individuals will greatly vary in the number of tree nodes, but not in tree depth. In the current version of the system, half of the individuals in the first population are created with depth 9, and the other half with a tree depth of 10. However, the system does support recursive production rules, and in this case an equal number of individuals for each depth – with depths varying from the minimum depth of the *Start* symbol to the maximum depth parameter set by the user – would be generated. The system guarantees that all the individuals in the first population are different from each other.

An issue which has to be given attention during the population initialization process is that some combinations of non-terminals/terminals in the grammar may generate individuals which are semantically invalid. By semantically invalid

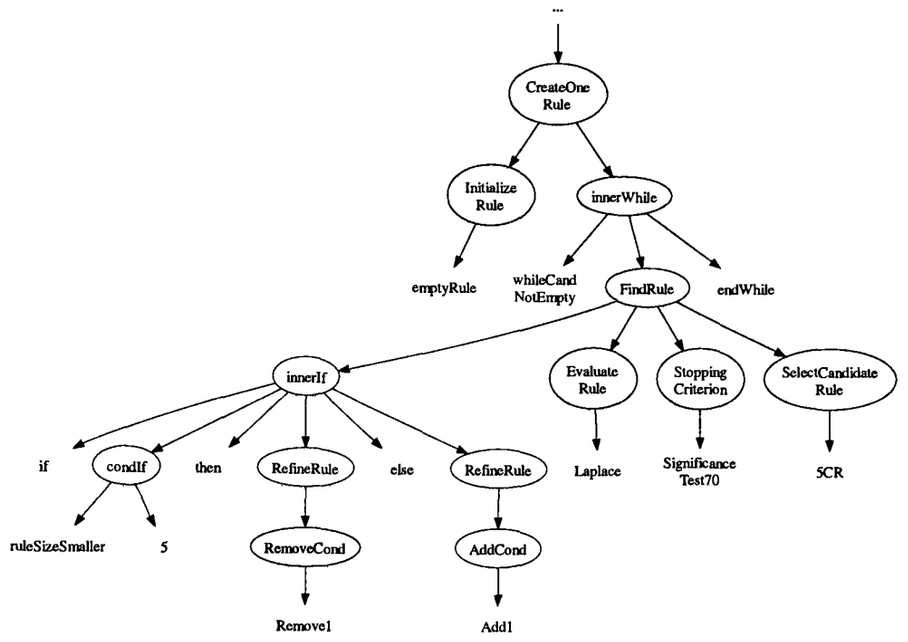

Figure 4.3: Example of an individual representing a rule induction algorithm which will lead to an infinite loop

individuals we mean that the rule induction algorithms generated by these individuals will, most of the time, execute infinite loops. Figure 4.3 shows an example of part of a simple rule induction algorithm defined by a GGP individual which would lead to an infinite loop.

The individual represented in Figure 4.3 starts producing a rule with an empty condition, and removes one condition-at-a-time from it as long as the size (number of conditions) of the produced rule is 5. As the first rule is empty, the *if* part of the *if* statement does not do anything, and the *else* part is unreachable. Because the set of candidate rules will never change, i.e., it will always have the empty rule, the condition to finish the *innerWhile* loop will never be satisfied.

Some infinite loop situations, like the one presented in Figure 4.3, can be predicted by a careful evaluation of the possible combinations of non-terminals and terminals in the grammar, and are set up as constraints during the population initialization process and breeding operations. The current version of the GGP system imposes the following constraints when creating individuals:

1. The non-terminal *AddCond* cannot be combined with the terminals *randomExample* and *typicalExample*, which may be generated by the non-terminal *InitializeRule*. This is because it does not make sense to add more conditions to a rule represented by a complete example taken from

the training set.

2. Using the same rationale, the non-terminal *RemoveCond* cannot be used in association with the terminal *emptyRule* generated by the non-terminal *AddCond*, or the terminals *numCond1* and *numCond2* generated by the non-terminal *MakeFirstRule*.

3. The non-terminal *innerIf* has to incorporate the constraints imposed by item 1, so that situations like the one presented in Figure 4.3 are avoided. The following combinations are not legal in an individual using an *innerif*:

   - When the non-terminals *condIfRule* and *RemoveCond* (in the *if* part of the *if* statement) are used together, they cannot be combined with the terminals *emptyRule*, *numCond1* or *numCond2*.
   - When the non-terminals *condIfRule* and *AddCond* (in the *else* part of the *if* statement) are used together, they cannot be combined with the terminals *randomExample* or *typicalExample*.
   - When the terminal > (<), generated by the non-terminal *condIfExamples*, is combined with *RemoveRule* (in the *else* (*if*) part of the *if* statement), they cannot be combined with the terminals *emptyRule*, *numCond1* or *numCond2*.
   - When the terminal > (<), generated by the non-terminal *condIfExamples*, is combined with *AddRule* (in the *else* (*if*) part of the *if* statement), they cannot be combined with the terminals *randomExample* or *typicalExample*.

Note that these constraints avoid the most general cases of infinite loops, but we cannot guarantee it avoids all of them. In any case, individuals generating infinite loops are penalized during the fitness evaluation process, as explained in Section 4.5, and are unlikely to survive for many generations.

Moreover, evaluations of the method used to generate the initial population showed that the constraints do decrease a lot the number of individuals executing infinite loops.

The efficiency of the population initialization process – in terms of preserving diversity in the population – can be observed in the graphs shown in Figures 4.4 through 4.7. These pictures show the distribution of some terminals during the

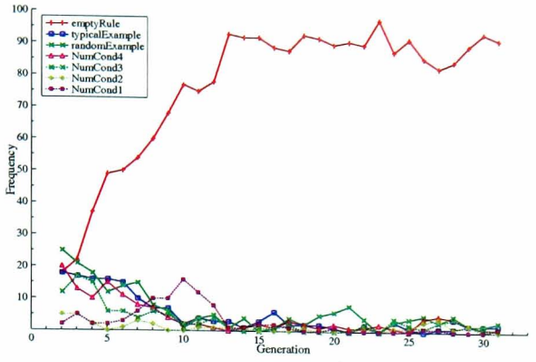

Figure 4.4: Frequencies of terminals responsible for the rule induction algorithm rule initialization process during the GGP evolution

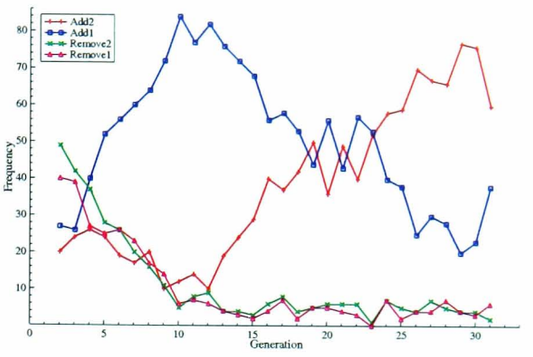

Figure 4.5: Frequencies of terminals responsible for the rule induction algorithm rule refinement process during the GGP evolution

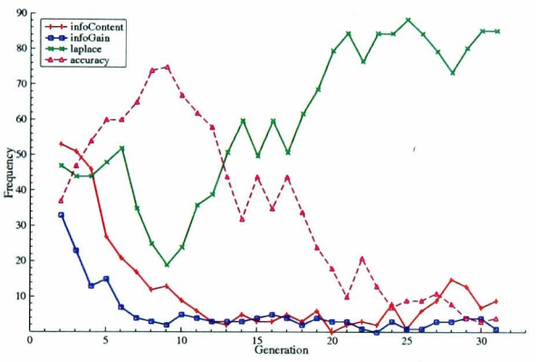

Figure 4.6: Frequencies of terminals responsible for the rule induction algorithm rule evaluation process during the GGP evolution

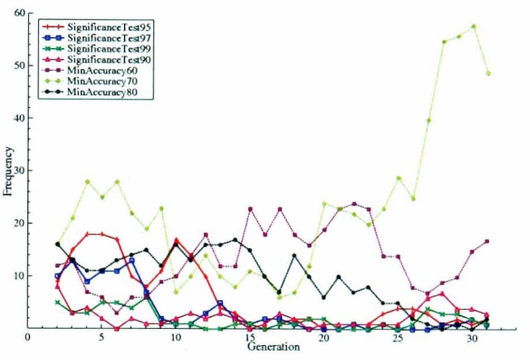

Figure 4.7: Frequencies of terminals responsible for the rule induction algorithm rule stopping criterion process during the GGP evolution

evolution of the GGP. In Figure 4.4, for instance, we can observe how the terminals generated by *InitializeRule* were distributed. Note that *InitializeRule* generates the terminals *emptyRule*, *randomExample* and *typicalExample*, and the non-terminal <*MakeFirstRule*>, which in turn generates the terminals *NumCond1*, *NumCond2*, *NumCond3* and *NumCond4*. For this particular run of the GGP, the terminal *emptyRule* can be found in 50% of the populations in generation 5. However, note that individuals containing the other terminals do not disappear.

In terms of the distribution of terminals in the first generation, there is always a balanced distribution of them in the graphs presented in Figures 4.5 through 4.7. Figure 4.5 shows a good example of how a symbol which starts dominating a population can be replaced by another during the evolution. This is what happens with *Add1* and *Add2*. However, in the case of this particular graph,

individuals containing *if* statements can present both terminals *Add1* and *Add2*. In Figure 4.6 we can also observe how the *accuracy* measure initially is more used than the *Laplace* estimation measure, but at a certain stage in evolution this situation is reversed. In Figure 4.7 we observe a more balanced distribution of terminals which determine the *StoppingCriterion*, but suddenly at generation 27 the use of a minimum accuracy of 70% appears in 60% of the population.

4.5 Individual Evaluation

An evolutionary algorithm works by selecting the fittest individuals of a population to reproduce and generate new offspring. Individuals are selected based on how good their corresponding candidate solutions are to solve the problem being tackled. In our case, we need to evaluate how good a rule induction algorithm is.

In the rule induction algorithm literature, comparing different classification algorithms is not a straightforward process. There is a variety of metrics which can be used to estimate how good a classifier is, including classification accuracy, sensitivity/specificity and ROC analysis [48] . There are studies comparing these different metrics, and showing advantages and disadvantages in using each of them [50, 18]. Nevertheless, as pointed out by Caruana and Niculescu-Mizil [18], in supervised learning there is one ideal model, and "we do not have performance metrics that will reliably assign best performance to the probabilistic true model given finite validation data".

Classification accuracy is still the most common metric used to compare classifiers, although some authors tried to show the pitfalls of using classification accuracy when evaluating induction algorithms [121] – specially because it assumes equal misclassification costs and known class distributions – and others tried to introduce ROC analysis as a more robust standard measure. Based on these facts and on the idea of using a simpler measure when first evaluating the individuals produced by the GGP, we chose to use a measure based on accuracy to compose the fitness of the GGP system.

In this framework, a rule induction algorithm RI_A is said to outperform a rule induction algorithm RI_B if RI_A has better classification accuracy in a set of classification problems. Thus, in order to evaluate the rule induction algorithms being evolved, we selected a set of classification problems, and created a meta-training set. The meta-training set consists of a set of data sets, each of them divided as usual into training and validation sets.

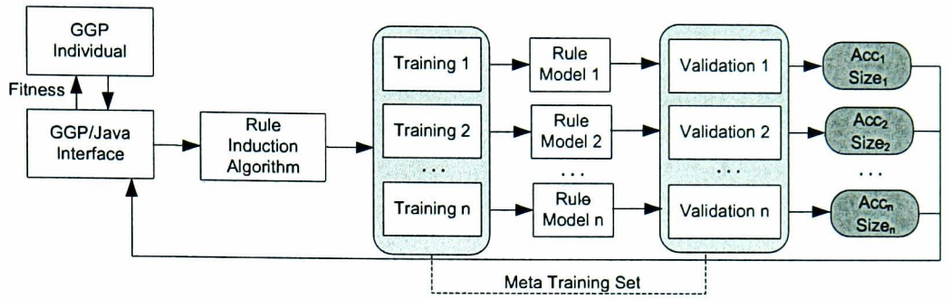

Figure 4.8: Fitness evaluation process of a GGP Individual

As illustrated in Figure 4.8, each individual in the GGP population is decoded into a rule induction algorithm using a GGP/Java interface, as will be detailed in Section 4.5.1. The Java code is then compiled, and the resulting rule induction algorithm run in all the data sets belonging to the meta-training set. It is a conventional run where, for each data set, a set or list of rules is built using the set of training examples and evaluated using the set of validation examples. After the rule induction algorithm is run in all the data sets in the meta-training set, the accuracy in the validation set and the rule lists/sets produced for all data sets are returned. These two measures can be used to calculate a fitness function. In this work, we investigated two approaches to calculate the fitness. The first one uses only the accuracy to generate the fitness value, and will be described in Section 4.5.2. The second one uses the concept of Pareto dominance discussed in Section 3.2.1 to simultaneously optimize both the accuracy and the size of the rule lists/sets, and will be described in Section 4.5.3.

4.5.1 From a Derivation Tree to Java Code

In the process of evaluating a rule induction algorithm, the derivation tree initially created by using the production rules of the grammar has to be converted into real machine code, which can be executed in a set of classification problems and generate rule models for the corresponding data sets. This process of conversion of GGP trees into code is the focus of this section, and is illustrated in Figure 4.9.

The first thing to point out in Figure 4.9 is that each terminal in the grammar is associated with a Java code, which is an implementation of the building block represented by the associated terminal. Recall that each terminal in the grammar represents a building block, that is, an "atomic" operation (at a high level of abstraction, from the point of view of the grammar) performed by a rule induction algorithm. For instance, the terminal *Add1* is associated with the building-block

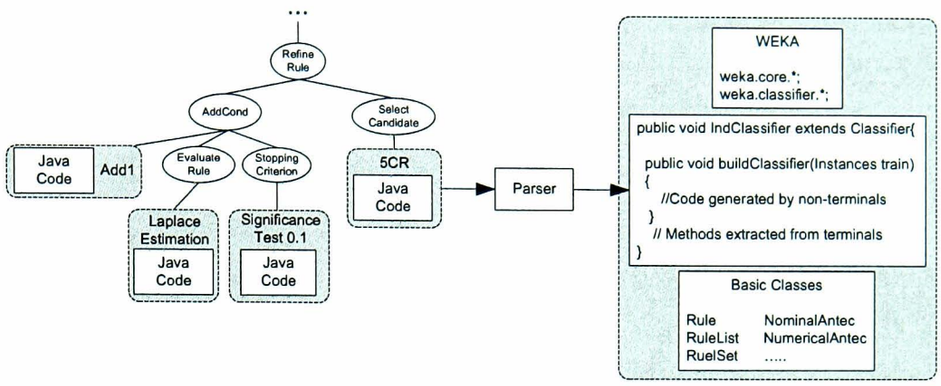

Figure 4.9: Example of the method used to extracted Java code from the GGP individuals

described in Alg. 4.2. This building block is implemented as the Java method *public List add1Condition(Rule r)*, which takes the current rule as a parameter and returns a list of possible rule refinements. The system was implemented in Java because, instead of implementing the whole system from scratch, we used WEKA [145] – an open source data mining tool written in Java – to speed up the implementation process.

The building blocks (Java codes) associated with the terminals of the grammar were implemented in two phases, with the purpose of quickly obtaining a first set of results to validate the proposed idea. In the first phase, the grammar terminals dealt only with nominal attributes. In the second phase, their implementation was extended to also handle numerical attributes. The terminals whose implementation went through major extensions in this second phase were the ones responsible for refining rules by adding/removing conditions to/from it. After the completion of this second phase, the grammar became flexible enough to produce algorithms that represent rule conditions of the form "<attribute, operator, value>", where operator is "=" in the case of nominal attributes, and operator is "$\geq$" or "$\leq$" in the case of numerical attributes.

The approach followed to generate numerical conditions is similar to the one implemented by the Ripper and C4.5 algorithms, where the values of a numerical attribute are sorted, and all threshold values considered. The best threshold value is chosen according to the information gain associated with that attribute-value pair.

As observed in Fig. 4.9, in the center of the GGP-tree/Java code conversion process there is a parser. The parser reads the symbols in the tree, and inserts the code associated with them to a Java class named *IndClassifier*. When reading

Algorithm 4.2: AddCond(Rule R)

```
refinements = ∅
for i = 0 to i < number of attributes A in the training set do
    for j = 0 to j < number of values V A_i can assume do
        newRule = R ∪ (A_i,V_j)
        refinements = refinements ∪ newRule
return refinements
```

a non-terminal in the tree, the parser might add to the method *buildClassifier* of the class *IndClassifier* either (1) some standard code, (2) a call for a method defined by a terminal or (3) take no action. We call standard code the code which is usually found in all rule induction algorithms which contain that non-terminal, despite of the terminals it generates. When reading a terminal, the parser adds to the class *IndClassifier* the method(s) associated with that terminal.

During the creation of the rule induction algorithm, there is another point which has to be taken into account: infinite loops. As explained in Section 4.4, although all the methods associated with a terminal perform the task they are supposed to, combinations of some symbols of the grammar may lead to infinite loops.

The population initialization process handles some of these invalid combinations (again, see Section 4.4), but there might be unexpected situations. As the number of non-terminals/ terminals combinations is very high, conditions to halt the run of a GGP were added to its two main loops. In the case of the *whileLoop* non-terminal, the rule production process will be stopped if the number of loop iterations reaches the same number of instances in the training set plus one (it is not worth to have a rule model with more rules than examples). For the *innerWhile* loop, the maximum number of iterations allowed is equal to the number of attributes in the data set times two (as a numerical attribute might appear more than once in a rule, we set up this upper bound as number of attributes times two).

At the end of the parse process, we have a complete Java class *IndClassifier*. This class is then associated with other basic Java classes (like *Rule*, *Antecedent*, *RuleList*, *RuleSet*, etc) and some of the WEKA packages, allowing us to run the generated rule induction algorithm in a set of classification problems. Note that although the original idea was to use as much of the WEKA code as possible, during the development of the system we realized that the implementation of the classifiers in WEKA was not modular enough to make the automatic generation

of the rule induction algorithms' code simple. Hence, the system does not use the methods implemented for specific classifiers in WEKA, but only its *core* package (which deals with the basic operations of any classifier, like reading the examples and generating data statistics) and some of the classes in the package *classifier*.

4.5.2 Single-Objective Fitness

As showed in Fig. 4.8, once we convert the GGP trees into rule induction algorithm Java codes, we run them in a set of classification problems. For each classification problem (i.e., each data set in the meta-training set), we obtain a rule model from the training set and an accuracy in the validation set, and then we need to summarize this information into a straightforward fitness value.

The most intuitive way of performing this summarization, and which was first used to evaluate the proposed GGP, was to use the average of the accuracies of the GGP-RI (the Rule Induction algorithm evolved by the GGP) over all classification problems as the fitness value. However, analyzing the results obtained in the first runs of the GGP with this fitness function, we realized it was not a good one. There is a simple explanation for that. The GGP is calculating an average over different data sets, with very different baseline accuracies (i.e., accuracies obtained when using the class of the majority of the training examples to classify new examples). For instance, let us assume we have a data set DS_1 with baseline accuracy 90% and a data set DS_2 with baseline accuracy 60%. The GGP creates two rule induction algorithms GGP-RI_1 and GGP-RI_2. GGP-RI_1 obtains accuracies of 92% and 70% for DS_1 and DS_2, while GGP-RI_2 obtains accuracies of 100% and 62% for DS_1 and DS_2. The average of the accuracies of GGP-RI_1 is the same as the average of the accuracies of GGP-RI_2, which is 81%. However, improving the accuracy from 90% to 100% is much more difficult than increasing it from 60% to 70%.

It was clear that the accuracy itself was not a good measure of how much better GGP-RI_1 was when compared to GGP-RI_2. Hence, we tried two other fitness functions. The first was based on the average over the values of the function $sens_i \times specif_i$ (described in Eq. (4.3)) for each data set i in the meta training set. Eq. (4.3) describes the $n_{classes}$-root of the product of the sensitivity (true positive rate) $\times$ the specificity (true negative rate) of a data set. The TP, FP, TN and FN terms stand for the number of true positives (positive examples correctly classified as positive), false positives (negative examples wrongly classified as positive), true

negatives (negative examples correctly classified as negative) and false negatives (positive examples wrongly classified as negative). In the case of data sets with more than two classes, when calculating the values of *TP*, *FP*, *TN* and *FN*, each class is turn is considered as the positive class, and the remaining classes as the negative class.

$$sens_i \times specif_i = \sqrt[nClasses]{\prod_{i=1}^{nClasses} \frac{TP_i}{TP_i + FN_i} \times \frac{TN_i}{TN_i + FP_i}} \tag{4.3}$$

The second fitness function was implemented as the average of the values of function fit_i (defined in Eq. (4.4)) for each data set i in the meta training set. In the definition of fit_i given in Eq. (4.4), Acc_i represents the accuracy (on the validation set) obtained by the rules discovered by the rule induction algorithm in data set i. $DefAcc_i$ represents the default accuracy (the accuracy obtained when using the class of the majority of the training examples to classify new examples in the validation set) in data set i.

$$fit_i = \begin{cases} \frac{Acc_i - DefAcc_i}{1 - DefAcc_i}, & \text{if } \mathrm{Acc}_i > \mathrm{DefAcc}_i \\ \frac{Acc_i - DefAcc_i}{DefAcc_i}, & \text{otherwise} \end{cases} \tag{4.4}$$

According to the definition of fit_i, if the accuracy obtained by the classifier is better than the default accuracy, the improvement over the default accuracy is normalized, by dividing the absolute value of the improvement by the maximum possible improvement. In the case of a drop in the accuracy with respect to the default accuracy, this difference is normalized by dividing the negative value of the difference by the maximum possible drop (the value of $DefAcc_i$).

Hence, fit_i returns a value between -1 (when $Acc_i = 0$) and 1 (when $Acc_i = 1$). As explained before, the degree of difficulty of the classification task depends strongly on the value of $DefAcc_i$. The above fitness function recognizes this and returns a positive value of fit_i when $Acc_i > DefAcc_i$. For instance, if $DefAcc_i$= 0.95, then Acc_i=0.90 would lead to a negative value of fit_i, as it should.

A set of experiments was performed using both the fitness functions defined in Equations (4.3) and (4.4). The graph in Figure 4.10 shows the average values for both the fitness functions defined in Equations (4.3) and (4.4) over all the 100 individuals of the GGP evolved for 30 generations. The graph was produced in experiments where the meta-training set contained the data sets *monks-2*, *monks-3*, *balance-scale*, *lymph* and *zoo*, obtained from [104]. As we can observe, the curve

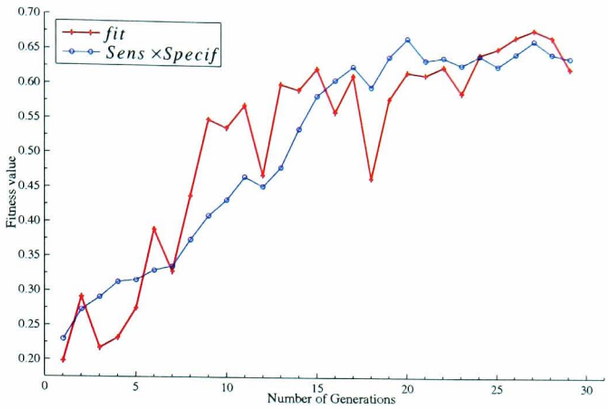

Figure 4.10: Comparing the average fitness values given by fit and $sens \times specif$ in a GGP run with 100 individuals and 30 generations

generated by the average value of $sens_i \times specif_i$ ($Sens \times Specif$) over all data sets i is smoother than the average fit_i curve (*fit*), but in general the behavior of both measures is similar. While the improvements of fit along the generations grows faster than the $sens \times specif$, the same is true for the declines.

Although the graph shows that in general the behavior of the two measures is consistent, the rule induction algorithms evolved by GGPs using fit were more robust when evaluated in a meta-test set containing data sets different from the ones used in the meta-training set. Hence, based on these empirical results, the average value of fit_i over all the data sets (fit_i being defined in Eq. (4.4)) was chosen as the current GGP fitness function. More precisely, the fitness is computed as $\sum_{i=1}^{nDataSets} fit_i/nDataSets$, where $nDataSets$ is the number of data sets in the meta-training set.

Recall that some of the rule induction algorithms generated by the GGP can generate algorithms whose runs will enter into infinite loops. As mentioned earlier, during the development of the GGP system, we tried to predict most of the situations in which infinite loops might occur. However, there are still some programs which will be halted when an infinite loop is detected. These programs have to be penalized in their fitness function, so that they have a very small chance of breeding.

We choose to penalize programs which enter into infinite loops in a particular data set by setting their fitness in that data set to *minus* the number of data sets in the meta-training set. In this way, the fitness value in that data set has a big impact in the value of the average fitness over all the data sets in the meta-training set.

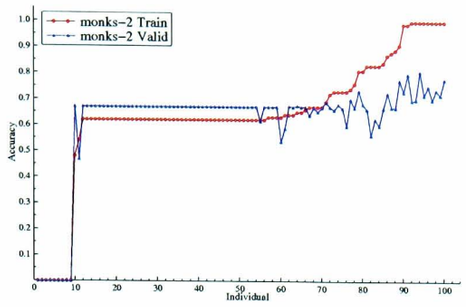

Figure 4.11: Classification accuracies for data set *monks-2* in Generation 1

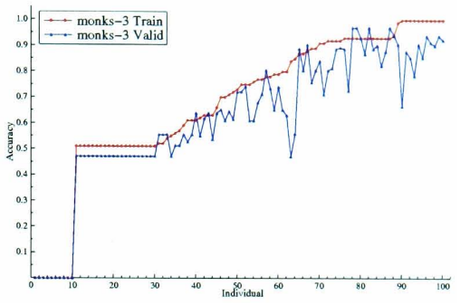

Figure 4.12: Classification accuracies for data set *monks-3* Generation 1

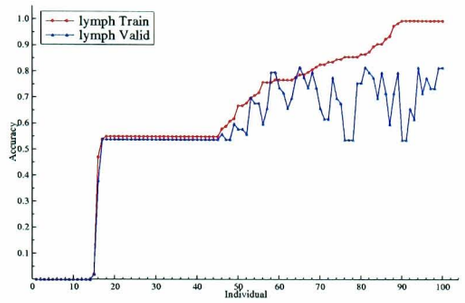

Figure 4.13: Classification accuracies for data set *lymph* in Generation 1

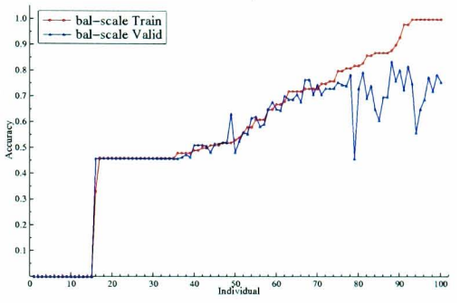

Figure 4.14: Classification accuracies for data set *balance-scale* in Generation 1

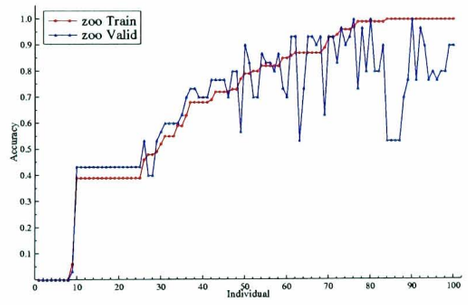

Figure 4.15: Classification accuracies for data set *zoo* in Generation 1

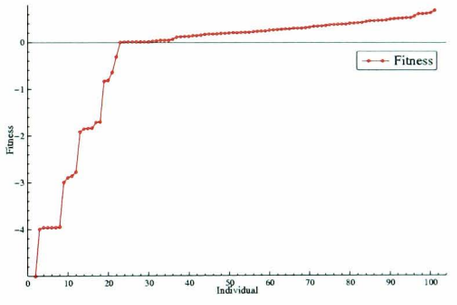

Figure 4.16: Fitness values of the GGP individuals in the first generation

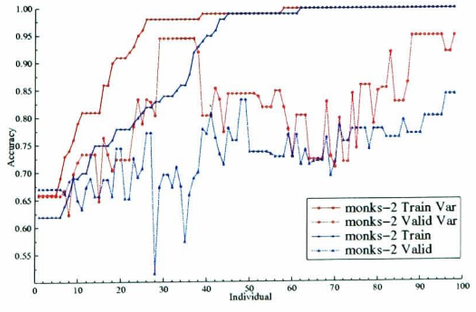

Figure 4.17: Classification accuracies for data set *monks-2* in Generation 10

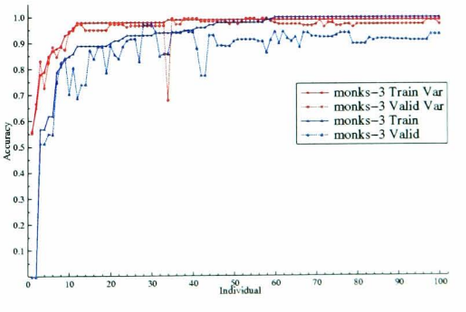

Figure 4.18: Classification accuracies for data set *monks-3* in Generation 10

When analyzing the results of these preliminary experiments run to choose the more appropriated fitness function to guide the GGP, we noticed three things: (1) the values of the fitness obtained by the individuals when evaluating them in the meta-training set were much superior than the ones obtained when running the evolved rule induction algorithms in the meta-test set (a set of data sets different from the ones used in the meta-training set); (2) The population was converging too fast and, on average, after 20 generations, approximately 70% of the individuals in the population were the same; (3) The algorithm was using an elitist strategy, and the elitist individual would be found in the very early generations.

Taking into account these three observations, we realized all of them were related to the same thing: over-fitting. At first this was not very obvious, since the value of the fitness in the meta-training set is expected to be greater than the values of the fitness in the meta-test set (since the GGP "saw" the data in the meta-training set many times during evolution). The population could be converging because the fitness and/or the selection procedure were not working properly, and in consequence the long-term survival of an elitist individual found in an early generation would reflect the badly guided search.

We solved the over-fitting problem with a simple and effective solution borrowed from the literature on GP for data mining [14, 21]: at each generation, the data used in the training and validation sets of the data sets in the meta-training set are merged and randomly redistributed. This means that, at each generation, the GGP individuals are evaluated in a different set of validation data, helping to avoid over-fitting. In order to illustrate the advantages of using this approach, Figures 4.11 to 4.15 show, for all the 100 individuals of the population in generation 1, the values of the predictive accuracy found in the training and validation sets of the 5 data sets used in the meta-training set, namely *monks-2*, *monks-3*, *balance-scale*, *lymph* and *zoo*. The accuracies are ordered from the smallest to the biggest values, so the values of the second points in the graphs for *monks-2* and *monks-3*, for instance, might not be obtained by the same individual. Accuracy values of 1 in the training set followed by much lower accuracies in the validation set usually indicate that the rule induction algorithm is over-fitting the model to the training data.

All the graphs presented in Figure 4.11 to 4.15 have some similar features. First, all of them present individuals which obtain accuracies of 0. The value 0 in accuracy indicates that this particular individual entered into an infinite loop when

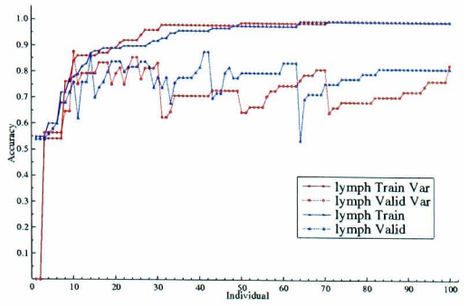

Figure 4.19: Classification accuracies for data set *lymph* in Generation 10

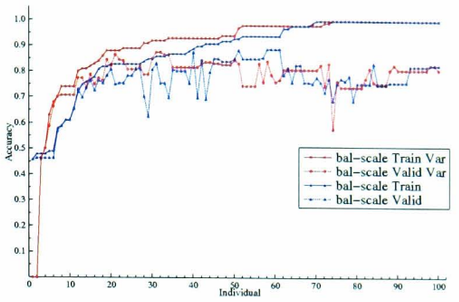

Figure 4.20: Classification accuracies for data set *balance-scale* in Generation 10

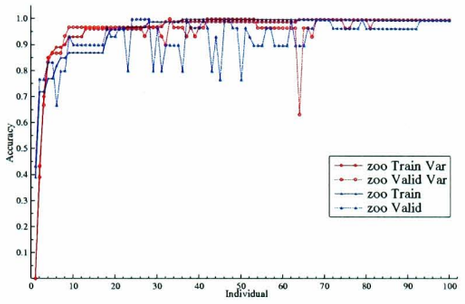

Figure 4.21: Classification accuracies for data set *zoo* in Generation 10

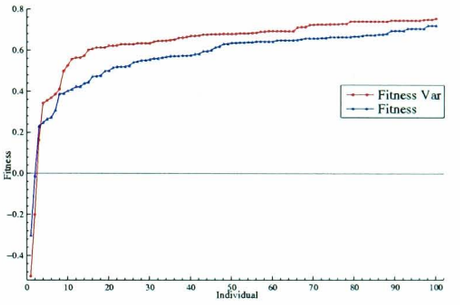

Figure 4.22: Fitness values of the GGP individuals after 10 generations

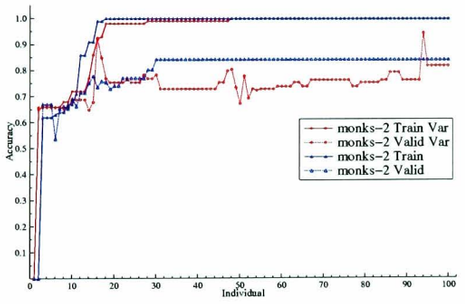

Figure 4.23: Classification accuracies for data set *monks-2* in Generation 20

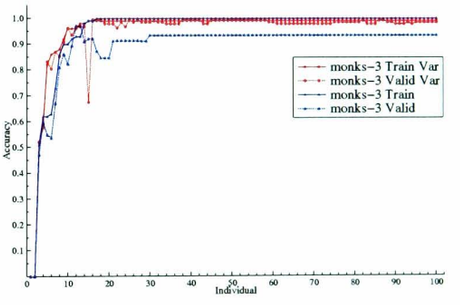

Figure 4.24: Classification accuracies for data set *monks-3* in Generation 20

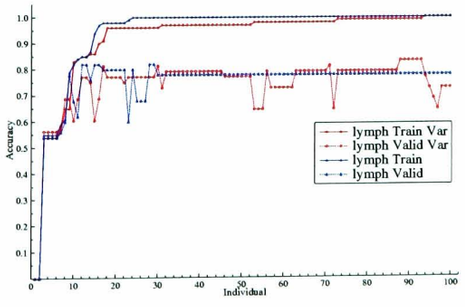

Figure 4.25: Classification accuracies for data set *lymph* in Generation 20

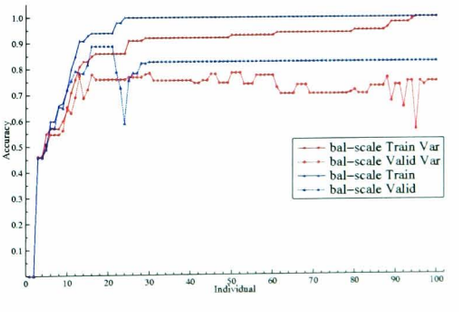

Figure 4.26: Classification accuracies for data set *balance-scale* in Generation 20

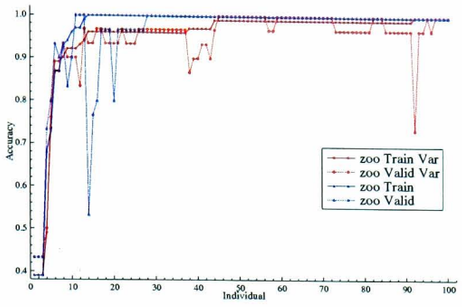

Figure 4.27: Classification accuracies for data set *zoo* in Generation 20

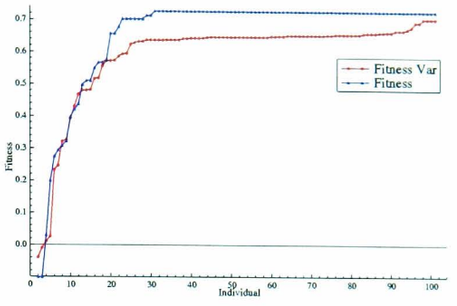

Figure 4.28: Fitness values of the individuals evolved by the GGP after 20 generations

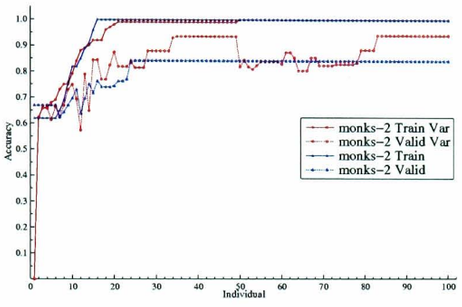

Figure 4.29: Classification accuracies for data set *monks-2* in Generation 30

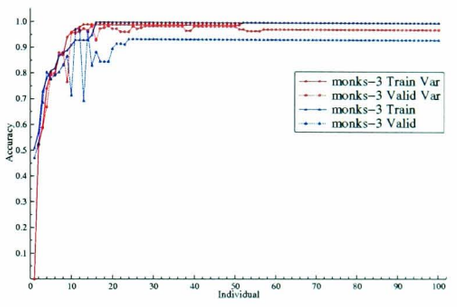

Figure 4.30: Classification accuracies for data set *monks-3* in Generation 30

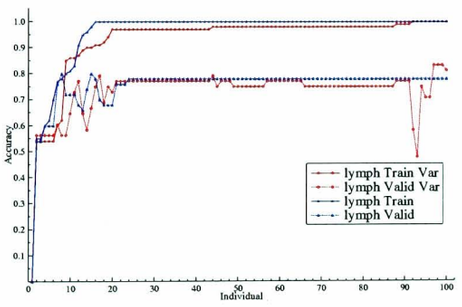

Figure 4.31: Classification accuracies for data set *lymph* in Generation 30

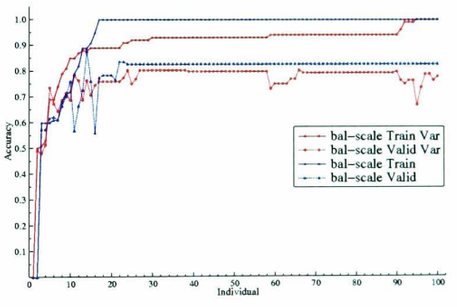

Figure 4.32: Classification accuracies for data set *balance-scale* in Generation 30

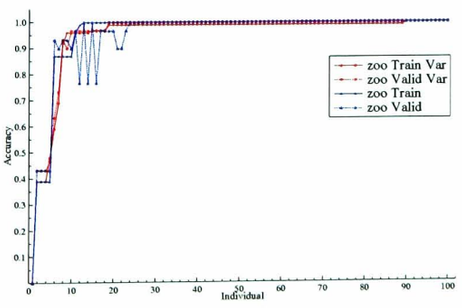

Figure 4.33: Classification accuracies for data set *zoo* in Generation 30

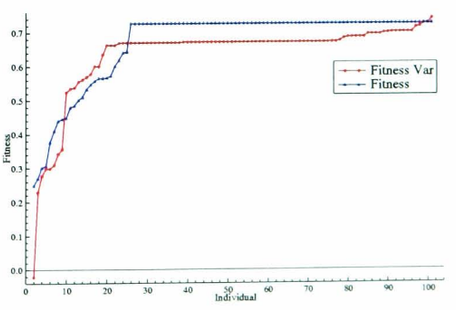

Figure 4.34: Fitness values of the GGP individuals after 30 generations

generating a classification model for that particular data set. Secondly, the graphs for all data sets contain a flat region, which represents a number of individuals which obtain the same predictive accuracy for that data set. These flat regions actually represent individuals obtaining the data set's default accuracy (using the class of the majority of examples in the training set to classify new examples in the validation set). Thirdly, we can observe that not all the individuals with the same classification accuracy in the training set obtain the same predictive accuracy in the validation set. These are usually individuals with small variations, like their criteria to stop pruning rules or the size of the beam in the beam search. At last, all of the graphs show individuals obtaining accuracy values of 1 in the training set.

Figure 4.16 shows the fitness values obtained by all the individuals, which is the average of the values of fit_i in all the data sets. As the graph illustrates, the first 17 individuals have fitness values smaller than -1, that is, they entered into an infinite loop in at least one data set in the meta training set. As it can also be observed, the maximum fitness value obtained by a GGP individual was smaller than 0.3.

The graphs presented in Figure 4.11 to 4.15 illustrate the accuracies of the individuals when evaluated in the first generation. In turn, Figures 4.17 to 4.21 show the classification accuracies in the training and validation sets for all the data sets in the meta-training set in generation 10, using the same training and validation sets to evaluate the individuals during the evolution, and also varying them at each generation. Figures 4.23 to 4.27 and Figures 4.29 to 4.33 show the same information for generations 20 and 30, and Figures 4.22, 4.28 and 4.34 show the evolution of the fitness values for all the individuals in generations 10, 20 and 30, respectively. In all graphs, legends referring to *data-set-name Train/Valid* indicate that the graph was built using fixed training/validation sets, while legends referring to *data-set-name Train Var/Valid Var* indicate that the graph was built using training/validation set variations. Note that the accuracies shown in the *data-set-name Train/Valid* graphs were not obtained in the same training/validation sets than the accuracies in *data-set-name Train Var/Valid Var* graphs, because in the latter the training and validation sets change at each generation. Nonetheless, these comparisons are still interesting, as they reflect the behavior of the GGP during the evolution using these two different strategies.

As we can observe in Figures 4.17 (*monks-2*), 4.18 (*monks-3*) and 4.21 (*zoo*), in these data sets the gap between the values of classification accuracy in the

training and validation sets is smaller in the GGP runs using data sets variation than in the GGP runs using fixed data. Considering particularly the cases where the values of the accuracy in the training set is 1, in the data set *monks-2*, for example, there are cases in which the accuracy obtained by the GGP with variable validation set is 0.7 (probably due to over-fitting) and others where the accuracy in the validation set is 0.95. Nonetheless, for GGP using fixed data, the accuracy in the validation set varied from 0.77 to 0.83. In Figure 4.19 (data set *lymph*) we observe the opposite situation, where the gap in the variations of fitness in the training and validation sets is smaller when using the GGP with no variation in data. In Figure 4.20 (data set *balance-scale*), both runs of the GGP with fixed and varied validation data obtained similar results. So, in generation 10, the benefits of varying the training/validation sets at each generation are not clear yet.

However, in generation 20, the data over-fitting with GGP using fixed data becomes more clear. In the data set *monks-2* (Figure 4.17), for example, 84 out of 100 individuals in the version of the GGP with no data variation obtain accuracy 1 in the training set. The same is true for 52 out of 100 individuals in the GGP run with data variation. However, the classification accuracy of the former 84 individuals is the same in the validation set, while the same is not true for the latter. This means that most of these 84 individuals are the same, and are over-fitted to the data. In the data set *lymph* (Figure 4.19), the scenario changed dramatically since generation 10, and now the GGP with data variation has better classification accuracies on the validation set and more diverse individuals. For *balance-scale*, the values of accuracy of the GGP with fixed data are much higher. *Zoo* is the only data set where most of the individuals which obtain accuracy 1 in the training set of the GGP using fixed data also classified examples in the validation set with 100% accuracy.

In conclusion, in generation 20, roughly 70% of the population was dominated by one individual, which over-fitted all the data sets but *zoo* when using a fixed set of data. This is not true when the GGP uses variable data sets at each generation. In this case, in generation 20, the population is more diverse and the cases of over-fitting account for less than 5% of the individuals in the population.

The analysis of this same experiment in generation 30 confirms the over-fitting of the GGP individuals trained with fixed data. Now more than 80% of the population is composed by one individual which obtains 100% accuracy in most of the training sets. Regarding the fitness of the individuals at generation 30, in general, the fitness of the GGP working with fixed data is better than the fitness

of the variable data GGP, but it has the same values it had 10 generation earlier. In contrast, the fitness of the individuals of the GGP working with variable data is more diverse.

4.5.3 Multi-Objective Fitness

As emphasized before, one of the main motivations to use rule induction algorithms to learn knowledge from data is to take advantage of the simplicity and human-readable format of the induced rules. Nevertheless, the fitness function presented in the previous section did not take into account the simplicity of the model being built by the evolved rule induction algorithm.

In a second phase of this research, we added to the GGP a fitness function based on the concept of Pareto optimization, which simultenously optimizes two objectives:

1. It maximizes the value of the single-objective fitness function, based on classification accuracy (see Eq. (4.4)).

2. It minimizes the number of rule conditions in the produced rule model.

This fitness function based on the Pareto optimization concept has to treat one special case: individuals whose rule induction algorithms produce models consisting of only one empty rule, i.e., individuals predicting the class of the most common examples in the training set for new examples in the validation set. For these individuals, the number of conditions in the produced model will always be 0. As we are dealing with a minimization problem, 0 represents the best possible value, and this individuals would always appear in the Pareto front. To avoid this situation, the number of rule conditions of a model with 0 conditions is set to 100,000.

Apart from the individuals' evaluation process, some other modifications were added to the single-objective version of the GGP to deal with the multi-objective approach, as illustrated in the pseudo-code of the multi-objective GGP (MOGGP), described in Alg. 4.3. First, in the MOGGP selection process, the individuals have to be selected according to a relationship of dominance instead of a simple fitness value. In a tournament selection of size 2, for instance, which is the method used in this system, the winner of the tournament is the individual that dominates the other (recall that an individual Ind_1 dominates an individual Ind_2 if Ind_1 is not worse than Ind_2 in any of the objectives being optimized, and Ind_1 is strictly

Algorithm 4.3: Pseudo-code of the Multi-objective version of the GGP

```
Let the meta-training set be a set of data sets
pop = N individuals ramdomly generated from the grammar
for i = 1 to maximum number of generations do
    non-dominated = ∅
    for every individual Ind in pop do
        Run the rule induction algorithm represented by Ind in the
          meta-training set
        obj1 = avg of the normalized accuracy in the meta-training set
        obj2 = avg of the number of rule conditions in the meta-training set
    non-dominated = Individuals in pop non-dominated according to the
                    Pareto criterion using obj1 and obj2
    newPop = ∅
    if non-dominated > N/2 then
        Calculate value of f3 for individuals in non-dominated
        Sort non-dominated according to f3
        newPop = N/2 individulas with highest value of f3
    else newPop = newPop ∪ non-dominated
    while newPop < N do
        Use tournament selection to select Ind1 and Ind2 from pop
        Apply crossover and mutation operators to Ind1 and Ind2 according
          to user-defined probabilities
        newPop = Ind1 ∪ Ind2
    pop = newPop
non-dominated = Individuals in pop non-dominated according to the
                Pareto criterion using obj1 and obj2
Calculate value of f3 for individuals in non-dominated
Sort non-dominated according to f3
Return the individual with the highest value of f3
```

better than Ind_2 in at least one of the objectives being optimized). In the case that none of the individuals dominate the other, a tie-break criterion decides the winner.

This tie-break criterion considers the difference of the number of individuals in the entire population which are dominated by an individual, and the number of individuals in the population which dominate that individual [114]. This function is named f_3, as it acts like a third objective being optimized by the GGP. If neither Ind_1 dominates Ind_2 nor vice-versa with respect to accuracy and rule set size, the winner of the tournament is the individual with the largest value of f_3.

The second modification introduced in the MOGGP concerns both the elitist strategy and the final solution returned to the user. The single-objective version

of the GGP works with elitism, so it preserves the best individual found by the GGP during the evolution process. This individual is the one returned to the user.

In the case of the MOGGP, at each generation, all the solutions in the Pareto front (individuals not dominated by any other individual in the population) are preserved, as long as their number does not exceed half of the size of the population. If they do, then the individuals with the highest value of f_3 are preserved. If after applying f_3 the number of individuals is still higher than half of the population size, the individuals with better *fit* (fitness function defined in the previous section) are given priority. At the last generation, this same logic is applied when selecting the best individual to be returned to the user and tested in the meta-test set.

4.6 Crossover and Mutation Operations

In a GGP system, the new individuals produced by the crossover and mutation operators have to be consistent with the grammar. For instance, when performing crossover the system cannot select a subtree with root *EvaluateRule* to be exchanged with a subtree with root *SelectCandidateRules*, because this would create an invalid individual according to the grammar.

Therefore, crossover operations have to exchange subtrees whose roots contain the same non-terminal, apart from *Start*. Crossing over two individuals swapping the subtree rooted at *Start* (actually, the entire tree) would generate exactly the same two individuals, and so it would be useless.

Mutation can be applied to a subtree rooted at a non-terminal or applied to a terminal. In the former case, the subtree undergoing mutation is replaced by a new subtree, produced by keeping the same label in the root of the subtree and then generating the rest of the subtree by a new sequence of applications of production rules, so producing a new derivation subtree. When mutating terminals, the terminal undergoing mutation is replaced by another "compatible" symbol, i.e., a terminal or non-terminal which represents a valid application of the production rule whose antecedent is that terminal's parent in the derivation tree. The probability of mutating a non-terminal is 90%, while the probability of mutating a terminal is 10%.

However, not all of the terminals can be mutated. Terminals like *if*, *then*, *else* and *while*, which would not introduce any modifications to the new individual, are

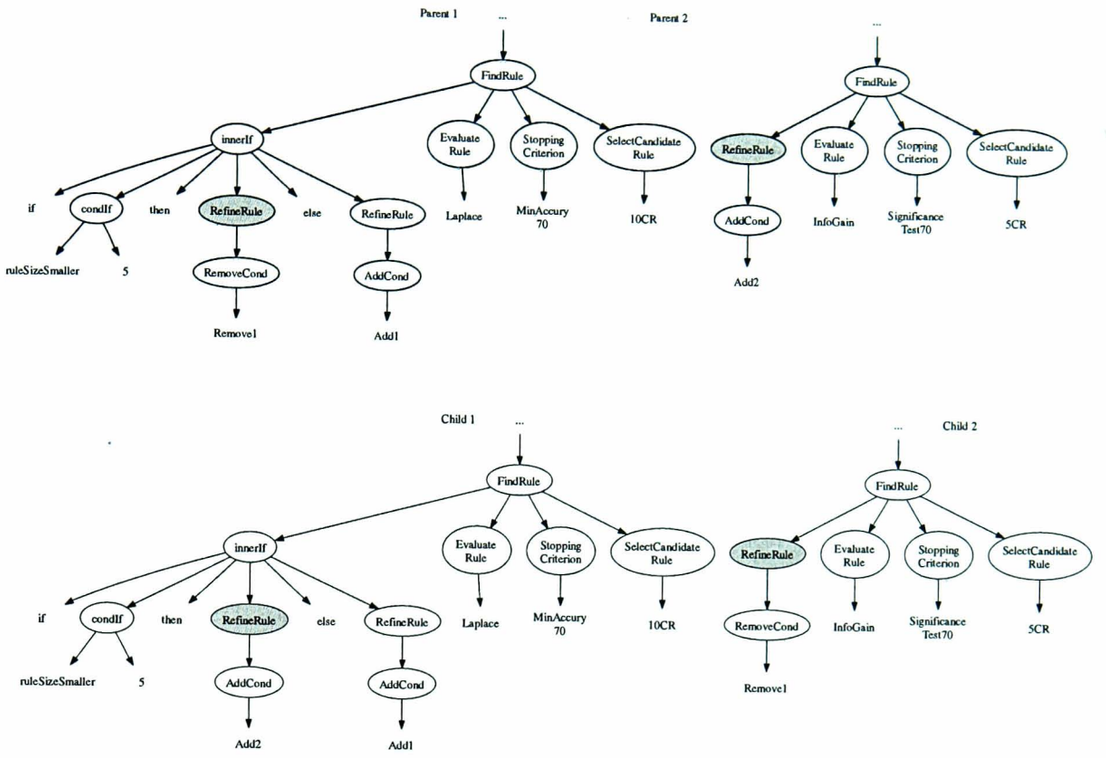

Figure 4.35: Example of Crossover in the proposed GGP

unconsidered during mutation operations. Crossover is not performed in terminal nodes, but it is equivalent to crossing over a non-terminal which generated only a terminal node.

From the 26 non-terminals present in the grammar, eight out of them are more likely to introduce significant changes into the rule induction algorithms represented by the GGP individuals. They are, namely, *CreateOneRule*, *InitializeRule*, *innerWhile*, *innerIf*, *FindRule*, *EvaluateRule*, *RefineRule* and *PrePruneRule*. These non-terminals have a greater chance of being selected to be swapped during crossover operations, or replaced during mutation operations. After some preliminary experiments, the probability of crossover/mutate these non-terminal was set to 70%, being the probability of selecting the remaining ones 30%.

Figure 4.35 shows an example of a crossover operation. Note that just part of the individuals are shown, for the sake of simplicity. The process works as follows. *Parent 1* has a node probabilistically selected for crossover. In the example illustrated, the chosen node is *RefineRule*. The node *RefineRule* is then searched in the derivation tree of *Parent 2*. As *Parent 2* has a node named *RefineRule*, their subtrees are swapped, generating *Child 1* and *Child 2*. If *RefineRule* is not present in the tree of *Parent 2*, a new non-terminal is selected from the tree of

Parent 1. The GGP performs at most 10 attempts to select a node which can be found in both parents. If after 10 attempts it does not happen, both individuals undergo mutation operations.

Recall that there is a set of combinations of terminals and non-terminals of the grammar which are not allowed because they generate invalid individuals (see Section 4.4). These combinations are avoided through a set of constraints considered in the population initialization process. These same constraints are also considered during the crossover and mutation operations.

4.7 Related Work

For many years, Evolutionary Algorithms (EAs) have been used to automate the difficult and time-consuming task of optimizing parameters of learning algorithms. Some examples include the use of EAs for searching the training weights of an artificial neural network [120] or optimizing the kernel parameters of a support vector machine [54]. Since there is a very large numnber of EAs for evolving parameters of a learning system, and there are already comprehensible surveys about this kind of work, we do not further review these EAs here. The interested reader is referred to [52].

After their success on optimizing the parameters of learning systems, EAs started to be applied to optimize the components of these same learning systems. Let us consider, for instance, their use in instance-based learning (IBL) systems. Recall that an IBL algorithm is composed of three main components: (a) a similarity function; (b) a "typical" instance selection function; (c) a classification function. EAs were used to select the instances and the attributes which are stored by the algorithm, i.e., as the "typical" instance selection function [128], and also to evolve instances' prototypes [94]. Another example of EAs applied to optimize components of learning systems is the GP proposed in [75], which evolves the kernel of a Support Vector Machine (SVM) system (note that SVMs are based on two key elements: a general purpose learning algorithm and a problem specific kernel).

In the context of automatically evolving components of learning systems, we are aware of one attempt to evolve the evaluation function of a rule induction algorithm. Wong [147] used an individual-encoding-solution GGP to automatically evolve the evaluation function (or scoring function) of the FOIL algorithm (an inductive logic programming algorithm). The GGP proposed by Wong creates

a population of evaluation functions by following the production rules of a logic grammar, which uses terminals like the current information gain of the rule being evaluated, the number of positive and negative examples covered by the rule being evaluated, and random numbers. Individuals in the first population can also be provided by the user or induced by other learning systems.

The individuals (evaluation functions) generated by the GGP are then incorporated into a generic version of a top-down first-order logic learning algorithm based on FOIL, and the learning algorithm as a whole is evaluated. The fitness of an individual is calculated as the sum of the number of examples misclassified in the validation set of 4 learning tasks, which are randomly created from a single *member* data set at the beginning of each generation.

If EAs are capable of evolving specific learning systems' components, it would be natural that, as some point, EAs would be able to evolve all the necessary components and, consequently, a complete learning system, without any human intervention. The system proposed in this thesis falls in this category, as it uses a GGP to automatically evolve complete rule induction algorithms. Although, to the best of our knowledge, EAs were not used to evolve complete rule induction algorithms before, EAs have been successful when evolving another type of learning system: artificial neural networks [150].

In the research area of ANN, after the first works dedicated to automatically evolve the training weights of ANNs, many others devoted to evolve the ANN's architecture (i.e. number of hidden layers, number of hidden neurons and the interconnections between neurons), node transfer functions and the ANN's learning rules were developed, creating evolutionary artificial neural networks (EANNs) [150, 126].

The main characteristic of EANNs is their capability of adapting to an environment as well as to changes in the environment. They can evolve just the ANN architecture [25] or integrate the evolution of training weights, architecture and learning rules [2], leading to an entirely automated process of ANN design.

In the case of EANNs, researchers already went one step further, and are using EAs to create ensembles of EANNs [23]. In this same context of ensembles of classifiers, [86] also used a GP to combine ANN and the C4.5 decision tree algorithm. However, in this latter work, ANNs and decision trees were not evolved by the GP, but inserted into its function set. The GP was only used as a tool to create an ensemble with the available classifiers. Although the evolution of ensembles of classifiers is out of the scope of this work, this is a possible future

research direction (see Section 7.2).

Apart from EANNs, there is another system which proposed the evolution of learning systems. This system is described here because, at the end of the evolutionary process, it can produce a decision tree or a neural network algorithm. However, it manipulates more the data which the learning system will work with than the system itself. Suyama *et* al. [134] used a hybrid of a GP and a local search method to evolve a classification algorithm. Their system, named CAMLET, differs from a conventional GP in the sense that it does not use traditional GP operators, but rather three strategies which use heuristics based on empirical analysis to generate new individuals. These strategies were named greedy alteration, random generation and heuristic alteration. The first two are adaptations of the crossover and mutation operations, while the last is a kind of local search.

However, CAMLET uses an ontology rather than a grammar to guide its search. The ontology used in [134] has 15 coarse-grained building blocks, where a leaf node of the ontology is a full classification algorithm, like a decision tree, a genetic algorithm or a neural network. By contrast, our grammar is much more fine-grained; its building blocks are programming constructs ("while","if", etc), search strategies and evaluation procedures not used in [134].

The fitness of CAMLET is the accuracy of the individual in the target data set, and the search for solutions stops when the best fitness(accuracy) so far reaches a satisfactory value pre-defined by the user.

In a more recent work, Abe and Yamaguchi [1] compared a parallelized version of CAMLET to three stacking methods, but replaced its hybrid search mechanism by a simple genetic algorithm, using tournament selection, crossover and mutation during the evolution.

The work proposed in this thesis cannot be compared to the ones which evolve neural networks, as they generate complete different learning systems. However, it can be contrasted with the systems proposed by Wong [147] and Suyama *et* al. [134]. When compared to the former, the search space for our algorithm is the space of sequential covering rule induction algorithms, whilst the search space for Wong's GGP is just the space of evaluation functions for FOIL. As a result, our grammar is much more elaborated than the grammar used by Wong.

Furthermore, in both [147] and [134], the GP was trained with a single data set, like in any other use of GP for discovering classification rules. By contrast, this thesis proposed the very novel idea that the GGP is trained with several data sets (from different application domains) in the "meta-training set" in the

same run of the GGP, in order to evolve a truly generic and robust rule induction algorithm, and not just a rule induction algorithm tailored to one particular data set. In any case, in addition to this idea, we also performed experiments where the GGP was used to create a new rule induction algorithm tailored to a specific data set, as discussed in much more detail in Chapter 6.

4.8 Summary

This chapter introduced a grammar-based genetic programming (GGP) system to automatically evolve rule induction algorithms. It described in detail the main components of the system, including the grammar, the individual representation, the population initilization process, the individual evaluation procedure and the crossover and mutation operators.

The grammar includes components already implemented in many well-known rule induction algorithms, and also elements we thought would work well in the context of rule induction algorithms, but that were not tried before.

An individual encodes a rule induction algorithm, and is represented by a derivation tree created by applying a set of production rules from the grammar. An individual is evaluated using a meta-training set (a set of data sets). For each data set in the meta-training set, a predictive accuracy and a rule list/set is obtained. The GGP system can then use a single-objective approach or a multi-objective approach (based on the concept of Pareto optimality) to select the individuals which will be passed to the next generation. The single objective approach aims at maximizing classification accuracy only, while the multi-objective approach aims at both maximizing classification accuracy and minimizing the size of the classification model.

We also showed how the crossover and mutation operations were adapted to generate only individuals which are valid according to the grammar, and compared the proposed system to two other works which used GGP or a similar type of system to automatically evolve some aspects of a rule induction algorithm – but not a full rule induction algorithm as shown in this thesis.

The next chapters present the results of experiments performed to evaluate the efectiveness of the proposed GGP system in automatically evolving rule induction algorithms, its sensitivity to parameters, and how the effectiveness of GP as a search method can be compared to the effectiveness of a more conventional, simpler search method (hill-climbing search).

Chapter 5

Evaluating the Proposed System for Evolving Robust Algorithms

5.1 Introduction

The basic idea of automatically evolving rule induction algorithms, as proposed in this thesis, is a framework which can be used in at least two different approaches:

1. To evolve a *robust* rule induction algorithm from multiple data sets (from different application domains)

2. To evolve a rule induction algorithm tailored to *one specific data set* (from a single application domain)

The GGP can produce rule induction algorithms following any of these two approaches simply by modifying accordingly the data sets used during the GGP training phase. For instance, when creating *robust* rule induction algorithms, a diverse set of data sets is used in the meta-training set required by the algorithm. In theory, the more diverse the data sets are, the more robust the produced rule induction algorithm will be. In contrast, when creating rule induction algorithms for a specific data set, only the target data set is used in the meta-training set.

This chapter reports computational results obtained for the proposed GGP using the first described framework, while results using the second framework will be reported in Chapter 6.

In the context of evolving robust rule induction algorithms, the proposed GGP was evaluated in five phases. In the first phase, as described in Section 5.2, we study the impact of different GGP parameter values (specially for the crossover

and mutation rates) in the results obtained by the method in order to select a good parameter setting for further experiments.

In the second phase, as described in Section 5.3, we evaluate the rule induction algorithms generated by the GGP by comparing them with four well-known manually designed rule induction algorithms: the ordered [27] and unordered [26] versions of CN2, Ripper [33] and C4.5Rules [124].

In the third phase, described in Section 5.4, we analyze the rule induction algorithms automatically evolved by the GGP (GGP-RIs), and compare their structure (in terms of their algorithmic components) with the human-designed rule induction algorithms.

In the fourth phase, we examine the influence of the number of data sets used in the meta-training set of the GGP in the rule induction algorithms produced. Is a GGP trained with 3 data sets able to produce algorithms competitive with the ones produced by the GGP trained with 5 or 7 data sets? This question is the subject of Section 5.5.

In the fifth phase, we moved from changing GGP parameters to changing GGP components. We performed experiments with three variations of the original grammar proposed in Section 4.2, and analyzed how the GGP-RIs produced by these variations compare to the GGP-RIs evolved by the system with the original version of the grammar.

Furthermore, in order to evaluate the effectiveness of the GGP in producing good rule induction algorithms, Section 5.7 reports the results of using a hill-climbing search method to automatically evolve rule induction algorithms, and compares its results with the rule induction algorithms evolved by the GGP.

At last, Section 5.8 presents results when using a variation of the GGP evaluated in the previous sections, where a multi-objective fitness function is considered.

5.2 Investigating the GGP Sensitivity to Parameters

All the experiments performed using the proposed GGP need two sets of parameters to be defined: (1) the parameters for the GGP algorithm and (2) the data sets used during the training phase of the algorithm. In this section we analyze how the GGP parameters – specially the crossover and mutation rates – influence the results obtained by the GGP.

Table 5.1: Data sets used by the GGP

Data set	Examples	Attributes		Classes	Def. Acc.
		Nomin.	Numer.		(%)
monks-2	169/432	6	-	2	67
monks-3	122/432	6	-	2	52
bal-scale-discr	416/209	4	-	3	46
lymph	98/50	18	-	4	54
zoo	71/28	16	-	7	43
monks-1	124/432	6	-	2	50
mushroom	5416/2708	23	-	2	52
promoters	70/36	58	-	2	50
wisconsin	456/227	9	-	2	65
splice	2553/637	63	-	3	52
pima	513/255	-	8	2	65
heart-c	202/101	7	6	2	54.5
ionosphere	234/117	-	34	2	64
hepatitis	104/51	14	6	2	78
sonar	139/69	-	60	2	53
vehicle	566/280	-	18	4	26
vowel	660/330	3	10	11	9
crx	461/229	9	6	2	67.7
glass	145/69	-	9	7	35.2
segment	1540/770	-	19	7	14.3

In order to investigate the influence the GGP parameters have in the quality of the rule induction algorithms produced, we first define the data sets which will be used in the GGP meta-training and meta-test sets. However, it is not clear how many data sets should be used in each of these meta-sets of data, or what would be the best criteria to distribute them into these two meta-sets. Intuitively, the larger the number of data sets in the meta-training set, the more robust the evolved rule induction algorithm should be. On the other hand, the smaller the number of data sets in the meta-test set, the less information we have about the ability of the evolved rule induction algorithm to obtain a high predictive accuracy for data sets unseen during the evolution of the algorithm.

Table 5.1 shows the 20 data sets used in the experiments. The figures in the column *Examples* indicate the number of examples present in the training and validation data sets – numbers before and after the "/", respectively, followed by the number of nominal attributes, numerical attributes and classes. The last column shows the default accuracy (the accuracy obtained when using the most

frequent class in the training set to classify new examples in the validation – or test – set). It is important to note that during the evolution of the rule induction algorithm by the GGP, for each data set in the meta-training set, each candidate rule induction algorithm (i.e., each GGP individual) is trained with 70% of the examples, and then validated in the remaining 30% – with the exception of the data sets which had pre-defined training and validation sets, such as *monks*. In this case the original sets were kept. Recall that the set of examples used to train a rule induction algorithm is called training set, whilst the set of examples used to validate the classification model built by a rule induction algorithm is called the validation set. These 2 sets of data vary from generation to generation to avoid over-searching of the rule induction algorithm being evolved to the data (see Section 4.5.2). In contrast, in the meta-test set, the evolved rule induction algorithms are evaluated using a well-known 5-fold cross validation procedure [145].

Considering the set of data sets listed in Table 5.1, experiments were organized in two phases. In the first phase, we worked only with the top 10 listed data sets, which do not contain any numerical attributes. This is because the first version of the GGP system did not support numerical attributes.

In a second phase, the feature to support numerical attributes was implemented, and experiments were performed using the bottom 10 data sets in Table 5.1. Besides the fact that this experiment included data sets with numerical attributes, they were also very interesting to study the GGP ability to produce rule induction algorithms when working with a completely different set of data in the meta-training and meta-test sets. In a third phase, experiments were executed using all the 20 data sets in Table 5.1.

In the first phase, we divided the 10 nominal data sets in two groups, placing 5 of them in the meta-training set and the other 5 in the meta-test set. We selected the data sets which compose the meta-training set based on the execution time of rule induction algorithms, so that we included in the meta-training set the data sets leading to faster runs of the rule induction algorithms. The data sets selected to be part of the training set were *monks-2*, *monks-3*, *balance-scale-discr*, *lymph* and *zoo*. The meta-test set is composed by *monks-1*, *mushroom*, *wisconsin*, *promoters* and *splice*.

Following this same approach, the experiments reported using numerical data sets (i.e., the bottom 10 data sets listed in Table 5.1) use *pima*, *hepatitis*, *vowel*, *vehicle* and *glass* in the meta-training set. *Crx*, *ionosphere*, *segment*, *sonar* and

Table 5.2: Accuracy rates (%) obtained by the rule induction algorithms evolved by the GGP using nominal data sets in the meta-training set

Data set	Cross. 0.5 Mut. 0.45	Cross. 0.6 Mut. 0.35	Cross. 0.7 Mut. 0.25	Cross. 0.8 Mut. 0.15	Cross. 0.9 Mut. 0.05
monks-2	78.08±1.67	79.38±1.41	78.12±1.26	76.72±1.47	73.34±0.73
monks-3	90.94±0.4	91.14±0.56	91.1±0.45	90.9±0.5	91.74±0.29
balance-scale	85.34±2.04	82.78±1.07	84.12±1.43	81.22±1.83	83.54±1.66
lymph	78.8±1.74	76.4±1.94	81.6±1.6	82±1.55	78.4±1.47
zoo	97.36±0.66	97.36±0.66	97.36±0.66	96.68±1.06	98.02±0.81

Table 5.3: Accuracy rates (%) obtained by the rule induction algorithms evolved by the GGP using nominal data sets in the meta-test set

Data set	Cross. 0.5 Mut. 0.45	Cross. 0.6 Mut. 0.35	Cross. 0.7 Mut. 0.25	Cross. 0.8 Mut. 0.15	Cross. 0.9 Mut. 0.05
monks-1	100±0	100±0	100±0	100±0	100±0
mushroom	99.98±0.01	99.88±0.03	99.98±0.01	99.84±0.04	99.96±0.02
wisconsin	95.17±0.76	94.82±1.08	95.08±0.97	94.17±0.56	95.17±0.52
promoters	77.42±2.54	75.83±3.48	77.1±3	74.54±2.91	77.86±1.84
splice	88.59±0.33	87.72±0.53	88.2±0.44	87.93±0.49	88.43±0.6

heart-c were used in the meta-test set. In the third phase of experimentation, we merged the meta-training and meta-test sets used in the first and second phases to generate a third meta-training and meta-test sets, respectively.

After creating the meta-training and meta-test sets, we turned to the GGP parameters: population size, number of generations, tournament size and crossover, mutation and reproduction rates. In all the experiments reported in this section, the population size is set to 100, the number of generations to 30 and the tournament size to 2. These three figures were chosen when evaluating the GGP evolution in preliminary experiments, but are not optimized. Regarding crossover, mutation and reproduction rates, GPs usually use a high rate of crossover and low rates of mutation and reproduction. However, the balance between these three numbers is an open question, and may be very problem dependent [12].

The experiments showed in this section aimed to find a good trade-off between the crossover and mutation rates. In order to do so, the reproduction rate was set to 0.05, and the balance between the crossover and mutation rates varied.

We report the results of the accuracy of the evolved rule induction algorithms in the data sets of both the meta-training and meta-test sets of the GGP. It

should be stressed that the accuracies in the validation sets of the data sets in the meta-training set cannot be used to evaluate the predictive accuracy of the GGP-derived rule induction algorithms, because each validation set in the meta-training set was seen many times during the GGP evolution. Nonetheless, the accuracy on the validation sets of the meta-training set is useful to evaluate the success of the training of the GGP, and so it is reported here.

It is also important to state that the evolved rule induction algorithms may not perform well in future data sets (meta-test set), specially if their characteristics are completely different from the ones present in the data sets in the meta-training set. However, this emphasizes the importance of keeping different data sets form different application domains in both the meta-training and meta-test sets, in order to measure the generalization capability of the evolved rule induction algorithms.

Tables 5.2 and 5.3 show the average classification accuracy – in the meta-training and meta-test sets, respectively – of the rule induction algorithms evolved by the GGP with different values for crossover and mutation rates. For each cell of Tables 5.2 and 5.3, the reported accuracy is the average over 5 runs of the GGP, using a different random seed to initialize the population at each run. The numbers after the symbol $\pm$ are standard deviations. The header of each column shows the crossover and mutation rates used by the GGP. Recall that in the meta-training set each rule induction algorithm (i.e., each GGP individual) is evaluated using a validation set, whilst in the meta-test set a 5-fold cross-validation procedure is performed to evaluate the rule induction algorithms produced by the GGP.

All the results in this section are compared using a statistical two-tailed Student's t-test with significance levels 0.05 and 0.01 [145]. In the analysis of the results, for the sake of simplicity, we will refer to the GGP using a crossover rate of 0.5 as GGP-0.5, to the GGP using a crossover rate of 0.6 as GGP-0.6 and so on. The correspondent mutation rate is given by the subtracting the crossover rate from 0.95 – recall that the reproduction rate is 0.05.

The results in Table 5.3 refer to the meta-test set. An analysis of these results using a statistical t-test with a significance level of 0.05 reveals significant differences in the predictive accuracies obtained by the evolved rule induction algorithms using different parameter configurations in only one data set: *mushroom*. In the *mushroom* data set, GGP-0.5 and GGP-0.7 obtained results statistically better than the ones obtained by GGP-0.6 and GGP-0.8. For the other 4 data sets, no parameter configuration showed better results than the others. However,

Table 5.4: Accuracy rates (%) obtained by the rule induction algorithms evolved by the GGP using numerical data sets in the meta-training set

Data set	Cross. 0.5 Mut. 0.45	Cross. 0.6 Mut. 0.35	Cross. 0.7 Mut. 0.25	Cross. 0.8 Mut. 0.15	Cross. 0.9 Mut. 0.05
pima	73.82±1.77	71.12±1.32	72.4±1.92	72.78±1.98	70.84±2.54
hepatitis	72.7±0.48	72.68±1.47	71.62±0.87	72.92±1.58	70.8±1.52
vowel	61.32±4.19	67.52±2.46	64.68±2.52	62.62±2.63	66.74±1.96
vehicle	65.26±0.63	67.28±1.03	66.36±1.85	65.44±0.46	68.18±0.59
glass	62.32±4.78	64.94±2.58	68.72±1.27	68.42±1.92	65.8±1.34

Table 5.5: Accuracy rates (%) obtained by the rule induction algorithms evolved by the GGP using numerical data sets in the meta-test set

Data set	Cross. 0.5 Mut. 0.45	Cross. 0.6 Mut. 0.35	Cross. 0.7 Mut. 0.25	Cross. 0.8 Mut. 0.15	Cross. 0.9 Mut. 0.05
crx	82.55±0.94	81.74±1.05	82.14±0.43	82.35±0.96	81.68±0.85
ionosphere	85.89±2.26	85.03±2.51	86.98±2.54	85.84±2.35	85.6±2.02
segment	91.39±0.63	94.95±0.19	94.75±0.41	91.68±0.89	95.18±0.39
sonar	70.05±1.67	72.89±1.46	72.72±1.95	71.06±1.73	71.74±1.21
heart-c	78.44±2.14	76.43±1.34	75.13±1.52	77.5±0.93	75.14±1.44

if the significance level of the t-test is strengthened to 0.01, all the pairwise comparisons among the parameter configurations in Table 5.3 produce statistically insignificant differences.

Based on these results, we cannot say that any of the GGP parameter configurations evaluated is better than the others. For some data sets one of them can be slightly better than the others, but overall the results obtained were very similar. From these observations we can conclude that the proposed GGP is very robust to the setting of crossover and mutation parameters when trained with the previously mentioned nominal data sets in the meta-training set.

After this first conclusion was drawn, the GGP system was extended with features to manipulate numerical attributes, and then we started a second experimental phase. During this phase, we used data sets with at least one numerical attribute (the bottom 10 data sets listed in Table 5.1) to train and test the GGP. This was a good way to study the behavior of the system when trained with completely different data sets, and check if the results obtained with different parameter configurations would also be consistent with the ones obtained during the first experimental phase.

Table 5.6: Accuracy rates (%) obtained by the rule induction algorithms evolved by the GGP using both nominal and numerical data sets in the meta-training set

Data set	Cross. 0.5 Mut. 0.45	Cross. 0.6 Mut. 0.35	Cross. 0.7 Mut. 0.25	Cross. 0.8 Mut. 0.15	Cross. 0.9 Mut. 0.05
monks-2	73.76±0.85	72.02±0.33	75.22±1.77	71.98±1.27	74.8±1.84
monks-3	90.92±0.37	91.14±0.23	91.26±0.22	92.04±0.6	91.24±0.37
balance-scale	84.76±1.55	84.88±1.14	84.5±1.52	82.56±1.38	84.78±2.28
lymph	80±1.1	78.4±1.17	80±1.67	79.6±0.98	77.6±1.94
zoo	96.7±0	96.7±0	96.7±0	96.7±0	92.7±4
pima	69.64±1.17	70.64±1.3	70.18±0.93	69.56±0.1	72.06±1.42
hepatitis	70.58±0.66	71.2±0.22	70.28±0.93	71.52±0.29	72±1.67
vowel	68.62±1.1	70.3±0.47	71.26±0.72	71.98±0.86	68.6±3.11
vehicle	66.28±1.07	67.14±0.27	68.08±1.5	68.28±1.12	68.22±2.1
glass	71.02±2.2	70.14±2.73	66.66±2.25	67.82±3.67	66.38±2.66

Tables 5.4 and 5.5 report the classification accuracies obtained by the evolved rule induction algorithms in the meta-training and meta-test sets, respectively, when using data sets with at least one numerical attribute. As in the previous experiments, according to a statistical t-test with a significance level 0.05, there was only one data set in which some of the GGP parameter configurations were better than the others: *segment*. For this data set, in Table 5.5 both GGP-0.5 and GGP-0.8 obtained classification accuracies statistically worse than GGPs-0.6, 0.7 and 0.9. The differences in predictive accuracy among GGP-0.5, GGP-0.6 and GGP-0.9 are also statistically significant with a 0.01 significance level. Once more these results do not allow us to consider one of the parameter configurations better than the others in general, and so these results reinforce the previous conclusion about the robustness of the GGP to changes in the crossover and mutation rates.

The third and last experimentation phase executed in order to set the GGP crossover and mutation rates was based on a greater number of data sets in the meta-training and meta-test sets. In total, 10 data sets were used to train and 10 data sets were used to test the GGP. In both cases, the data sets chosen to compose each of the meta data sets were the same used in phases 1 and 2 to train and test the GGP. Tables 5.6 and 5.7 show the results obtained.

Applying the t-test with significance level 0.05 to the results in Table 5.7, we find significant differences in predictive accuracies in 3 data sets: *mushroom*, *segment* and *splice*. Interestingly, *mushroom* and *segment* are the two data sets in which, in previous experiments, there was also a significant difference among

Table 5.7: Accuracy rates (%) obtained by the rule induction algorithms evolved by the GGP using both nominal and numerical data sets in the meta-test set

Data set	Cross. 0.5 Mut. 0.45	Cross. 0.6 Mut. 0.35	Cross. 0.7 Mut. 0.25	Cross. 0.8 Mut. 0.15	Cross. 0.9 Mut. 0.05
crx	77.75±3.81	80.95±0.82	77.46±3.8	79.54±1.62	81.85±0.97
segment	93.98±0.57	95.04±0.48	95.06±0.26	95.34±0.31	93.95±0.37
sonar	72.83±1.91	72.52±1.87	72.34±1.91	71.48±2.3	69.24±1.14
ionosphere	87.2±2.13	86.59±1.97	87.04±2.2	85.88±1.64	87.61±2.24
heart-c	74.67±1.7	75.47±1.77	76.72±1.5	75.44±0.66	77.37±1.19
monks-1	99.82±0.18	99.82±0.18	99.93±0.07	99.64±0.36	100±0
mushroom	99.23±0.05	99.96±0.01	99.99±0	100±0	99.75±0.05
wisconsin	95.11±0.47	95.97±0.38	95.58±0.74	95.02±0.5	95.23±0.7
promoters	80.5±2.29	83.59±2.7	78.98±2.93	78.4±1.9	75.97±1.82
splice	89.16±0.47	89.82±0.42	88.68±0.31	89.82±0.44	82.19±0.47

the results of different parameter configurations. In *mushroom*, the accuracies of all the other 4 parameter configurations are significantly better than the accuracy of GGP-0.5, and the accuracies of GGP-0.6, GGP-0.7 and GGP-0.8 are also significantly better than the one of GGP-0.9. In *segment*, GGP-0.8's accuracy is significantly better than GGP-0.5's accuracy, while in *splice* the accuracies in all parameters configurations are significantly better than the one obtained by GGP-0.9. In these cases, when the significance level is strengthened to 0.01, the statistical significance of results in the *splice* data set and the comparisons among GGP-0.5 and the other configurations in *mushroom* do not change.

From the three experiments previously run using 5 different parameter configurations each, we can conclude that the proposed GGP is robust to the variations in the crossover and mutation rates. However, can the automatically generated rule induction algorithms obtain better classification accuracies than well-known manually designed rule induction algorithms? That is the subject of Section 5.3.

5.3 Comparing GGP-derived Rule Induction Algorithms with Conventional Rule Induction Algorithms

In Section 5.2 we compared the results obtained by the rule induction algorithms evolved by the GGP with different parameter configurations. These

comparisons were useful to show the robust nature of the method concerning variations in the crossover and mutation parameters, but it does not give any insights about how competitive the automatically designed algorithms are when compared to some well-known human-designed rule induction algorithms. In this section, we compare the results obtained by the GGP-derived rule induction algorithms (GGP-RIs) with 4 well-known rule induction algorithms: the ordered [27] and unordered [26] versions of CN2, Ripper [33] and C4.5Rules [124].

From these four algorithms, C4.5Rules is the only one which does not follow the sequential covering approach, which is the approach followed by the GGP-RIs. However, as C4.5Rules has been used as a benchmark algorithm for classification problems for many years, we also included it in our set of baseline comparison algorithms.

It is also important to observe that the current version of the grammar does not include all the components present in Ripper, but does include all the components present in both versions of CN2. In other words, the space of candidate rule induction algorithms searched by the GGP includes CN2, but it does not include C4.5Rules nor the complete version of Ripper.

As in the previous section, we first report the comparisons between the GGP-RIs and baseline algorithms for the experiments using only nominal data sets, followed by the comparisons for experiments using data sets with at least one numerical attribute. Finally, we show the results of comparisons for experiments where both nominal and numerical data sets were used together.

Table 5.8 shows a summary of the comparisons among the GGP-RIs with specific GGP parameter configurations and the baseline algorithms. In Table 5.8, the first column shows the crossover/mutation rates used by the GGPs, while the second and third columns show the number of times the GGP-RIs obtain predictive accuracies significantly better(worse) than the baseline algorithms using both a significance level of 0.05 and 0.01, respectively. Theses figures represent the number of significant "wins/losses" out of 20 comparisons (5 data sets x 4 manually-designed classification algorithms).

As expected, the numbers of significantly better(worse) results do not vary a lot from one parameter configuration to another (as GGPs with different parameter configurations produce similar results). The number of cases where GGP-RIs obtain significantly better results than the baseline algorithms is always 2 with a significance level 0.05, and apart from GGP-0.5, that number is reduced to 1 with a 0.01 significance level. Not surprisingly, all these comparisons refer to the

Table 5.8: Comparing the GGP-RIs trained with different parameters (using nominal data sets) to the baseline algorithms using a statistical t-test

GGP-RIs	Signif. level	
Cross/Mut	0.05	0.01
0.5/0.45	2(2)	2(1)
0.6/0.35	2(4)	1(1)
0.7/0.25	2(2)	1(1)
0.8/0.15	2(3)	1(1)
0.9/0.05	2(2)	1(1)

Table 5.9: Comparing predictive accuracy rates (%) for the nominal data sets in the meta-test set - results obtained with crossover rate = 0.5 and mutation rate = 0.45

Data Set	GGP-RIs	OrdCN2	UnordCN2	Ripper	C45Rules
monks-1	100±0	100 ± 0	100 ± 0	93.84 ± 2.93	100 ± 0
mushroom	99.98±0.01	100 ± 0	100 ± 0	99.96 ± 0.04	98.8 ± 0.06
wisconsin	95.17±0.76	94.58 ± 0.68	94.16 ± 0.93	93.99 ± 0.63	95.9 ± 0.56
promoters	77.42±2.54	81.9 ± 4.65	74.72 ± 4.86	78.18 ± 3.62	83.74 ± 3.46
splice	88.59±0.33	90.32 ± 0.74	74.82 ± 2.94	93.88 ± 0.41	89.66 ± 0.78

same algorithms in the same data sets, as will be discussed later. Considering the number of times the baseline algorithms obtain better results than the GGP-RIs, this number varies from 2 to 4 with a 0.05 significance level, but is always 1 with a 0.01 significance level.

Instead of showing a more detailed comparison of the GGP-RIs with the baseline algorithms for all the parameter configurations, here we choose to present the detailed results of GGP-0.5 only, since the results produced by the other parameter configurations are similar. This parameter configuration, which uses a crossover rate of 0.5 and a mutation rate of 0.45, will also be used in further experiments involving nominal data sets, described in Section 5.5. This choice is based on a slightly better performance of GGP-0.5 when compared to the other classification algorithms using the 0.01 significance level, as shown in the last column of Table 5.8

Table 5.9 shows the average predictive accuracy obtained by the rule induction algorithms produced by the GGP-0.5 using nominal data sets in 5 different runs of the GGP varying the random seed, followed by the results of runs of Ordered-CN2, Unordered-CN2, Ripper and C4.5Rules (all of them using default

Table 5.10: Comparing the GGP-RIs trained with different parameters (using numerical data sets) to the baseline algorithms using a statistical t-test

GGP-RIs	Signif. level	
Cross/Mut	0.05	0.01
0.5/0.45	1(2)	1(2)
0.6/0.35	1(0)	1(0)
0.7/0.25	1(0)	1(0)
0.8/0.15	1(2)	1(0)
0.9/0.05	1(0)	1(0)

parameter values). All the results were obtained using a 5-fold cross-validation procedure. The numbers after the symbol "±" are standard deviations. Results were again compared using a statistical t-test with significance level 0.01. Cells in dark gray represent significant wins of the GGP-RIs against the corresponding baseline algorithm, while light gray cells represent GGP-RIs losses.

In total, Table 5.9 contains 20 comparative results between GGP-RIs and baseline algorithms – 5 data sets × 4 baseline classification algorithms. Out of theses 20 cases, the accuracy of GGP-RIs was statistically better than the accuracy of the baseline algorithms in 2 cases, whilst the opposite was true in one case. In the other 17 cases there was no significant difference.

The GGP-RIs' predictive accuracies are statistically better than the C4.5Rules' accuracy in *mushroom* and Unordered-CN2's accuracy in *splice*. Naturally, these 2 cases involve algorithms with the worst accuracy for the respective data set. It is also in a comparison among Ripper and the GGP-RIs in *splice* where the GGP-RIs obtain a significantly worse accuracy than Ripper. However, we can claim that the accuracies obtained by Ripper in *splice* are much superior to the ones found by all the other baseline algorithms, and as the GGP-RIs did not have all the algorithmic components present in Ripper in its search space, it was not possible to produce an GGP-RI with such a high accuracy in *splice*.

The same type of comparison and analysis done with the experiments involving only nominal data sets in the meta-training and meta-test sets was performed for the set of experiments using data sets with at least one numerical attribute (5 data sets in the meta-training and 5 data sets in the meta-test sets) and the experiments using all the 20 data sets previously utilized. We first show the results involving only data sets having at least one numerical attribute – hereafter

Table 5.11: Comparing predictive accuracy rates (%) for the numerical data sets in the meta-test set - results obtained with crossover rate = 0.7 and mutation rate = 0.25

Data Set	GGP-RIs	OrdCN2	UnordCN2	Ripper	C45Rules
crx	82.14±0.43	80.16 ± 1.27	80.6 ± 0.93	84.37 ± 1.21	84.82 ± 1.53
ionosphere	86.98±2.54	87.6 ± 2.76	90.52 ± 2.03	89.61 ± 1.75	89.06 ± 2.71
segment	94.75±0.41	95.38 ± 0.28	85.26 ± 0.87	95.44 ± 0.32	88.16 ± 7.72
sonar	72.72±1.95	70.42 ± 2.66	72.42 ± 1.4	72.88 ± 4.83	72.4 ± 2.68
heart-c	75.13±1.52	77.9 ± 1.96	77.54 ± 2.85	77.53 ± 1.1	74.2 ± 5.43

denoted numerical data sets for short.

Table 5.10 shows the results of the comparisons among the GGP-RIs with parameter variations and the baseline algorithms. It uses the same notation used in Table 5.8, showing the number of results significantly better(worse) than the baseline algorithms using both a significance level of 0.05 and 0.01. Observe that, with both significance levels 0.01 and 0.05, the GGP-RIs always obtain significantly better accuracies than 1 baseline algorithm in every data set. Significantly worse accuracies than the baseline algorithms are obtained in just 2 cases by GGP-0.5 and GGP-0.8. Once more the fact that no GGP configuration is considerably better than the others is clear in Table 5.10. Taking into account the losses of GGP-0.5 and GGP-0.8, the results obtained by the GGP configurations using crossover rates 0.6, 0.7 and 0.9 were pretty much the same. We chose to report the results obtained by GGP-0.7, once its crossover rate is in between the values 0.5 and 0.9. We now report in detail the comparisons among the GGP-0.7 and the baseline algorithms.

Table 5.11 shows the average predictive accuracy obtained by the rule induction algorithms produced by the GGP-0.7 using numerical data sets in 5 different runs of the GGP varying the random seed (recall that each discovered GGP-RI was, in turn, run for each data set using a 5-fold cross-validation procedure), followed by the cross-validation results of runs of Ordered-CN2 and Unordered-CN2, Ripper and C4.5Rules (using default parameter values in all these algorithms). As we can observe, all the comparisons but one revealed accuracies that were statistically insignificant at the 0.01 significance level from the ones obtained by the baseline algorithms. The exception is the Unordered-CN2 algorithm in the data set *segment*. This algorithm presents a very low predictive accuracy when compared to the other algorithms, and its accuracy is indeed much lower than the one obtained by the GGP-0.7.

Table 5.12: Comparing the GGP-RIs trained with different parameters (using nominal and numerical data sets) to the baseline algorithms using a statistical t-test

GGP-RIs	Signif. level	
Cross/Mut	0.05	0.01
0.5/0.45	3(4)	3(4)
0.6/0.35	3(3)	3(3)
0.7/0.25	3(1)	3(1)
0.8/0.15	3(1)	3(1)
0.9/0.05	2(8)	2(5)

At last, we report the results of comparisons involving the GGPs trained with both nominal and numerical data sets. Table 5.12 reports the number of times the GGP-RIs produced with a specific crossover/mutation rate are significantly better(worse) than the baseline algorithms using the two-tailed t-test with significance levels 0.05 and 0.01. In this case, the better(worse) results are obtained out of 40 comparisons - 10 data sets x 4 baseline classification algorithms. In this particular set of experiments, we can observe that all the GGP-RIs but one obtained classification accuracies statistically better than the baseline algorithms in 3 cases when analyzing the results using both levels of significance. The exceptions are the rule induction algorithms produced by GGP-0.9, which obtained better results in 2 cases instead of 3. However, in these experiments, there was a large variation among the number of results obtained by the GGP-RIs that were significantly worse than the baseline algorithms. They varied from 1 to 8 using a 0.05 significance level and from 1 to 5 when using 0.01. According to their number of losses, GGPs using 0.7 and 0.8 crossover rates produced GGP-RIs slightly better than the others, with 3 results statistically better and 1 result statistically worse than the ones obtained by the baseline algorithms. To be consistent with the choice of parameters made for the experiments using numerical data sets, Table 5.13 presents and compares the results obtained by the rule induction algorithms evolve by GGP-0.7 with the baseline algorithms.

Table 5.13 shows the average accuracy obtained by the rule induction algorithms produced by the GGP-0.7 using both nominal and numerical data sets in 5 different runs, followed by the results of runs of Ordered-CN2, Unordered-CN2, Ripper and C4.5Rules (using default parameter values in all these algorithms). Comparing the results in Table 5.13 with the results in Tables 5.9 and 5.11, we discover the former looks like a summary of these previous two tables reporting

Table 5.13: Comparing predictive accuracy rates (%) for the nominal/numerical data sets in the meta-test set - results obtained with crossover rate = 0.7 and mutation rate = 0.25

Data Set	GGP-RIs	OrdCN2	UnordCN2	Ripper	C45Rules
crx	77.46±3.8	80.16 ± 1.27	80.6 ± 0.93	84.37 ± 1.21	84.82 ± 1.53
segment	95.06±0.26	95.38 ± 0.28	85.26 ± 0.87	95.44 ± 0.32	88.16 ± 7.72
sonar	72.34±1.91	70.42 ± 2.66	72.42 ± 1.4	72.88 ± 4.83	72.4 ± 2.68
heart-c	76.72±1.5	77.9 ± 1.96	77.54 ± 2.85	77.53 ± 1.1	74.2 ± 5.43
ionosphere	87.04±2.2	87.6 ± 2.76	90.52 ± 2.03	89.61 ± 1.75	89.06 ± 2.71
monks-1	99.93±0.07	100 ± 0	100 ± 0	93.84 ± 2.93	100 ± 0
mushroom	99.98±0.02	100 ± 0	100 ± 0	99.96 ± 0.04	98.8 ± 0.06
wisconsin	95.58±0.74	94.58 ± 0.68	94.16 ± 0.93	93.99 ± 0.63	95.9 ± 0.56
promoters	78.98±2.93	81.9 ± 4.65	74.72 ± 4.86	78.18 ± 3.62	83.74 ± 3.46
splice	88.68±0.31	90.32 ± 0.74	74.82 ± 2.94	93.88 ± 0.41	89.66 ± 0.78

results of experiments using only nominal or numerical data sets: the GGP-RIs also obtain significantly better accuracies (at the 0.01 significance level) than Unordered-CN2 in *segment* and *splice* and C4.5Rules in *mushroom*. The single result significantly worse than the results obtained by other classifiers is, again, obtained by the GGP-RIs in *splice*, which is significantly worse than the Ripper result for that data set.

These three sets of experiments performed with different data sets in the meta-training and meta-test sets lead us to conclude that the GGP-RIs can easily outperform classifiers which are not competitive with the other baseline algorithms. For example, in *splice* the predictive accuracy of Unordered-CN2 is 74.82 ± 2.94, while the other algorithms obtain accuracies close to 90%. In this case, the GGP-RIs can easily find a better accuracy than the one found by Unordered-CN2.

On the other hand, we can say that the GGP was not able to find a rule induction algorithm good enough to outperform the predictive accuracies of Ripper in *splice* because it did not have all the components necessary to do that in its grammar. However, note that the accuracy obtained by Ripper in *splice* is also statistically better than the ones obtained by C4.5Rules and CN2- Ordered when applying a t-test with 0.01 significance level.

Finally, recall that the search space of the GGP includes both Unordered and Ordered CN2. Hence, it seems fair to expect that the GGP-RIs would never obtain a predictive accuracy significantly worse than either version of CN2. Indeed, this was the case in the experiments reported in this section. Considering the results

of Tables 5.9, 5.11 and 5.13 as a whole, the GGP-RIs significantly outperformed Unordered-CN2 in 4 cases (dark gray cells in those tables), and there was no case where either version of CN2 significantly outperformed the GGP-RIs.

So far we have shown that the evolved GGP-RIs are competitive to traditional human-designed rule induction algorithms. But how similar the former are to the latter? The next section presents some rule induction algorithms generated by the GGPs and compares them to well-known human-designed ones.

5.4 To What Extent are GGP-RIs Different from Manually-Designed Rule Induction Algorithms?

The main goal of this work was to automatically evolve rule induction algorithms robust enough to obtain a competitive predictive accuracy in new data sets when compared to manually-designed rule induction algorithms. Section 5.3 showed that it is possible to automatically generate competitive rule induction algorithms. However, how similar are they to well-known manually-designed algorithms? Are the GGP-RIs innovations in any aspect of manually-designed algorithms? This section describes the way the CN2 and Ripper algorithms induce rules, and then compares these algorithms with the discovered GGP-RIs. C4.5Rules will not be part of this comparison, as it is does not follow the sequential covering approach. As previously explained in Section 2.4, C4.5Rules extracts a rule set from a decision tree built from the data.

Alg. 5.1 shows the pseudo-code for the Ordered-CN2 algorithm [26]. Note that this algorithm, as all the others described in this section, are instantiations of Alg. 2.2 and Alg. 2.3 (in the case of a rule list) or Alg. 2.1 (in the case of a rule set), described in Section 2.2. For the sake of simplicity, statements presented in these algorithms that do not vary from one rule induction algorithm to another, such as *Remove from T all the examples covered by R'* or *If(CR is better than bestRule) then bestRule = CR*, are omitted.

Alg. 5.1 starts to produce rules with an empty condition, adds one condition-at-a-time to it, evaluates the rule with the new condition using the Laplace estimation, and selects the best 5 produced rules to go on into the refinement process. Notice that a rule is considered a candidate to be the best rule only if it is significant according to a statistical significance test. However, the default setting of CN2 completely ignores this test. The size of the beam shown in the pseudo-code

is 5 because this value is used as the CN2 default one. The algorithm keeps inserting new rules to the rule list while there are uncovered examples in the training set.

Algorithm 5.1: Main part of the pseudo-code of the Ordered-CN2 algorithm

```
RuleList = ∅
repeat
    bestRule = an empty rule
    candidateRules = ∅
    candidateRules = candidateRules ∪ bestRule
    while candidateRules ≠ ∅ do
        for each candidateRule CR do
            newCandidateRules = ∅
            Add 1 condition-at-a-time to CR
            Evaluate CR using the Laplace estimation
            if CR is significant then
                newCandidateRules = newCandidateRules ∪ CR
        candidateRules = 5 best rules selected from newCandidateRules
    RuleList = RuleList ∪ bestRule
until all examples in the training set are covered
```

The unordered version of CN2 presents just one major modification when compared to Alg. 5.1. An outer loop *for* is inserted to repeat the algorithm described in Alg. 5.1 as many times as the number of classes presented in the data. It is important to point out that, when applying the model generated by Unordered-CN2 to unseen examples in the test set, in cases where more than one rule predicting different classes cover the same example, the class of the example is decided using the following probabilistic method. For all the rules which cover the example, the number of covered examples in each class is counted. Then, for each class in turn, the number of examples covered by each of the fired rules is summed up. The class which presents the largest number of examples covered by those rules is chosen as the new example's class.

As showed in Alg. 5.1, CN2 is a fairly simple algorithm. Alg. 5.2 shows the main part of the pseudo-code of Ripper [33]. As observed, this is a much more sophisticated algorithm. It actually builds a rule list of ordered classes, that is, for each class in turn it builds a rule list which separate the current class from the other classes in the data set. It works in three phases: it first grows a rule which covers no negative examples by adding one condition-at-a-time to it, and evaluates the possible candidate rules using the information gain criterion. Once

a rule which covers no negative examples is found, it is pre-pruned by removing from it a set of final conditions. During the pruning phase, new rules are evaluated using the formula $(p - n)/(p + n)$, where p is the number of positive examples covered by the pruned rule and n the number of negative examples covered by the pruned rule. Rules are inserted into the rule set until the minimum description length (MDL) of the rule set (described in Section 2.2.3) is larger than a parameter d (representing a number of bits) plus the size of the smallest MDL found so far, or they are no positive examples uncovered. Once the rule set for one class is complete, it goes through an optimization phase. During this phase, each rule is analyzed in turn, and can be replaced by another rule or revised [33]. This decision depends again on the MDL heuristic, which can also lead to a rule being removed from the model.

Algorithm 5.2: Main part of the pseudo-code of the Ripper algorithm

```
Let n be the number of classes
Sort classes in ascendent order, C1,...,Cn, according to their number of
examples
RuleSet = ∅
for i = 1 to n-1 do
    Positive = examples from Ci
    Negative = examples from remaining classes
    RuleSet' = ∅
    repeat
        Divide the training data in Grow and Prune
        R = an empty rule
        while R covers negative examples do
            newCandidateRules = ∅
            newCandidateRules = Add 1 condition-at-a-time to R
            Evaluate newCandidateRules using information gain in Grow
            R = best rule in newCandidateRules
        R' = Rule produced when removing last condition from R
        while (p-n)/(p+n)(R') > (p-n)/(p+n)(R) in Prune do
            R = R'
            R'= Rule produced when removing last condition from R
        RuleSet' = RuleSet' ∪ R
        Remove examples covered by R from training set
    until Positive ≠ ∅ OR MDL of RuleSet' is d bits > the smallest MDL
          found so far
    RuleSet = RuleSet ∪ Optimized RuleSet'
    Remove all examples covered by RuleSet' from the training set
Make class Cn the default class
```

It is important to emphasize that the current version of the grammar does not use the MDL criterion to stop producing rules, and also does not implement all of the steps required by Ripper's optimization process. Implementing the MDL heuristic and the complete Ripper optimization process is not straightforward, and integrating it to the current version of the system is even more time consuming, so these extensions were left for future research.

Now we describe three different GGP-RIs, each of them produced by an experiment using different data sets in the meta-training set. The GGP-RIs were chosen according to the largest value of the difference in the number of times they obtained significantly better and significantly worse accuracies than CN2 and Ripper. In many cases, there were more than one algorithm with the same number of wins/losses against the CN2 and Ripper baseline algorithms. In this case, we randomly chose one of them to be shown here.

Algorithm 5.3: Example of a Decision List Algorithm created by the GGP using Nominal Data Sets

```
RuleList = ∅
repeat
    bestRule = an empty rule
    candidateRules = ∅
    candidateRules = candidateRules ∪ bestRule
    while candidateRules ≠ ∅ do
        for each candidateRule CR do
            newCandidateRules = ∅
            if RuleList covers more than 90% of the training examples then
                Add 1 condition-at-a-time to CR
            else
                Add 2 conditions-at-a-time to CR
            Evaluate CR using the Laplace Estimation
            if accuracy(CR) > 0.7 then
                newCandidateRules = newCandidateRules ∪ CR
        candidateRules = best rule selected from newCandidateRules
    RuleList = RuleList ∪ bestRule
until all examples in the training set are covered
```

Alg. 5.3 shows an example of an algorithm found when training the GGP with nominal data sets. It creates an empty rule, refines it according to the number of examples currently covered by the rule list, and evaluates it using the Laplace estimation. It selects just the best rule to undergo refinements, and requires that the accuracy of the selected rule is greater than 70%. Rules are generated until all the examples in the training set are covered. The main difference among Alg. 5.3

and our baseline comparison algorithms is on the way the rules are refined. In Alg. 5.3, an *if* condition determines whether to add one or two conditions-at-a-time to a candidate rule according to the number of examples covered by the rule list.

Table 5.14: Predictive accuracy rates (%) produced by Alg. 5.3

Data Set	GGP-RIs	OrdCN2	UnordCN2	Ripper	C45Rules
monks-1	100±0	100 ± 0	100 ± 0	93.84 ± 2.93	100 ± 0
mushroom	100±0	100 ± 0	100 ± 0	99.96 ± 0.04	98.8 ± 0.06
wisconsin	94.88±1.02	94.58 ± 0.68	94.16 ± 0.93	93.99 ± 0.63	95.9 ± 0.56
promoters	71.82±5.06	81.9 ± 4.65	74.72 ± 4.86	78.18 ± 3.62	83.74 ± 3.46
splice	88.36±0.67	90.32 ± 0.74	74.82 ± 2.94	93.88 ± 0.41	89.66 ± 0.78

When the rule list starts to be built, there is a lot of examples available in the training set, and the algorithm tries to find the best combination of two conditions to be added to the rule antecedent at once. As the number of examples covered by the rules in the list is increased and reaches 90% of the total number of examples, the algorithm starts adding just one condition-at-a-time to the new candidate rules. The algorithm pre-prune rules just by ignoring the ones with accuracy smaller than 70%, and performs a greedy search.

It should be noted that the strategy of switching the number of conditions added (in a single step) to the current rule depending on the number of examples covered by the rule list is *innovative*. In general, manually-designed rule induction algorithms do not have this flexibility, they simply use a fixed number of conditions to be added to the current rule throughout the run of the algorithm, regardless of how many examples are covered by the rule list. Furthermore, the flexible strategy discovered by the GGP seems to have a rationale: adding two conditions-at-a-time is less greedy (and potentially more effective) than adding one condition-at-a-time. However, as the number of examples not covered by the rule list decreases, it is intuitively easier to find one attribute-value pair with good statistical support to be added to the current rule than a combination of two attribute-values.

Table 5.14 shows the predictive accuracies obtained by Alg. 5.3 in the data sets of the meta-test set using a 5-fold cross-validation procedure, and its comparison to the baseline algorithms.

The results are very similar to the ones reported in Table 5.9, and comparisons among the algorithms show exactly the same pattern of results: the GGP-RIs obtained accuracy rates statistically better (at the 0.01 significance level) than

C4.5Rules in *mushroom* and *splice*, whilst GGP-RI obtained accuracy rates statistically worse than Ripper in the *splice* data set.

Algorithm 5.4: Example of a Rule list Algorithm created by the GGP using Numerical Data Sets

```
RuleList = ∅
repeat
    Divide the training data in Grow and Prune
    bestRule = an empty rule
    candidateRules = ∅
    candidateRules = candidateRules ∪ bestRule
    while candidateRules ≠ ∅ do
        for each candidateRule CR do
            newCandidateRules = ∅
            Add 1 condition-at-a-time to CR
            Evaluate CR using the Laplace estimation in Grow
            newCandidateRules = newCandidateRules ∪ CR
        candidateRules = 8 best rules selected from newCandidateRules
    bestRule' = Pre-prune bestRule by removing its last added condition
    Evaluate bestRule' using Laplace estimation in Prune
    if bestRule' better than bestRule then
        bestRule = bestRule'
    RuleList = RuleList ∪ bestRule
until 99% of the examples in Grow are covered
```

Table 5.15: Predictive accuracy rates (%) produced by Alg. 5.4

Data Set	GGP-RIs	OrdCN2	UnordCN2	Ripper	C45Rules
crx	84.78±0.95	80.16 ± 1.27	80.6 ± 0.93	84.37 ± 1.21	84.82 ± 1.53
ionosphere	86.34±3.37	87.6 ± 2.76	90.52 ± 2.03	89.61 ± 1.75	89.06 ± 2.71
segment	94.24±0.58	95.38 ± 0.28	85.26 ± 0.87	95.44 ± 0.32	88.16 ± 7.72
sonar	70.2±4.69	70.42 ± 2.66	72.42 ± 1.4	72.88 ± 4.83	72.4 ± 2.68
heart-c	73.94±3.08	77.9 ± 1.96	77.54 ± 2.85	77.53 ± 1.1	74.2 ± 5.43

Alg. 5.4 shows the pseudo-code of one rule induction algorithm produced by the GGP when trained with numerical data sets, and Table 5.15 reports its predictive accuracies. Alg. 5.4 works as follows. It first divides the available training data into two sets: a grow set and a prune set, as Ripper does. The grow set is used to create a rule, and the prune set is used to pre-prune a rule before inserting it into the final rule set. At each iteration, the examples covered in both grow and prune sets are removed from them, and these two sets are merged and redivided.

Table 5.16: Total number of rules produced and pruned by the GGP-RI described in Alg. 5.4

Data Set	Produced	Pruned
pima	24	15
hepatitis	6	3
vowel	37	15
vehicle	27	16
glass	14	5
crx	15.2	9
ionosphere	7.6	2.8
segment	19.4	7.8
sonar	7.4	3.8
heart-c	8.2	2.8

The algorithm starts the rule induction process with an empty rule, adds one condition-at-a-time to it and evaluates the new rule using the Laplace estimation. At each iteration, the 8 best rules are kept into the refinement process. After one rule is produced, the algorithm tries to prune it by removing the last condition added to it, and evaluates the rule using again the Laplace estimation, but in the prune set. The induction process terminates when 99% of the examples in the grow set are covered, in order to avoid over-fitting of the rule set to the training data.

Analyzing the accuracies showed in Table 5.15, the GGP-RI described in Alg. 5.4 obtained significantly better accuracies than Unordered-CN2 in two data sets: *crx* and *segment*. The Unordered-CN2 classifier, as explained earlier, has a really bad accuracy when compared to other algorithms in the data set *segment*, and so it was easy for all the GGP-RIs produced by the GGP to obtain better accuracies than it. The same is not true when analyzing the performance of both versions of CN2 in *crx*, and the GGP-RI was significantly superior to both versions of CN2 in this data set.

To conclude, we can say that Alg. 5.4 applies the same growing and pruning phases used by algorithms like IREP [61] and Ripper, with a difference: while other algorithms build a rule which covers no negative examples and select just the best rule to undergo further refinements, Alg. 5.4 selects 8 rules and add conditions to it until a rule better than the current best one can be found. After that it removes just the last condition from the produced rule. At first glance, one could argue that maybe Alg. 5.4 rarely improves the rule by trying to pre-prune

Algorithm 5.5: Example of a Decision List Algorithm created by the GGP using both Nominal and Numerical Data Sets

```
RuleList = ∅
repeat
    bestRule = an empty rule
    candidateRules = ∅
    candidateRules = candidateRules ∪ bestRule
    while candidateRules ≠ ∅ do
        for each candidateRule CR do
            newCandidateRules = ∅
            Add 2 conditions-at-a-time to CR
            Evaluate CR using the Laplace estimation
            newCandidateRules = newCandidateRules ∪ CR
        candidateRules = 5 best rules selected from newCandidateRules
    if accuracy(bestRule) < 0.7 then break
    else RuleList = RuleList ∪ bestRule
until all examples in the training set are covered
```

it, and it is simply a version of CN2 with the beam size (or star, as named by the authors in [26]) set to 8. However, that is not the case. Table 5.16 shows the number of rules produced for each data set followed by the number of rules which were improved by the pruning process. The top part of the table shows the number of rules produced and pruned by a single execution of the algorithm in each of the 5 data sets belonging to the meta-training set. The bottom part of the table shows the average number of rules produced and pruned in each of the 5 data sets in the meta-test set, using a 5-fold cross-validation process.

We have run a new experiment comparing the results of the algorithm described in Alg. 5.4 with the same algorithm, but excluding the pruning process. Both algorithms obtained similar accuracy rates in all data sets, according to a t-test with 0.05. Hence, we can say that Alg. 5.4 has a limited degree of innovation because, although it does not improve the accuracy of the models produced by CN2, the rule models generated might be more compact due to the pruning process.

We finally present one algorithm produced by the GGP when trained and tested with a total of 20 data sets containing both nominal and numerical attributes. 3 out of 5 algorithms produced by the GGP-0.7 in this case shared one characteristic: they added two conditions instead of one condition at-a-time to an empty rule, as shown in Alg. 5.5. Alg. 5.5 is a version of CN2 where two conditions are added to a rule at-a-time. The only other difference with respect

Table 5.17: Predictive accuracy rates (%) produced by Alg. 5.5

Data Set	GGP-RI	OrdCN2	UnordCN2	Ripper	C45Rules
crx	66.26±16.58	80.16 ± 1.27	80.6 ± 0.93	84.37 ± 1.21	84.82 ± 1.53
segment	95.12±0.42	95.38 ± 0.28	85.26 ± 0.87	95.44 ± 0.32	88.16 ± 7.72
sonar	74.5±0.96	70.42 ± 2.66	72.42 ± 1.4	72.88 ± 4.83	72.4 ± 2.68
heart-c	71.34±3.09	77.9 ± 1.96	77.54 ± 2.85	77.53 ± 1.1	74.2 ± 5.43
ionosphere	87.5±2.48	87.6 ± 2.76	90.52 ± 2.03	89.61 ± 1.75	89.06 ± 2.71
monks-1	100±0	100 ± 0	100 ± 0	93.84 ± 2.93	100 ± 0
mushroom	100±0	100 ± 0	100 ± 0	99.96 ± 0.04	98.8 ± 0.06
wisconsin	95.3±1.5	94.58 ± 0.68	94.16 ± 0.93	93.99 ± 0.63	95.9 ± 0.56
promoters	76.56±4.88	81.9 ± 4.65	74.72 ± 4.86	78.18 ± 3.62	83.74 ± 3.46
splice	87.74±0.77	90.32 ± 0.74	74.82 ± 2.94	93.88 ± 0.41	89.66 ± 0.78

to CN2 lies on the condition used to stop inserting rules to the model. Once the rule induction algorithm cannot find a rule with predictive accuracy superior to 70%, it stops and returns the current rule list.

Table 5.17 shows the predictive accuracies obtained by Alg. 5.5 in the data sets of the meta-test set using a 5-fold cross-validation procedure. Again, in terms of comparison with the baseline algorithms, this algorithm produced results similar to the results presented in Table 5.13. It has significantly better accuracies than the weakest algorithms, obtaining a significant better result in three cases, and worse accuracy than Ripper in *splice*.

In summary, looking at the three algorithms presented in this section, we can say that Alg. 5.3 is the most innovative one, in the sense that it changes the way rules are refined according to the amount of training data that is covered by the current rule list. This flexibility is an innovation over manually-designed algorithms, which in general do not exhibit such a flexible, data-driven behavior. But why most of the algorithms produced are so similar to CN2, for example?

The UCI data sets [104] are very popular in the machine learning community, and they have been used to benchmark classification algorithms for a long time. To a certain extent, most of the rule induction algorithms were first designed or later modified targeting these data sets. The fact that the evolved rule induction algorithms are so similar to CN2, for instance, is evidence that CN2 is actually one of the best algorithms in terms of average predictive accuracy in a set of data sets available in the UCI repository. At the same time, as the rule induction algorithms produced by the GGP showed, there are many other variations of the basic sequential covering pseudo-code which obtain accuracies competitive to the

ones produced by CN2, Ripper or C4.5Rules. In general, the evolved algorithms did not obtain significantly better accuracies than the baseline classification algorithms, but the former obtained slightly better results than the latter, overall. This can be observed in Tables 5.9, 5.11 and 5.13, which overall contain more significant wins (dark gray cells) than significant losses (light gray cells) for the evolved algorithms.

5.5 Meta-Training Set Variations

In Section 5.2 we explained that the GGP needed two sets of elements in order to automatically evolve rule induction algorithms: the GP evolution parameters – such as population size, number of generations and crossover and mutation rates, and the number of data sets (and the data sets themselves) used in the meta-training set of the GGP algorithm.

As the data sets in the meta-training set are used to calculate the fitness of a GGP individual (candidate rule induction algorithm), they are a key point in the GGP evolution process. Changing the data sets in the meta-training set can significantly change the value of the fitness of the GGP individuals and, consequently, the final rule induction algorithm produced by the GGP.

Hence, in this section we study the impact of changing the number of data sets in the meta-training set. As in the previous sections, we run the experiments in three phases: the first one involving only data sets with nominal attributes, the second one using data sets with at least one numerical attribute (again, called numerical data sets for short), and finally we merged the data sets used in the previous experiments to create a third set of experiments. In the first two sets of experiments, we vary the number of data sets in the meta-training (meta-test) sets from the original numbers 5(5) to 3(7), 4(6), 6(4) and 7(3), respectively. In the third experiment, which originally used 10 data sets in the meta-training set and 10 data sets in the meta-test set, we vary the size of the meta-training set to 6, 8, 12 and 14 (making the size of the meta-test set 14, 12, 8 and 6, respectively). The choice of which data sets would be removed or added to the original meta-training set was, as before, based on execution time. Note that all the GGP parameter values were fixed in all the experiments reported in this section, and the crossover/mutation rates used here are the same ones chosen when presenting the comparisons among the GGP-RIs and other baseline algorithms in Section 5.3: crossover/mutation rates of 0.5/0.45 for experiments with nominal data sets, and

Table 5.18: Comparing predictive accuracy rates (%) for the nominal data sets in the meta-test set when training the GGP with 3 data sets

Data Set	GGP-RIs	OrdCN2	UnordCN2	Ripper	C45Rules
monks-1	100±0	100 ± 0	100 ± 0	93.84 ± 2.93	100 ± 0
mushroom	99.99±0	100 ± 0	100 ± 0	99.96 ± 0.04	98.8 ± 0.06
wisconsin	94.76±1.03	94.58 ± 0.68	94.16 ± 0.93	93.99 ± 0.63	95.9 ± 0.56
promoters	77.83±4.47	81.9 ± 4.65	74.72 ± 4.86	78.18 ± 3.62	83.74 ± 3.46
splice	88.32±0.43	90.32 ± 0.74	74.82 ± 2.94	93.88 ± 0.41	89.66 ± 0.78
zoo	92.52±0.96	92.64 ± 1.33	92.52 ± 2.21	89.47 ± 1.66	92.56 ± 1.45
monks-3	97.07±0.54	97.46 ± 0.74	99.1 ± 0.4	98.54 ± 0.46	94 ± 4.89

Table 5.19: Comparing predictive accuracy rates (%) for the nominal data sets in the meta-test set when training the GGP with 4 data sets

Data Set	GGP-RIs	OrdCN2	UnordCN2	Ripper	C45Rules
monks-1	100±0	100 ± 0	100 ± 0	93.84 ± 2.93	100 ± 0
mushroom	99.98±0.01	100 ± 0	100 ± 0	99.96 ± 0.04	98.8 ± 0.06
wisconsin	94.64±0.78	94.58 ± 0.68	94.16 ± 0.93	93.99 ± 0.63	95.9 ± 0.56
promoters	77.44±3.49	81.9 ± 4.65	74.72 ± 4.86	78.18 ± 3.62	83.74 ± 3.46
splice	88.79±0.46	90.32 ± 0.74	74.82 ± 2.94	93.88 ± 0.41	89.66 ± 0.78
zoo	90.76±1.05	92.64 ± 1.33	92.52 ± 2.21	89.47 ± 1.66	92.56 ± 1.45

0.7/0.25 for experiments using both numerical and nominal and numerical data sets.

First we present the results for the four variations introduced in the meta-training set in the set of experiments using only nominal data sets. In relation to the data sets used in the experiments with 5 data sets in each of the meta-data sets, we observe that *monks-3* and *zoo* were moved from the meta-training to the meta-test set for the experiments with 3 data sets in the meta-training set. In the case were 4 data sets remained in the meta-training set, just *zoo* was moved to the meta-test set. For experiments with 6 and 7 data sets in the meta-training set, *monks-1* was included in the meta-training set of both experiments, and *wisconsin* completed the 7 data sets needed in the meta-training set for the last experiment.

Tables 5.18, 5.19, 5.20 and 5.21 show the results for nominal data sets using 3(7), 4(6), 6(4) and 7(3) data sets in the meta-training (meta-test) set, respectively. The tables have the same structure as the ones shown in the previous section, where the first column (after the column with data set names) shows

Table 5.20: Comparing predictive accuracy rates (%) for the nominal data sets in the meta-test set when training the GGP with 6 data sets

Data Set	GGP-RIs	OrdCN2	UnordCN2	Ripper	C45Rules
mushroom	99.96±0.01	100 ± 0	100 ± 0	99.96 ± 0.04	98.8 ± 0.06
wisconsin	95.56±0.56	94.58 ± 0.68	94.16 ± 0.93	93.99 ± 0.63	95.9 ± 0.56
promoters	81.17±1.48	81.9 ± 4.65	74.72 ± 4.86	78.18 ± 3.62	83.74 ± 3.46
splice	89.19±0.41	90.32 ± 0.74	74.82 ± 2.94	93.88 ± 0.41	89.66 ± 0.78

Table 5.21: Comparing predictive accuracy rates (%) for the nominal data sets in the meta-test set when training the GGP with 7 data sets

Data Set	GGP-RIs	OrdCN2	UnordCN2	Ripper	C45Rules
mushroom	99.98±0.0001	100 ± 0	100 ± 0	99.96 ± 0.04	98.8 ± 0.06
promoters	80.57±3.55	81.9 ± 4.65	74.72 ± 4.86	78.18 ± 3.62	83.74 ± 3.46
splice	89.1±0.46	90.32 ± 0.74	74.82 ± 2.94	93.88 ± 0.41	89.66 ± 0.78

the average accuracy obtained by the GGP-RIs in 5 different runs (with different random seeds), and it is followed by the results of runs of Ordered-CN2, Unordered-CN2, Ripper and C4.5Rules (using default parameter values in all these algorithms). All the results were obtained using a 5-fold cross-validation procedure, and the numbers after the symbol "±" are standard deviations. Results were compared using a two-tailed statistical t-test with significance levels 0.05 and 0.01, but the comparisons shown in the tables consider the significance level 0.05. Cells in dark gray represent significant wins of the GGP-RIs against the corresponding baseline algorithm, while light gray cells represent GGP-RIs' significant losses.

Looking at these 4 tables, and also considering the results obtained when training the GGP with 5 data sets in the meta-training set, presented in Table 5.14, we observe an interesting fact: the GGPs trained with 3 and 4 data sets produced results slightly better than the ones trained with a larger number of data sets. In Table 5.18, the GGP-RIs obtained two results significantly better and two results significantly worse than the baseline algorithms out of 28 results (7 data sets x 4 baseline algorithms), while in Tables 5.19 they also obtained two results significantly better and only one result significantly worse than the baseline algorithms out of 24 results. In turn, in Tables 5.20 and 5.21, the GGP-RIs also obtained two significantly better results but three significantly worse results than the baseline algorithms. However, Tables 5.20 and 5.21 show 16 and 9 results, respectively.

Observing the results in the aforementioned tables, we notice that, as in previous experiments, the significantly better accuracies of GGP-RIs (according to a 0.05 significance level) always occur in the same data sets when compared with the same baseline classifiers. As explained in Section 5.4, these are classifiers which obtain accuracy rates much worse than the ones obtained by any of the other baseline algorithms. Now let us consider the cases where the GGP-RIs obtained significantly worse accuracies than the baseline algorithms. Tables 5.20 and 5.21 present exactly the same qualitative results when comparing the GGP-RIs with the baseline algorithms: accuracy significantly smaller than Ripper in *splice* and both versions of CN2 in *mushroom*. Tables 5.18 and 5.19 also share the case of Ripper in *splice* with the previous tables, which was explained before (Ripper obtains a much higher accuracy in this data set than the other baseline classifiers).

Now let us analyze the significantly smaller accuracies obtained by the GGP-RIs when compared with both versions of CN2 in *mushroom*. This case can be considered a special one. For *mushroom*, the accuracies of both versions of CN2 are 100%. We can observe that the results of the GGP-RIs varied from 99.96 to 99.98. This reflects the fact that from the 5 different algorithms found in 5 different runs of the GGP, only 1 could not obtain an accuracy of 100% in a 5-fold cross-validation for *mushroom*. However, as the value of these accuracies is close to 100%, one significant loss out of 5 runs reflects in the value of the average over all runs.

In Table 5.18, besides the case of *splice* in Ripper, a second significantly worse accuracy is observed in the comparison of the GGP-RIs with the result of Unordered-CN2 in *monks-3*. This is the first experiment where *monks-3* is used in the meta-test set, and in this case the GGP-RIs could not obtain a significantly better accuracy than Unordered-CN2, although the former is comparable with all the other baseline algorithms.

Moreover, the results of the experiments using 6(4) and 7(3) data sets in the meta-training (meta-test) set are also valid with a significance level of 0.01. In the case of the results with 3 data sets in the meta-training set, just the win in *mushroom* and the loss in *splice* are now significant. At last, the results where the GGP-RIs significantly win over the baseline algorithms for experiments with 4 data sets in the meta-training set do not change, but the significant losses are reduced from 3 to 1 (both results of CN2 in *mushroom* are not significantly better than the GGP-RIs results anymore).

In conclusion, experiments using nominal data sets showed that varying the

Table 5.22: Comparing predictive accuracy rates (%) for the numerical data sets in the meta-test set when training the GGP with 3 data sets

Data Set	GGP-RIs	OrdCN2	UnordCN2	Ripper	C45Rules
crx	81.89±0.5	80.16 ± 1.27	80.6 ± 0.93	84.37 ± 1.21	84.82 ± 1.53
ionosphere	86.81±2.13	87.6 ± 2.76	90.52 ± 2.03	89.61 ± 1.75	89.06 ± 2.71
segment	95.3±0.38	95.38 ± 0.28	85.26 ± 0.87	95.44 ± 0.32	88.16 ± 7.72
sonar	71.28±1.89	70.42 ± 2.66	72.42 ± 1.4	72.88 ± 4.83	72.4 ± 2.68
heart-c	75.6±1.37	77.9 ± 1.96	77.54 ± 2.85	77.53 ± 1.1	74.2 ± 5.43
hepatitis	82.74±1.16	81.94 ± 5.02	83.34 ± 1.83	86.03 ± 1.14	83.36 ± 0.9
pima	68.46±1.01	69.34 ± 2.13	74.6 ± 0.38	73.91 ± 1.65	71.04 ± 1.67

Table 5.23: Comparing predictive accuracy rates (%) for the numerical data sets in the meta-test set when training the GGP with 4 data sets

Data Set	GGP-RIs	OrdCN2	UnordCN2	Ripper	C45Rules
crx	81.35±0.69	80.16 ± 1.27	80.6 ± 0.93	84.37 ± 1.21	84.82 ± 1.53
ionosphere	86.4±1.69	87.6 ± 2.76	90.52 ± 2.03	89.61 ± 1.75	89.06 ± 2.71
segment	95.17±0.42	95.38 ± 0.28	85.26 ± 0.87	95.44 ± 0.32	88.16 ± 7.72
sonar	70.56±1.26	70.42 ± 2.66	72.42 ± 1.4	72.88 ± 4.83	72.4 ± 2.68
heart-c	75.99±1.22	77.9 ± 1.96	77.54 ± 2.85	77.53 ± 1.1	74.2 ± 5.43
pima	69.65±0.78	69.34 ± 2.13	74.6 ± 0.38	73.91 ± 1.65	71.04 ± 1.67

Table 5.24: Comparing predictive accuracy rates (%) for the numerical data sets in the meta-test set when training the GGP with 6 data sets

Data Set	GGP-RIs	OrdCN2	UnordCN2	Ripper	C45Rules
crx	82.5±0.61	80.16 ± 1.27	80.6 ± 0.93	84.37 ± 1.21	84.82 ± 1.53
segment	89.44±0.94	95.38 ± 0.28	85.26 ± 0.87	95.44 ± 0.32	88.16 ± 7.72
sonar	72.39±1.5	70.42 ± 2.66	72.42 ± 1.4	72.88 ± 4.83	72.4 ± 2.68
heart-c	74.88±2.86	77.9 ± 1.96	77.54 ± 2.85	77.53 ± 1.1	74.2 ± 5.43

Table 5.25: Comparing predictive accuracy rates (%) for the numerical data sets in the meta-test set when training the GGP with 7 data sets

Data Set	GGP-RIs	OrdCN2	UnordCN2	Ripper	C45Rules
crx	84.18±1.05	80.16 ± 1.27	80.6 ± 0.93	84.37 ± 1.21	84.82 ± 1.53
segment	91.54±0.49	95.38 ± 0.28	85.26 ± 0.87	95.44 ± 0.32	88.16 ± 7.72
sonar	70.47±2.21	70.42 ± 2.66	72.42 ± 1.4	72.88 ± 4.83	72.4 ± 2.68

number of data sets in the meta-training set from 3 to 7 do not have a great impact in the results obtained. In general, the GGP-RIs usually obtain predictive accuracies competitive with well-known human-designed algorithms. Nevertheless, before speculating about why changing the number of data sets in the meta-training set of the experiments with nominal data sets did not affect the results obtained by the evolved rule induction algorithms, we need to check if the same occurs when using different kinds of data sets in the meta-training set.

Regarding experiments with numerical data sets, Tables 5.22, 5.23, 5.24 and 5.25 show the results obtained when using 3(7), 4(6), 6(4) and 7(3) data sets in the meta-training (meta-test) set, respectively. In the experiments with 3 data sets in the meta-training set, *pima* and *hepatitis* were moved from the meta-training to the meta-test set. In the case where 4 data sets remained in the meta-training set, just *pima* was moved to the meta-test set. For experiments with 6 and 7 data sets in the meta-training set, *ionosphere* was included in the meta-training set of both experiments, and *heart-c* completed the 7 data sets needed for the last experiment. Let us first compare the results of these experiments with the ones obtained when using 5 data sets in the meta-training set, whose results were presented in Table 5.11.

From the results shown in Table 5.11, recall that there was just one comparison in which the GGP-RIs obtained a significantly better accuracy than the baseline algorithms: Unordered-CN2 in *segment* (in all the other cases the GGP-RIs' accuracies were statistically as good as the ones found by the baseline algorithms). This same result is presented in Tables 5.22, 5.23, 5.24 and 5.25. Besides, in Table 5.25, the GGP-RIs are also significantly better than both versions of CN2 when considering the data set *crx*. However, all these tables also present comparisons in which the GGP-RIs obtained significantly worse accuracies than other algorithms. In Table 5.22 this is the case in *pima*, while in Table 5.23 the GGP-RIs significantly lose in both *pima* and *crx*. In Tables 5.24 and 5.25 this is the case of *segment*.

Let us start with the *pima* data set. Table 5.22 represents the first time this data set appeared in the meta-test set, and in both Tables 5.22 and 5.23 the GGP-RIs obtained, in this data set, an accuracy significantly worse than the accuracies of Unordered-CN2 and Ripper. In Table 5.23 the GGP-RIs obtained a significantly worse accuracy than Ripper for the data set *crx*.

Now let us look at the *segment* data set. We can say that increasing the number of data sets in the meta-training set reduced dramatically the performance of the

Table 5.26: Comparisons of the number of rules and predictive accuracies per class generated by both versions of CN2 in the data set *segment*

	Ordered-CN2		Unordered-CN2	
Class	# Rules	Accuracy (%)	# Rules	Accuracy (%)
C1	3	96.4	3	98.2
C2	1	100	1	100
C3	6	94.5	9	90.9
C4	5	**95.5**	6	**61.8**
C5	8	**90**	13	**65.5**
C6	2	98.2	2	97.3
C7	1	98.2	1	98.2
Overall	26	96.1	35	87.4

GGP-RIs in *segment*. When training the algorithm with 5 data sets, the GGP-RIs obtained an average accuracy of 94.75% (± 0.41), decreasing the number of data sets to 3 and 4 changed the GGP-RIs' average accuracies in *segment* to 95.3% (± 0.38) and 95.17% (± 0.42), respectively. However, increasing the number of data sets in the meta-training set to 6 and 7 changed the GGP-RIs' average accuracies to 89.44% (± 0.94) and 91.54% (± 0.49). Why did that happen?

An analysis of the actual GGP-RIs produced might give an explanation for that. In the case of the algorithms which generated the results presented in Table 5.24, 4 out of 5 produced unordered rule sets, rather than ordered rule lists in the *segment* data set. For the rule induction algorithms which generated the results in Table 5.25, 3 out of 5 also produced unordered rule sets. As we observe in both tables, the results produced by Unordered-CN2 (i.e. CN2 producing unordered rule sets) are much inferior than the ones produced by rule induction algorithms which generate ordered rule lists. But why unordered rule sets perform much worse than ordered rule lists particularly in *segment*?

Segment (originally called *segmentation* in the UCI data set repository [104]) is a data set about classifying an image into one of the following seven classes: brick-face, sky, foliage, cement, window, path or grass. However, the class labels associated with each instance were produced by first running a segmentation algorithm and then manually labeling each region (a 3x3 pixel) in a computer screen. As already pointed out by other researchers, this data set presents a high degree of class noise [16]. This class noise can be explained by two factors. First, one region in the image might have elements from two classes (grass and sky, for example), where one element predominates. Secondly, as the images were labeled by

Table 5.27: Comparing predictive accuracy rates (%) for the nominal and numerical data sets in the meta-test set when training the GGP with 6 data sets

Data Set	GGP-RIs	OrdCN2	UnordCN2	Ripper	C45Rules
crx	81.07±0.9	80.16 ± 1.27	80.6 ± 0.93	84.37 ± 1.21	84.82 ± 1.53
segment	95.14±0.39	95.38 ± 0.28	85.26 ± 0.87	95.44 ± 0.32	88.16 ± 7.72
sonar	73.12±1.44	70.42 ± 2.66	72.42 ± 1.4	72.88 ± 4.83	72.4 ± 2.68
heart-c	77.18±0.9	77.9 ± 1.96	77.54 ± 2.85	77.53 ± 1.1	74.2 ± 5.43
ionosphere	87.33±2.17	87.6 ± 2.76	90.52 ± 2.03	89.61 ± 1.75	89.06 ± 2.71
pima	63.86±3.16	69.34 ± 2.13	74.6 ± 0.38	73.91 ± 1.65	71.04 ± 1.67
hepatitis	82.51±1.89	81.94 ± 5.02	83.34 ± 1.83	86.03 ± 1.14	83.36 ± 0.9
monks-1	100±0	100 ± 0	100 ± 0	93.84 ± 2.93	100 ± 0
mushroom	100±0	100 ± 0	100 ± 0	99.96 ± 0.04	98.8 ± 0.06
wisconsin	95.53±0.74	94.58 ± 0.68	94.16 ± 0.93	93.99 ± 0.63	95.9 ± 0.56
promoters	81.61±3.08	81.9 ± 4.65	74.72 ± 4.86	78.18 ± 3.62	83.74 ± 3.46
splice	88.76±0.38	90.32 ± 0.74	74.82 ± 2.94	93.88 ± 0.41	89.66 ± 0.78
zoo	92.89±0.87	92.64 ± 1.33	92.52 ± 2.21	89.47 ± 1.66	92.56 ± 1.45
monks-3	96.89±0.78	97.46 ± 0.74	99.1 ± 0.4	98.54 ± 0.46	94 ± 4.89

humans, there is always some subjectivity introduced into the process. Defining the boundaries of a region, for instance, might be tricky in some cases.

Table 5.26 shows the number of rules found per class by both versions of CN2, and also lists the predictive accuracy obtained by each classifier by class and overall. As indicated by the figures in bold, the accuracies obtained by Unordered-CN2 for classes 4 (cement) and 5 (window) are much lower than the ones obtained by Ordered-CN2. In these cases, many examples which actually belonged to classes 4 (cement) and 5 (window) were classified as belonging to class 1 (brickface).

One explanation of why rule lists are much better predictors than rule sets for *segment* is that, as negative examples covered by one rule are not excluded from the training set during the creation of the rule set model, the noise examples are all represented by rules with very low coverage. The set of unordered rules has 7 rules with coverage lower than 5 examples, while the ordered rule list has just 2 such rules.

In summary, when varying the number of data sets in the meta-training sets for the experiments with numerical data sets, it was observed that the number of cases in which the GGP-RIs obtained significantly better results than the baseline algorithms remained the same for all experiments but the one where the GGP was trained with 7 data sets. There was also a small increase in the number of

Table 5.28: Comparing predictive accuracy rates (%) for the nominal and numerical data sets in the meta-test set when training the GGP with 8 data sets

Data Set	GGP-RIs	OrdCN2	UnordCN2	Ripper	C45Rules
crx	80.57±0.9	80.16 ± 1.27	80.6 ± 0.93	84.37 ± 1.21	84.82 ± 1.53
ionosphere	86.75±2.12	87.6 ± 2.76	90.52 ± 2.03	89.61 ± 1.75	89.06 ± 2.71
segment	94.99±0.41	95.38 ± 0.28	85.26 ± 0.87	95.44 ± 0.32	88.16 ± 7.72
sonar	72.73±1.74	70.42 ± 2.66	72.42 ± 1.4	72.88 ± 4.83	72.4 ± 2.68
heart-c	76.38±0.82	77.9 ± 1.96	77.54 ± 2.85	77.53 ± 1.1	74.2 ± 5.43
pima	68.6±0.94	69.34 ± 2.13	74.6 ± 0.38	73.91 ± 1.65	71.04 ± 1.67
monks-1	99.82±0.18	100 ± 0	100 ± 0	93.84 ± 2.93	100 ± 0
mushroom	100±0	100 ± 0	100 ± 0	99.96 ± 0.04	98.8 ± 0.06
wisconsin	95.76±0.75	94.58 ± 0.68	94.16 ± 0.93	93.99 ± 0.63	95.9 ± 0.56
zoo	92.65±1.12	92.64 ± 1.33	92.52 ± 2.21	89.47 ± 1.66	92.56 ± 1.45
promoters	80.23±3.29	81.9 ± 4.65	74.72 ± 4.86	78.18 ± 3.62	83.74 ± 3.46
splice	89.04±0.48	90.32 ± 0.74	74.82 ± 2.94	93.88 ± 0.41	89.66 ± 0.78

cases in which the GGP-RIs obtained significantly worse results than the baseline algorithms.

Generally speaking, as can be observed from Tables 5.22 through 5.25, in all the experiments the GGP-RIs obtained one significantly better result and two significantly worse results than the baseline algorithms (with the exceptions of the GGP-RIs trained with 7 and 4 data sets, which obtained two significantly better and three significantly worse results, respectively).

At last, we present the results varying the number of data sets in the meta-data sets using both the nominal and numerical data sets employed in the previous experiments. Tables 5.27, 5.28, 5.29 and 5.30 present the results using 6(14), 8(12), 12(8) and 14(6) data sets in the meta-training (meta-test) set, respectively.

Comparing these results with the ones presented in Table 5.13 (results of the GGP trained with 10 data sets), all the results are consistent: the cases in which the GGP-RIs obtained significantly better/worse accuracies than the baseline algorithms in particular data sets remain the same. However, as in the experiments performed with numerical data sets, when *pima* is inserted in the meta-test set (in experiments using 6 and 8 data sets in the meta-training set), the GGP-RIs obtained significantly worse accuracies than Unordered-CN2 and Ripper. This can be observed in Tables 5.27 and 5.28. Interestingly, as in Table 5.24 (results obtained by the GGP when trained with 6 numerical data sets), in Tables 5.27, 5.28 and 5.30, the GGP-RIs obtained accuracies significantly worse than the ones obtained by Ripper in *crx*. The same is true for the accuracies of the GGP-RIs

Table 5.29: Comparing predictive accuracy rates (%) for the nominal and numerical data sets in the meta-test set when training the GGP with 12 data sets

Data Set	GGP-RIs	OrdCN2	UnordCN2	Ripper	C45Rules
crx	78.12±3.37	80.16 ± 1.27	80.6 ± 0.93	84.37 ± 1.21	84.82 ± 1.53
segment	95.11±0.43	95.38 ± 0.28	85.26 ± 0.87	95.44 ± 0.32	88.16 ± 7.72
sonar	73.69±2.4	70.42 ± 2.66	72.42 ± 1.4	72.88 ± 4.83	72.4 ± 2.68
heart-c	75.08±2.44	77.9 ± 1.96	77.54 ± 2.85	77.53 ± 1.1	74.2 ± 5.43
mushroom	100±0	100 ± 0	100 ± 0	99.96 ± 0.04	98.8 ± 0.06
wisconsin	95.9±0.6	94.58 ± 0.68	94.16 ± 0.93	93.99 ± 0.63	95.9 ± 0.56
promoters	81.05±2.94	81.9 ± 4.65	74.72 ± 4.86	78.18 ± 3.62	83.74 ± 3.46
splice	89.3±0.39	90.32 ± 0.74	74.82 ± 2.94	93.88 ± 0.41	89.66 ± 0.78

presented in Table 5.28 when running C4.5Rules in *crx*.

Moreover, in Section 5.4 we observed that the results of experiments using both nominal and numerical data sets presented exactly the same results as the individual experiments when using only nominal or numerical data sets. This scenario is partially repeated in here. The exceptions are the significant losses for the data sets *mushroom* and *segment*. In the experiments performed with 6 and 7 data sets in the meta-training set using nominal data sets (Tables 5.20 and 5.21) and numerical data sets (Tables 5.24 and 5.25), respectively, the accuracies obtained in *mushroom* and *segment* by the GGP-RIs were significantly worse than the ones obtained by two of the baseline methods. This is not the case for the accuracies reported for these data sets in experiments using 12 and 14 data sets in the meta-training set (Tables 5.29 and 5.30).

Concerning *mushroom*, as already explained, the significantly worse accuracies obtained by the GGP-RIs in Tables 5.20 and 5.21, when compared to both versions of CN2 in experiments with the GGP trained with 6 and 7 nominal data sets, were caused by a single GGP-RI out of 5 which did not obtain 100% accuracy when classifying new examples.

In the case of the data set *segment*, rule sets clearly have a much poorer performance than rule list models (as reflected in the results found by Unordered-CN2 when compared with Ordered-CN2 in Tables 5.24 and 5.25). This is confirmed again in the results of the experiments shown in Tables 5.29 and 5.30. In these experiments, all the rule induction algorithms produced by the GGP were rule lists, and consequently competitive with Ordered-CN2 and Ripper.

After these three sets of experiments varying the number of data sets used in the meta-training and meta-test sets, we concluded that, overall, the results

Table 5.30: Comparing predictive accuracy rates (%) for the nominal and numerical data sets in the meta-test set when training the GGP with 14 data sets

Data Set	GGP-RIs	OrdCN2	UnordCN2	Ripper	C45Rules
crx	80.81±1.14	80.16 ± 1.27	80.6 ± 0.93	84.37 ± 1.21	84.82 ± 1.53
segment	95.21±0.43	95.38 ± 0.28	85.26 ± 0.87	95.44 ± 0.32	88.16 ± 7.72
sonar	71.94±1.46	70.42 ± 2.66	72.42 ± 1.4	72.88 ± 4.83	72.4 ± 2.68
mushroom	100±0	100 ± 0	100 ± 0	99.96 ± 0.04	98.8 ± 0.06
promoters	81.72±3.3	81.9 ± 4.65	74.72 ± 4.86	78.18 ± 3.62	83.74 ± 3.46
splice	89.33±0.38	90.32 ± 0.74	74.82 ± 2.94	93.88 ± 0.41	89.66 ± 0.78

produced are not significantly different from the ones produced when using 5 data sets in the meta-training and 5 data sets in the meta-test set. Our intuition was that decreasing the number of data sets in the meta-training set might lead the GGP to create algorithms with a poorer performance in the meta-test set, but that was not the case. The results obtained in experiments with fewer data sets in the meta-training set are in general as good as the ones obtained with more data sets in the meta-training set.

A possible explanation for this fact is that the rule induction algorithms produced with fewer data sets in the meta-training set are simpler than the ones built with a bigger number of data sets in the meta-training set. Simpler algorithms are more likely to be more robust and to have a better performance in very different kinds of data.

5.6 GGP's Grammar Variations

Sections 5.2 and 5.5 showed how sensitive the proposed GGP system is to the set of parameters it requires. In this section we show how the GGP system responds to a change in one of its main components: the grammar. The grammar determines the GGP's search space. Hence, changing the grammar might completely change the rule induction algorithms produced by the system.

It is not easy to judge whether a particular version of the grammar is better or worse than another. There are two main reasons for that. First, the GGP system can adapt according to the functionalities present in the grammar. As will be shown later, turning off the GGP's ability to produce rule induction algorithms following the top-down approach, for instance, led the GGP to produce completely

different but still overall competitive rule induction algorithms. Second, the grammar has a set of core components that form the bases for the construction of any rule induction algorithms. By making use of these components, the system will be able to, most of the time, produce a simple but reasonable rule induction algorithm. For these reasons, we will compare the GGP-RIs generated by the system when using different versions of the grammar by considering the predictive accuracy they obtained in the data sets in the meta-test set. We will first compare them with each other, and later with the four rule induction algorithms used as baseline results so far in this chapter.

Three sets of experiments were executed to evaluate the impact of adding/removing certain production rules to/from the grammar. In the first experiment, we removed from the grammar all its "new components", earlier described in Section 4.2.1. These "new components" correspond to functionalities which were added to the grammar but not used before (to the best of our knowledge) by the manual designed rule induction algorithms. They include the symbols *innerIf*, *MakeFirstRule*, *typicalExample* and *Remove2*.

In a second experiment, we removed from the grammar the symbols responsible for sophisticated pruning techniques: *PostProcess* and *PrePruneRule*. As explained in Chapter 2, pruning is one of the elements which is optional in rule induction algorithms. Although it is present in virtually all the newer rule induction algorithms, we wanted to evaluate how the lack of pruning would impact on the rule induction algorithms generated. Note that simpler ways of pruning rules, such as the ones provided by the non-terminal *StoppingCriterion*, remained in this version of the grammar.

At last, we modified the grammar to force the system to produce only bottom-up rule induction algorithms, by removing the symbols *emptyRule* and *MakeFirstRule* from the grammar. The great majority of rule induction algorithms uses a top-down approach. So the main objective of this experiment was to find out how exclusively bottom-up algorithms would perform in the data sets in the meta-test set.

All the experiments reported in this section were run with the same 20 data sets used in the experiments with both nominal and numerical data sets in the meta-training and meta-test sets, reported in Section 5.2. The GGP parameters are also the same ones defined in that section: population of 100 individuals, evolved for 30 generations, tournament size of 2, crossover rate of 0.7, mutation rate of 0.25 and reproduction rate of 0.05. Table 5.31 shows the predictive accuracies

Table 5.31: Comparing predictive accuracy rates (%) for a GGP run with nominal/numerical data sets in the meta-test set with different versions of the grammar

Data Set	Grammar			
	Original	Basic	NoPrune	BottomUp
crx	77.46±3.8	80.19±1.11	80.14±0.73	81.33±1.14
segment	95.06±0.26	95.95±0.19	94.32±0.24	88.47±1.2
sonar	72.34±1.91	76.38±3.04	74.45±2.64	60.86±0.85
heart-c	76.72±1.5	76.44±1.53	77.37±1.39	75.3±0.96
ionosphere	87.04±2.2	85.72±1.81	86.06±2.18	84.85±1.6
monks-1	99.93±0.07	100±0	100±0	100±0
mushroom	99.98±0.02	100±0	99.9±0.02	99.79±0.1
wisconsin	95.58±0.74	94.61±0.51	94.14±0.44	91.18±0.55
promoters	78.98±2.93	80.16±1.22	74.71±0.76	52.11±1.96
splice	88.68±0.31	90.08±0.44	82.85±0.44	50.72±0.49

obtained by the GGP-RIs in the meta-test set when using the original version of the grammar (from now on referred as GGP-RIs-Original), followed by the results of the three grammar variations described above, named GGP-RIs-Basic, GGP-RIs-NoPrune and GGP-RIs-BottomUp, respectively. As in the other tables of this chapter, cells in dark gray represent results in which the GGP-RIs-Original obtained better results than the other variations according to a Student t-test with 0.05 significance level. In turn, cells in light gray represent a better result of a grammar variation over the GGP-RIs-Original.

As can be observed in Table 5.31, the GGP-RIs-Basic find significantly better accuracies than the GGP-RIs-Original in two data sets, namely *segment* and *splice*. As this basic version of the grammar is simpler than the original one, we might say that it produced simpler rule induction algorithms. As observed in previous experiments in this chapter, *segment* and *splice* are very special data sets (see Section 5.5). For this reason, it is more likely that simpler rule induction algorithms will perform well in these data sets. In contrast, when looking at the accuracies obtained by the GGP-RIs-NoPrune, we notice that they obtained worse results than the GGP-RIs-Original in the data sets *mushroom* and *splice*. Finally, the GGP-RIs-BottomUp obtained significantly worse results than the GGP-RIs-Original in half of the data sets, and competitive results in the other half. Note that all the 5 rule induction algorithms produced by the GGP-Original followed the top-down approach. According to these results, the data sets *crx*, *heart-c*, *ionosphere*, *monks-1* and *mushroom* obtained the same kind of performance with

Table 5.32: Accuracy rates (%) obtained by the GGP in the meta-test set while using a modified version of the grammar which excludes its new components

Data Set	GGP-RIs	Cn2Ord	Cn2Unord	Ripper	C45Rules
crx	80.19±1.11	80.16 ± 1.27	80.6 ± 0.93	84.37 ± 1.21	84.82 ± 1.53
segment	95.95±0.19	95.38 ± 0.28	85.26 ± 0.87	95.44 ± 0.32	88.16 ± 7.72
sonar	76.38±3.04	70.42 ± 2.66	72.42 ± 1.4	72.88 ± 4.83	72.4 ± 2.68
heart-c	76.44±1.53	77.9 ± 1.96	77.54 ± 2.85	77.53 ± 1.1	74.2 ± 5.43
ionosphere	85.72±1.81	87.6 ± 2.76	90.52 ± 2.03	89.61 ± 1.75	89.06 ± 2.71
monks-1	100±0	100 ± 0	100 ± 0	93.84 ± 2.93	100 ± 0
mushroom	100±0	100 ± 0	100 ± 0	99.96 ± 0.04	98.8 ± 0.06
wisconsin	94.61±0.51	94.58 ± 0.68	94.16 ± 0.93	93.99 ± 0.63	95.9 ± 0.56
promoters	80.16±1.22	81.9 ± 4.65	74.72 ± 4.86	78.18 ± 3.62	83.74 ± 3.46
splice	90.08±0.44	90.32 ± 0.74	74.82 ± 2.94	93.88 ± 0.41	89.66 ± 0.78

both top-down and bottom-up algorithms. The top-down approach, however, is more suitable for the other data sets.

The analysis of the results reported in Table 5.31 showed how the GGP system with the modified versions of the grammar performed compared to its original version. Next, Tables 5.32, 5.33 and 5.34 compare the predictive accuracies obtained by the GGP-RIs-Basic, GGP-RIs-NoPrune and GGP-RIs-BottomUp with the four baseline rule induction algorithms used so far in our comparisons: Ordered-CN2, Unordered-CN2, Ripper and C45Rules. In these tables, cells in dark gray represent a significantly better result of the GGP-RIs produced with a modified version of the grammar over the corresponding baseline method (when using a Students' t-test with 0.05 significance level). Cells in light gray represent a significantly better result of the baseline methods over the GGP-RIs.

Recall that, in experiments with the original version of the grammar, the GGP-RIs-Original obtained significantly better results than the Unordered-CN2 in the data sets *segment* and *splice*, and C45Rules in *mushroom*. At the same time, they obtained one significantly worse result than Ripper in the data set *splice*. These results are reproduced in Table 5.32, where the GGP-Basic was used. These results show that a simpler version of the grammar can produce results which are competitive to the ones produced by a more sophisticated version of the grammar. These same results appear again in Table 5.33. However, Table 5.33 also presents 5 other cases where the GGP-RIs-NoPrune obtained significantly worse results than the baseline methods. These results reflect the importance of more sophisticated pruning methods in the current version of the grammar.

Table 5.33: Accuracy rates (%) obtained by the GGP in the meta-test set while using a modified version of the grammar which does not include pruning elements

Data Set	GGP-RIs	Cn2Ord	Cn2Unord	Ripper	C45Rules
crx	80.14±0.73	80.16 ± 1.27	80.6 ± 0.93	84.37 ± 1.21	84.82 ± 1.53
segment	94.32±0.24	95.38 ± 0.28	85.26 ± 0.87	95.44 ± 0.32	88.16 ± 7.72
sonar	74.45±2.64	70.42 ± 2.66	72.42 ± 1.4	72.88 ± 4.83	72.4 ± 2.68
heart-c	77.37±1.39	77.9 ± 1.96	77.54 ± 2.85	77.53 ± 1.1	74.2 ± 5.43
ionosphere	86.06±2.18	87.6 ± 2.76	90.52 ± 2.03	89.61 ± 1.75	89.06 ± 2.71
monks-1	100±0	100 ± 0	100 ± 0	93.84 ± 2.93	100 ± 0
mushroom	99.9±0.02	100 ± 0	100 ± 0	99.96 ± 0.04	98.8 ± 0.06
wisconsin	94.14±0.44	94.58 ± 0.68	94.16 ± 0.93	93.99 ± 0.63	95.9 ± 0.56
promoters	74.71±0.76	81.9 ± 4.65	74.72 ± 4.86	78.18 ± 3.62	83.74 ± 3.46
splice	82.85±0.44	90.32 ± 0.74	74.82 ± 2.94	93.88 ± 0.41	89.66 ± 0.78

Finally, in Table 5.34, in 16 out of 40 comparisons, the GGP-BottonUp obtained significantly worse results than the baseline methods, and only one significantly better result. These figures reinforce the analysis performed in Table 5.31, which showed that some data sets are more suitable to work with the bottom-up approach than others. Interestingly, these results were later confirmed when running experiments aiming to produce rule induction algorithms customized for a single data set. In these experiments, reported in Section 6.2, all the rule induction algorithms customized to the data sets *crx* and *heart-c* followed the bottom-up approach, while the same was true for 20, 15 and 5 algorithms out of 25 tailored to the data sets *monks-1*, *mushroom* and *ionosphere*.

In summary, this section showed that it is not simple to compare different versions of the grammar used for the GGP to guide its search, as the method adapts to the grammar and produces different types of rule induction algorithms according to the functionalities available. It also presented results which demonstrated that a simpler version of the grammar can be as effective as a more sophisticated one, and explained how the grammar can be manipulated to produce a particular kind of rule induction algorithm.

5.7 GGP *versus* a Grammar-based Hill Climbing Search Method

This work proposed a GGP system to search for accurate and innovative rule induction algorithms. The GGP was first chosen as the search method for this

Table 5.34: Accuracy rates (%) obtained by the GGP in the meta-test set while using a modified version of the grammar which produces exclusively bottom-up rule induction algorithms

Data Set	GGP-RIs	Cn2Ord	Cn2Unord	Ripper	C45Rules
crx	81.33±1.14	80.16 ± 1.27	80.6 ± 0.93	84.37 ± 1.21	84.82 ± 1.53
segment	88.47±1.2	95.38 ± 0.28	85.26 ± 0.87	95.44 ± 0.32	88.16 ± 7.72
sonar	60.86±0.85	70.42 ± 2.66	72.42 ± 1.4	72.88 ± 4.83	72.4 ± 2.68
heart-c	75.3±0.96	77.9 ± 1.96	77.54 ± 2.85	77.53 ± 1.1	74.2 ± 5.43
ionosphere	84.85±1.6	87.6 ± 2.76	90.52 ± 2.03	89.61 ± 1.75	89.06 ± 2.71
monks-1	100±0	100 ± 0	100 ± 0	93.84 ± 2.93	100 ± 0
mushroom	99.79±0.1	100 ± 0	100 ± 0	99.96 ± 0.04	98.8 ± 0.06
wisconsin	91.18±0.55	94.58 ± 0.68	94.16 ± 0.93	93.99 ± 0.63	95.9 ± 0.56
promoters	52.11±1.96	81.9 ± 4.65	74.72 ± 4.86	78.18 ± 3.62	83.74 ± 3.46
splice	50.72±0.49	90.32 ± 0.74	74.82 ± 2.94	93.88 ± 0.41	89.66 ± 0.78

task because of its global search nature, its intrinsic population-based parallelism and its definition of being a promising "automated invention machine". Besides, the use of a grammar as a way to provide previous available knowledge about the target problem and guide the search seemed appropriated in the case of automatically evolving a rule induction algorithm.

As explained before, in GGPs the grammar is responsible for determining the size of the search space. In the problem of automatically evolving rule induction algorithms, the grammar presented in Table 4.1 represents a search space of approximately 5 billion candidate rule induction algorithms (for details, see Appendix A). As shown in the results of the previous sections, the GGP met our expectations in finding rule induction algorithms competitive with human-designed ones. But is GGP a good way to automatically search for rule induction algorithms? Or could a simpler search method obtain the same kinds of results?

In order to find answers for these questions we implemented a hill climbing (HC) search method. The central idea of this hill climbing search is the following [129]. It randomly generates one solution to a specific problem (which we call current solution), evaluates it and then modifies it (a modification is equivalent to a mutation operation in the GGP). If the new modified solution is better than the current solution, it replaces it. In contrast, if the new solution is worse than the current one, it is discarded and the current solution (which remains unchanged) is modified again. This process is carried out until a maximum number of solutions are evaluated.

Table 5.35: Comparing the predictive accuracies (%) of the GGP-RIs and the GHC-RIs in the meta-test set for experiments with nominal data sets

Data Set	GGP-RIs	GHC-RIs
monks-1	100±0	99.93±0.07
mushroom	99.98±0.01	98.76±0.07
wisconsin	95.17±0.76	93.88±0.57
promoters	77.42±2.54	63.04±0.8
splice	88.59±0.33	70.64±0.3

In order to use this HC method to automatically evolve rule induction algorithms (which is comparable with the proposed GGP), we randomly generate a rule induction algorithm by following the production rules of the grammar, just like in the GGP. The rule induction algorithm is evaluated using the meta-training set (see Section 5.2) and the same fitness used by the GGP. It is then mutated and, if the new rule induction algorithm is better than the current one, it replaces it. Otherwise the unchanged solution (the rule induction algorithm before the mutation) undergoes a new mutation operation. This process is repeated 3000 times (which is equivalent to running the GGP with 100 individuals for 30 generations - parameter values used in all GGP experiments reported in the previous sections). We call this method GHC (Grammar-based Hill Climber).

Therefore, both the GGP and the GHC methods use the same grammar, the same individual representation, the same fitness function and evaluate the same number of candidate rule induction algorithms, making the comparison between the two methods as fair as possible. The two methods differ in that: (a) GGP works with a population of candidate solutions, whereas GHC works with just one candidate solution at a time; (b) As the GGP works with a population, individuals undergo a selection procedure before being modified, while in the GHC an "elitist" strategy is used, and only the best candidate solution is preserved (in other words, the GHC performs a local search, while the GGP performs a global search); and (c) GGP creates new solutions via crossover and mutation whereas GHC uses only "mutation", which is implemented by exactly the same mutation operator used by GGP.

Tables 5.35, 5.36 and 5.37 show the results obtained by the described GHC method when using nominal, numerical and both types of data sets in the meta-training and meta-test sets. The tables show the name of the data sets followed by

Table 5.36: Comparing the predictive accuracies (%) of the GGP-RIs and the GHC-RIs in the meta-test set for experiments with numerical data sets

Data Set	GGP-RIs	GHC-RIs
crx	82.14±0.43	80.29±0.77
ionosphere	86.98±2.54	84.58±2.58
segment	94.75±0.41	87.93±0.38
sonar	72.72±1.95	63.29±1.13
heart-c	75.13±1.52	75.7±1.13

Table 5.37: Comparing the predictive accuracies (%) of the GGP-RIs and the GHC-RIs in the meta-test set for experiments with both nominal and numerical data sets

Data Set	GGP-RIs	GHC-RIs
crx	77.46±3.8	82.66±1.14
segment	95.06±0.26	88.53±1.03
sonar	72.34±1.91	64.92±1.12
heart-c	76.72±1.5	78.75±1.03
ionosphere	87.04±2.2	84.64±1.97
monks-1	99.93±0.07	99.82±0.18
mushroom	99.99±0	99.03±0.07
wisconsin	95.58±0.74	93.56±0.56
promoters	78.98±2.93	60.26±1.96
splice	88.68±0.31	65.2±0.27

the predictive accuracies obtained by the GGP-RIs and the rule induction algorithms produced by the GHC method (denoted GHC-RIs) in 5 different runs (with different random seeds) of each method, respectively. As in the other tables presented in this chapter, all the results were obtained using a 5-fold cross-validation procedure, and the numbers after the symbol "±" are standard deviations.

Results were compared using a two-tailed statistical t-test with significance level 0.01, and cells in dark gray represent significant wins of the GGP-RIs against the GHC-RIs. Across Tables 5.35, 5.36 and 5.37, the GGP-RIs obtained significantly better results than the GHC-RIs in 10 out of 20 cases. There was no case in which the GHC-RIs obtained significantly better results than the GGP-RIs. Hence, in general, we can say that the GGP was much more effective in finding rule induction algorithms than the GHC method.

Analyzing the results obtained in the nominal data sets, we observe that the

Table 5.38: Comparing the GGP-RIs with a GHC-RIs using a Student's t-test

	GGP-RIs		GHC-RIs	
Significance level	0.05	0.01	0.05	0.01
Nominal (GGP-0.5)	2(2)	2(1)	0(9)	0(7)
Numerical (GGP-0.7)	1(0)	1(0)	1(5)	0(3)
Both (GGP-0.7)	3(1)	3(1)	0(14)	0(9)
Total	6(3)	6(2)	1(28)	0(19)

accuracies found by the GHC-RIs were surprisingly low in the data sets *promoters* and *splice*: while the GGP-RIs obtained accuracies of 77.42±2.54 for *promoters* and 88.59±0.33 for *splice*, the GHC-RIs obtained accuracies of 63.04±0.8 and 70.64±0.3 for *promoters* and *splice*, respectively. In any case, the GHC-RIs' accuracies are still comparable with the Unordered-CN2's ones. Studying the actual rule induction algorithms evolved, we notice that 2 out of 5 algorithms produced (in 5 different runs varying the random seed) by the GHC were responsible for the very low accuracy. These two algorithms produced unordered rule sets (rather than ordered rule lists), and post-processed them by removing one or two conditions-at-a-time from the final rule set. Moreover, during the post-processing phase, these algorithms evaluated the new pruned rule sets using accuracy and information-gain in a prune set, respectively. But these strategies were not very successful.

Regarding the experiments involving numerical data sets (Table 5.36), the predictive accuracies in *segment* and *sonar* also decreased significantly, from 94.75±0.41 to 87.93±0.38 and from 72.72±1.95 to 63.29±1.13, respectively.

Looking at the results reported in Table 5.37, once more the accuracies obtained by the GHC-RIs for the data sets *promoters*, *splice*, *segment* and *sonar* are as low as they were in the experiments involving only nominal or numerical data sets in the GHC meta-training set.

In addition to these direct comparisons among the GGP and GHC methods, Table 5.38 shows a comparison of the number of times the GGP generates a rule induction algorithm significantly better (worse) than the baseline rule induction algorithms (both versions of CN2, C4.5Rules and Ripper) against the number of times the GHC search does. The comparisons were made using a Student's t-test considering both significance levels 0.01 and 0.05. Comparing the second and fourth columns, we observe that while the total number of significantly losses of the GGP-RIs to the baseline algorithms is just three, the corresponding total

number for the GHC-RIs is 28 with a 0.05 significance level. The number of significant wins is six for the GGP-RIs and just one for the GHC-RIs.

The results comparing the rule induction algorithms produced by the GGP and GHC methods showed that the GGP-RIs are, overall, clearly superior to the GHC-RIs. In order to try to understand why, Figures 5.1, 5.2 and 5.3 show a comparison of the evolution of candidate rule induction algorithms along the GGP and GHC searches in experiments using nominal, numerical and both nominal and numerical data sets. The plots show fitness values against the number of evaluations produced in one run of each method, where in both runs the same random seed was used to initialize the candidate solution(s) of the corresponding method. Note that while for the GHC we have results for all the evaluations (since the GHC works with one candidate solution at a time), for the GGP we show the fitness of the best individual at the end of each generation (a multiple of 100 evaluations), since GGP is a population-based method.

Observe that in both searches the values of the fitness do not monotonically increase as the number of evaluations increase (as would be expected in a "static" fitness function scenario). This is because, as explained before, in order to avoid over-searching, at each generation of the GGP (which is equivalent to 100 evaluations in the GHC search), for each data set in the meta-training set, the training and validation subsets were changed, which effectively means the fitness landscape changes at every 100 individual evaluations. It is very interesting to notice the effect this design issue has on both algorithms.

In all the 3 graphs, there are always a number of points in which the GHC search has a better solution than the GGP does. Two of these points are illustrated in Figures 5.1 and 5.2 by a dashed vertical line. Notice that the dashed vertical line is in the evaluation number 2100 for the first graph and 2500 for the second graph. Both 2100 and 2500 represent one evaluation before the data sets in the meta-training sets were randomly re-divided into training and validation subsets. In other words, as the GHC improves a solution over the same set of data 100 times, it probably over-fits the rule induction algorithm to the data, which explains the better and better results from evaluations 2001 to 2100 in Figure 5.1, and a sudden and significant drop in evaluation 2101.

It is also worth noting that from the 3000 mutation operations performed by each run of the GHC search, on average only 97.2, 89.6 and 93 were successful in producing better rule induction algorithms than the previous one (i.e. the "parent" algorithm) in 5 runs with the experiments using nominal, numerical and

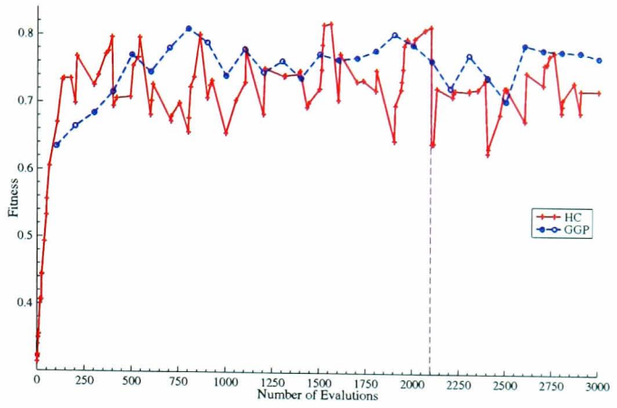

Figure 5.1: Evolution of the GGP *versus* GHC when using nominal data sets in the meta-data sets

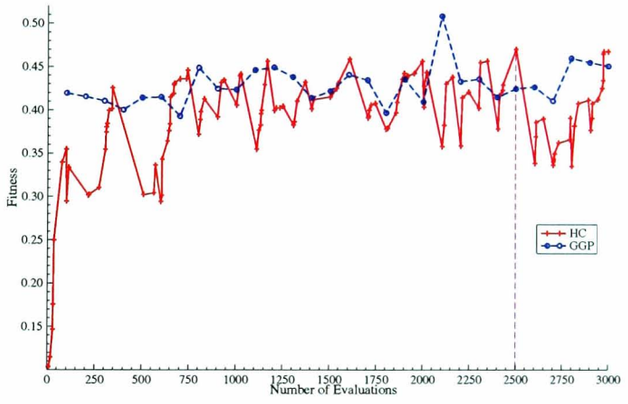

Figure 5.2: Evolution of the GGP *versus* GHC when using numerical data sets in the meta-data sets

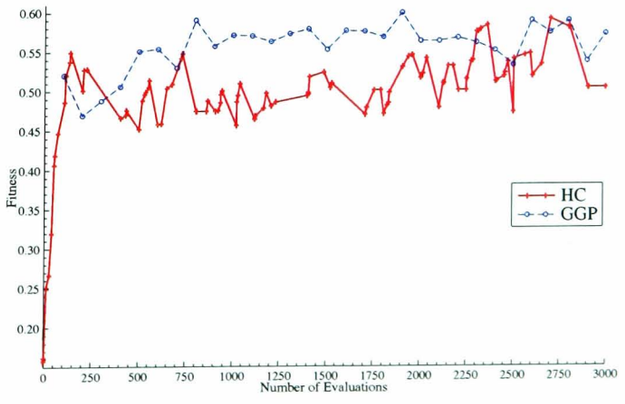

Figure 5.3: Evolution of the GGP *versus* GHC when using nominal and numerical data sets in the meta-data sets

both kinds of data sets, respectively. These numbers tell us that only $\simeq$ 3% of the mutation operations improved the current candidate solution. Unfortunately, we cannot do this same kind of analysis with the GGP. The best way to compare the improvement of the GGP individuals would be to check if the individuals produced after crossover and mutation operations were better than their parents. However, as the data changes at each generation, parents and their offspring are evaluated in different sets of data, and are not directly comparable.

In conclusion, while the GGP has the feature of searching in parallel (evaluating 100 individuals in each generation and breeding them), with the associated advantages of performing a global search, the GHC has the feature of performing a sequential step-wise search, which is simpler but constitutes a form of local search, intuitively with considerably less exploratory power than the GGP. When it comes to search for rule induction algorithms, the GGP clearly produces better results than the GHC method.

5.8 MOGGP: A Multi-Objective Version of the Proposed GGP

One of the motivations to automatically evolve rule induction algorithms is the simplicity of the classification models they generate. This is in contrast with more mathematically sophisticated algorithms such as support vector machines (see Section 2.4), which usually produce a "black-box" model, hardly interpretable by the user. However, in the first experiments performed in this work, the simplicity of the models generated by the evolved rule induction algorithms was not considered when evaluating those rule induction algorithms.

As explained in Section 4.5.3, we developed a more sophisticated version of the GGP that takes into account, in its fitness function, both the predictive accuracy of a rule induction algorithm and the total number of conditions belonging to all rules in the classification model built by that algorithm. These two fitness criteria are simultaneously taken into account using the concepts of multi-objective optimization and Pareto dominance, and so this new version of the GGP is denoted MOGGP.

This section presents the GGP-RIs evolved by the MOGGP, denoted MOGGP-RIs. As in the previous sections, results for experiments mining only nominal, numerical and then both nominal and numerical data sets are presented.

In contrast with the experiments run with the single-objective version of the

Table 5.39: Comparing accuracy rates (%) and rule sizes of the MOGGP-RIs and the SGGP-RIs for the nominal data sets in the meta-test set

Data Set	Pred. Accuracy		Rule Model Size	
	MOGGP-RIs	SGGP-RIs	MOGGP-RIs	SGGP-RIs
monks-1	97.68±1.94	100±0	10.76±0.37	14.2±1.56
mushroom	99.94±0.04	99.98±0.01	16.4±0.44	21±5.0
wisconsin	91.84±1.05	95.17±0.76	8.04±0.45	46.4±7.77
promoters	68.31±1.59	77.42±2.54	3.12±0.12	15.4±2.27
splice	80.32±0.96	88.59±0.33	38.32±1.09	307.6±33.04

GGP, for the MOGGP experiments the crossover and mutation rates were not optimized, and the MOGGP was run with the same sets of parameter values found in the experiments reported in Section 5.2. The justifications for using the same parameter values optimized for the single-objective GGP are two-fold. First, as concluded in Section 5.2, the single-objective GGP was not sensitive to crossover and mutation rate variations. Secondly, in order to compare the results of the single-objective and multi-objective versions of the GGP and draw conclusions concerning the advantages of a multi-objective approach, it is fair to compare their results when they are run with exactly the same parameter values.

Therefore, we kept all the parameter values for the MOGGP exactly the same ones used for the single-objective GGP: population size of 100, 30 generations and tournament size of 2. For the experiments with nominal data sets, the crossover rate is 0.5 and the mutation rate 0.45. For the other two experiments (with numerical data sets and both nominal and numerical data sets), the crossover rate is 0.7 and the mutation rate 0.25. In all experiments the reproduction rate is 0.05.

Table 5.39 presents the results of the predictive accuracy and number of conditions obtained by the rule models generated by the MOGGP-RIs (rule induction algorithms generated by the MOGGP), and compare them with the results obtained by the SGGP-RIs (rule induction algorithms generated by the single-objective version of the GGP). For further analysis, Tables 5.40 and 5.41 present the results of the predictive accuracy and number of conditions present in the rule sets for the MOGGP-RIs compared to the accuracies and number of conditions of the baseline algorithms: Ordered-CN2, Unordered-CN2, Ripper and C4.5Rules. All the results were obtained using a 5-fold cross-validation procedure, and the numbers after the symbol "±" are standard deviations. Results were, as before, compared using a two-tailed statistical t-test with significance level 0.05. Cells

Table 5.40: Comparing accuracy rates (%) of the MOGGP-RIs and the baseline algorithms for the nominal data sets in the meta-test set

Data Set	MOGGP-RIs	OrdCN2	UnordCN2	Ripper	C45Rules
monks-1	97.68±1.94	100 ± 0	100 ± 0	93.84 ± 2.93	100 ± 0
mushroom	99.94±0.04	100 ± 0	100 ± 0	99.96 ± 0.04	98.8 ± 0.06
wisconsin	91.84±1.05	94.58 ± 0.68	94.16 ± 0.93	93.99 ± 0.63	95.9 ± 0.56
promoters	68.31±1.59	81.9 ± 4.65	74.72 ± 4.86	78.18 ± 3.62	83.74 ± 3.46
splice	80.32±0.96	90.32 ± 0.74	74.82 ± 2.94	93.88 ± 0.41	89.66 ± 0.78

Table 5.41: Comparing the number of conditions in the rule sets of the MOGGP-RIs and the baseline algorithms for the nominal data sets in the meta-test set

Data Set	MOGGP-RIs	OrdCN2	UnordCN2	Ripper	C45Rules
monks-1	10.76±0.37	11 ± 0.71	61 ± 0	14 ± 7.51	61 ± 0
mushroom	16.4±0.44	15.6 ± 0.24	26 ± 0	12 ± 0	18.6 ± 2.73
wisconsin	8.04±0.45	32.6 ± 1.36	53.8 ± 2.91	13.2 ± 1.07	19.2 ± 0.86
promoters	3.12±0.12	10.4 ± 0.75	23.6 ± 1.36	6.2 ± 1.11	10.8 ± 1.02
splice	38.32±1.09	256.2 ± 5.08	172.6 ± 9.75	36.4 ± 3.75	119.8 ± 29.68

in dark gray represent statistically significant wins of the MOGGP-RIs against a baseline algorithm (or the SGGP-RIs), while light gray cells represent MOGGP-RIs' statistically significant losses.

As observed in Table 5.39, the MOGGP-RIs obtained significantly worse predictive accuracies than the SGGP-RIs in 3 cases, namely *wisconsin*, *promoters* and *splice*. However, at the same time, the MOGGP-RIs obtained rule models with a much smaller number of conditions than the SGGP-RIs in these same data sets. If we aim to optimize both the predictive accuracy and the simplicity of the rule models simultaneously, neither the MOGGP-RIs nor the SGGP-RIs can be considered better than the other when producing rule models for these data sets. In turn, in the data set *mushroom*, the MOGGP-RIs and SGGP-RIs obtained competitive accuracies, but the MOGGP-RIs produced simpler models.

Considering now Table 5.40, in terms of accuracy the MOGGP-RIs obtained significantly worse results than the baseline rule induction algorithms in 7 cases, involving again the data sets *wisconsin*, *promoters* and *splice*. In just one case (*mushroom* with C4.5Rules) the MOGGP-RIs obtained an accuracy rate better than the baseline algorithms, while in the other 12 cases the MOGGP-RIs' results were as good as the ones obtained by the baseline algorithms. At first glance,

Table 5.42: Comparing the MOGGP-RIs trained with nominal data sets to the baseline algorithms, taking into account both accuracy and number of conditions in the produced rule model, according to the concept of Pareto dominance

MOGGP-RI *vs* Baseline Algorithms		
Neutral	Dominates	Dominated
8	9	3

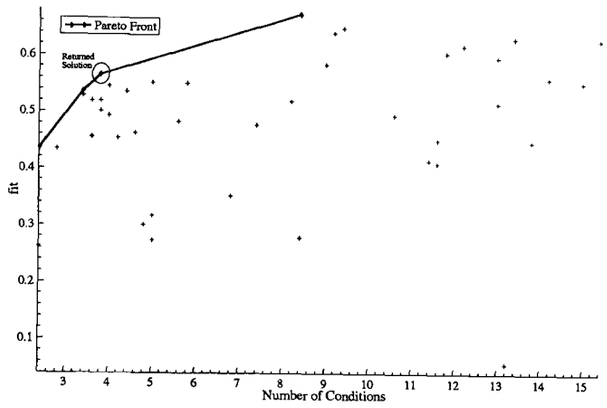

Figure 5.4: Objective values for the last population of individuals evolved by the MOGGP using nominal data sets in the meta-training set

these results could be considered much worse than the ones obtained by the single-objective GGP, which obtained 2 significantly better accuracies than the baseline algorithms and only one significantly worse.

However, when we look at the number of rule conditions present in the classification models produced by the MOGGP-RIs and by the baseline algorithms, we have a very different situation. In 13 out of 20 cases the GGP-RIs found significantly smaller models than the baseline algorithms, and in only one case (*mushroom*) it found a significantly bigger model than Ripper. What is interesting to notice is that, in 5 out of the 7 cases in which the MOGGP-RIs significantly lose to a baseline algorithm in predictive accuracy, they produce a significantly simpler model (by comparison with C4.5Rules in *promoters*, *wisconsin* and *splice* and Ordered-CN2 in *promoters* and *splice*). The same occurs in 8 out of 12 cases in which the MOGGP-RIs produce competitive accuracies with respect to the baseline algorithms: the former's models are much smaller. Therefore, taking into account both the predictive accuracy and the number of conditions in the rule model when evaluating the MOGGP-RIs against the baseline algorithms, the results are summarized in Table 5.42.

Table 5.43: Comparing accuracy rates (%) and rule sizes of the MOGGP-RIs and the SGGP-RIs for the numerical data sets in the meta-test set

Data Set	Pred. Accuracy		Rule Model Size	
	MOGGP-RIs	SGGP-RIs	MOGGP-RIs	SGGP-RIs
crx	82.51±1.41	82.14±0.43	8.96±0.55	87.6±15.56
ionosphere	83.6±2.54	86.98±2.54	5.6±0.27	23.6±3.01
segment	91.52±0.47	94.75±0.41	29.44±1.27	77.4±10.94
sonar	66.98±1.73	72.72±1.95	4.04±0.37	20.2±2.62
heart-c	75.38±1.4	75.13±1.52	4.72±0.53	43±5.74

Table 5.42 uses a different terminology when compared to the tables showed so far. This is because when evaluating the MOGGP regarding accuracy and number of conditions in the rule model (two objectives), we based our analysis of results on the Pareto dominance concept – adapted to consider statistically significant differences. As described before, this adapted Pareto dominance concept states that a solution S_1 dominates a solution S_2 if two conditions are satisfied. First, if every objective value of S_1 is not significantly worse than the corresponding objective value in S_2. Secondly, if at least one of the objective values of S_1 is significantly better than the corresponding objective value of S_2.

Hence, Table 5.42 presents the number of classification models produced by the MOGGP-RIs which are neither significantly better nor significantly worse than the classification models produced by the baseline algorithms (*Neutral* column), the number of models produced by the baseline algorithms which the MOGGP dominates (*Dominate* column), and finally the number of models produced by the MOGGP which are dominated by a model produced by a baseline algorithm (*Dominated* column). This table was built by applying the significance-adapted Pareto dominance concept to the results presented in Tables 5.40 and 5.41.

To illustrate the logic behind it, let us consider the cases of *mushroom* and *wisconsin* with Ripper in Tables 5.40 and 5.41. In both cases the accuracies of the MOGGP-RIs and Ripper are competitive. However, in *mushroom* the model generated by Ripper has a significantly smaller number of rule conditions than the model generated by the MOGGP-RIs, and so we say that Ripper dominates the MOGGP-RIs. The opposite situation occurs for *wisconsin*, where the models generated by MOGGP-RIs have a significantly smaller number of conditions than the models generated by Ripper, and we say that the MOGGP-RIs dominate Ripper. At last, if we have some cases in which the MOGGP-RIs are significantly better in one objective and a baseline algorithm is significantly better in the other

Table 5.44: Comparing predictive accuracy rates (%) of the MOGGP-RIs and the baseline algorithms for the numerical data sets in the meta-test set

Data Set	MOGGP-RIs	OrdCN2	UnordCN2	Ripper	C45Rules
crx	82.51±1.41	80.16 ± 1.27	80.6 ± 0.93	84.37 ± 1.21	84.82 ± 1.53
ionosphere	83.6±2.54	87.6 ± 2.76	90.52 ± 2.03	89.61 ± 1.75	89.06 ± 2.71
segment	91.52±0.47	95.38 ± 0.28	85.26 ± 0.87	95.44 ± 0.32	88.16 ± 7.72
sonar	66.98±1.73	70.42 ± 2.66	72.42 ± 1.4	72.88 ± 4.83	72.4 ± 2.68
heart-c	75.38±1.4	77.9 ± 1.96	77.54 ± 2.85	77.53 ± 1.1	74.2 ± 5.43

Table 5.45: Comparing the number of conditions in the rule sets of the MOGGP-RIs and the baseline algorithms for the numerical data sets in the meta-test set

Data Set	MOGGP-RIs	OrdCN2	UnordCN2	Ripper	C45Rules
crx	8.96±0.55	101.4 ± 3.46	101.6 ± 2.38	4.8 ± 1.62	34 ± 1.38
ionosphere	5.6±0.27	20.2 ± 1.53	37 ± 1.84	6.2 ± 1.56	10.2 ± 4.34
segment	29.44±1.27	73.8 ± 1.74	102.8 ± 2.15	47.6 ± 2.79	96.8 ± 12.71
sonar	4.04±0.37	19.4 ± 0.87	50.2 ± 3.02	5.4 ± 0.81	14.6 ± 4.3
heart-c	4.72±0.53	37.2 ± 1.24	70 ± 3.54	6 ± 0.55	22 ± 5.63

objective (such as when comparing the MOGGP-RIs with Ordered-CN2 in the *splice* data set), we say that two types of models have a neutral relationship. The two types of algorithms being compared are also considered to have a neutral relationship if there is no statistically significant difference in the values of each of the two objectives between the two types of algorithms.

Figure 5.4 shows a graph representing the objective values for the last population of individuals evolved by a run of the MOGGP. The x-axis shows the average number of rule conditions in the models built from data sets of the meta-training set, and the y-axis shows the value of *fit* as defined in Eq. (4.4). Each point in the graph represents an individual (a rule induction algorithm). The greater the predictive accuracy (and consequently the value of *fit*) and the smaller the number of rule conditions, the better. The graph also shows the Pareto front found by the MOGGP, which is formed by the set of individuals which are not dominated by any other individual in the population of the last generation. The circle indicates the individual in the Pareto front which was returned as the best single solution for the problem (see Section 4.5.3 for details about how it was chosen). The rule induction algorithm represented by this individual was the one evaluated in the meta-test set.

Table 5.46: Comparing the MOGGP-RIs trained with numerical data sets to the baseline algorithms, taking into account both accuracy and number of the conditions in the produced rule model, according to the concept of Pareto dominance

MOGGP-RI *vs* Baseline Algorithms		
Neutral	Dominates	Dominated
7	12	1

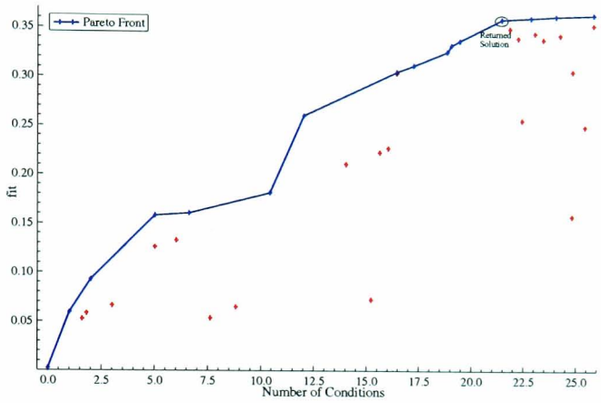

Figure 5.5: Objective values for the last population of individuals evolved by the MOGGP using numerical data sets in the meta-training set

Table 5.43 shows the predictive accuracy and the number of conditions obtained by the MOGGP-RIs and SGGP-RIs. Tables 5.44 and 5.45 show the results of the accuracy and number of conditions in the rule sets for the MOGGP-RIs (rule induction algorithms generated by the MOGGP) compared to the SGGP-RIs and the baseline algorithms when trained with numerical data sets. In Table 5.43, we observe that the MOGGP-RIs obtained significantly worse accuracies than the SGGP-RIs in two data sets, namely *segment* and *sonar*. In contrast, the models generated by the MOGGP-RIs were simpler than the models generated by the SGGP-RIs in all five data sets. Hence, we can conclude that for the data sets *crx*, *ionosphere* and *heart-c*, the MOGGP-RIs obtained simples models than the SGGP-RIs with competitive accuracies.

Regarding the comparisons of the MOGGP-RIs with the baseline algorithms, Table 5.44 has 3 cases in which the MOGGP-RIs obtained significantly worse accuracies than the baseline algorithms: Ordered-CN2 and Ripper in *segment* and Unordered-CN2 in *sonar*. Compared to the results of the single-objective GGP, the MOGGP-RIs maintain a significantly better accuracy over the *segment* classified by Unordered-CN2, but these 3 cases where MOGGP-RIs loses are new

Table 5.47: Comparing accuracy rates (%) and rule sizes of the MOGGP-RIs and the SGGP-RIs for both nominal and numerical data sets in the meta-test set

Data Set	Pred. Accuracy		Rule Model Size	
	MOGGP-RIs	SGGP-RIs	MOGGP-RIs	SGGP-RIs
crx	83.33±1.26	77.46±3.8	13.52±0.72	99.4±5.99
segment	92±0.67	95.06±0.26	25.64±1.22	83.2±6.13
sonar	68.04±1.74	72.34±1.91	4.6±0.75	20.2±1.32
heart-c	76.46±1.82	76.72±1.5	7.2±0.9	50.6±2.93
ionosphere	85.48±1.63	87.04±2.2	7.88±0.62	24.2±1.53
monks-1	99.78±0.22	99.93±0.07	11.64±0.39	13±2.04
mushroom	99.66±0.22	99.99±0	15.16±0.38	15.2±0.58
wisconsin	92.1±0.71	95.58±0.74	9.68±0.57	48±10.62
promoters	71.84±5.24	78.98±2.93	3.96±0.38	14.6±2.27
splice	87.68±0.5	88.68±0.31	42.52±2.3	271.8±12.02

when considering experiments with 5 data sets in the meta-training set. Recall that in experiments executed when training the GGP with 6 and 7 numerical data sets (see Tables 5.24 and 5.25), the accuracies of the GGP-RIs in *segment* were significantly worse than the accuracies of Ordered-CN2 and Ripper. As explained before, *segment* is a special data set, with plenty of class noise.

Analyzing the number of conditions in the rule sets, though, the MOGGP-RIs obtained significantly better results than the baseline algorithms in 15 out of the 20 cases, including *segment* with Ordered-CN2 and Ripper and *sonar* with Unordered-CN2, and only one significantly worse result in *crx* with Ripper. Summarizing the results using the Pareto dominance concept, we now have that the MOGGP-RIs dominate 12 of the baseline algorithms, and are dominated by only one of them, as shown in Table 5.46. As before, Figure 5.5 shows the objective values for individuals of the last population of a run of the MOGGP trained with numerical attributes, the Pareto front found by the MOGGP and the individual returned as the best rule induction algorithm evolved (marked by a circle).

At last, we present the results when evolving rule induction algorithms using MOGGP trained with both nominal and numerical data sets. Table 5.47 shows the predictive accuracy and the number of conditions obtained by the MOGGP-RIs and SGGP-RIs. As observed in Table 5.47, the MOGGP-RIs obtained accuracies statistically worse than the ones obtained by the SGGP-RIs in two data sets: *segment* and *wisconsin*. In the other 8 data sets, both the accuracies obtained by the MOGGP-RIs and the SGGP-RIs were statistically the same. In contrast, the

Table 5.48: Comparing predictive accuracy rates (%) of the MOGGP-RI and the baseline algorithms for both nominal and numerical data sets in the meta-test set

Data Set	MOGGP-RIs	OrdCN2	UnordCN2	Ripper	C45Rules
crx	83.33±1.26	80.16 ± 1.27	80.6 ± 0.93	84.37 ± 1.21	84.82 ± 1.53
segment	92±0.67	95.38 ± 0.28	85.26 ± 0.87	95.44 ± 0.32	88.16 ± 7.72
sonar	68.04±1.74	70.42 ± 2.66	72.42 ± 1.4	72.88 ± 4.83	72.4 ± 2.68
heart-c	76.46±1.82	77.9 ± 1.96	77.54 ± 2.85	77.53 ± 1.1	74.2 ± 5.43
ionosphere	85.48±1.63	87.6 ± 2.76	90.52 ± 2.03	89.61 ± 1.75	89.06 ± 2.71
monks-1	99.78±0.22	100 ± 0	100 ± 0	93.84 ± 2.93	100 ± 0
mushroom	99.66±0.22	100 ± 0	100 ± 0	99.96 ± 0.04	98.8 ± 0.06
wisconsin	92.1±0.71	94.58 ± 0.68	94.16 ± 0.93	93.99 ± 0.63	95.9 ± 0.56
promoters	71.84±5.24	81.9 ± 4.65	74.72 ± 4.86	78.18 ± 3.62	83.74 ± 3.46
splice	87.68±0.5	90.32 ± 0.74	74.82 ± 2.94	93.88 ± 0.41	89.66 ± 0.78

Table 5.49: Comparing the number of the conditions presented in the rule sets of the MOGGP-RI and the baseline algorithms for both nominal and numerical data sets in the meta-test set

Data Set	MOGGP-RIs	OrdCN2	UnordCN2	Ripper	C45Rules
crx	13.52±0.72	101.4 ± 3.46	101.6 ± 2.38	4.8 ± 1.62	34 ± 1.38
segment	25.64±1.22	73.8 ± 1.74	102.8 ± 2.15	47.6 ± 2.79	96.8 ± 12.71
sonar	4.6±0.75	19.4 ± 0.87	50.2 ± 3.02	5.4 ± 0.81	14.6 ± 4.3
heart-c	7.2±0.9	37.2 ± 1.24	70 ± 3.54	6 ± 0.55	22 ± 5.63
ionosphere	7.88±0.62	20.2 ± 1.53	37 ± 1.84	6.2 ± 1.56	10.2 ± 4.34
monks-1	11.64±0.39	11 ± 0.71	61 ± 0	14 ± 7.51	61 ± 0
mushroom	15.16±0.38	15.6 ± 0.24	26 ± 0	12 ± 0	18.6 ± 2.73
wisconsin	9.68±0.57	32.6 ± 1.36	53.8 ± 2.91	13.2 ± 1.07	19.2 ± 0.86
promoters	3.96±0.38	10.4 ± 0.75	23.6 ± 1.36	6.2 ± 1.11	10.8 ± 1.02
splice	42.52±2.3	256.2 ± 5.08	172.6 ± 9.75	36.4 ± 3.75	119.8 ± 29.68

rule models produced by the MOGGP-RIs are significantly smaller than the ones produced by the SGGP-RIs in 8 out of 10 cases, while in the other two cases the results are statistically the same.

Tables 5.48 and 5.49 present the predictive accuracies and number of conditions per rule model obtained by the MOGGP-RIs and the four baseline algorithms. Analyzing Table 5.48, we note that the MOGGP-RIs have significantly better accuracies than the baseline algorithms in 3 cases and significantly worse accuracies in 6 cases. Compared to the result for the single-objective GGP-RIs (show in Table 5.13), the 3 significant wins over the baseline algorithms remained the same, but the number of significant losses increased from 1 to 6, and occurred in the

Table 5.50: Comparing the MOGGP-RIs trained with both nominal and numerical data sets to the baseline algorithms, taking into account both accuracy and number of the conditions in the produced rule model according to the concept of Pareto dominance

MOGGP-RI *vs* Baseline Algorithms		
Neutral	Dominates	Dominated
13	24	3

data sets *segment*, *splice* and *wisconsin*.

Regarding the number of conditions in the generated rule models, as can be observed in Table 5.49, the MOGGP-RIs produced significantly smaller models in 28 out of the 40 cases. In 5 out of the 6 cases in which the MOGGP-RIs generated significantly smaller accuracies, they also created smaller models. As summarized in Table 5.50, an analysis of Pareto dominance taking into account both the accuracy and the size of the models produced shows that in 24 out of the 40 cases the MOGGP-RIs' models dominate the baseline algorithms' models, and the former are dominated by the latter in only 3 cases.

In summary, as discussed in this section, a more comprehensive comparison of the rule induction algorithms produced by the MOGGP with the baseline rule induction algorithms should take into consideration not just the accuracies generated by them, but also the size of the rule models created. Comprehensibility of the classification model is a key point in rule induction algorithms, and an analysis of the models produced by the MOGGP-RIs against the models produced by the baseline algorithms showed that, for example, for all the data sets in the meta-test sets of the three sets of experiments performed, the MOGGP-RIs always produced rule models smaller than the Unordered-CN2 ones. Besides, in 80% of the cases, the MOGGP-RIs' models are also smaller than the Ordered-CN2's models.

5.8.1 An Insight About the MOGGP-RIs

In the previous section, we showed that an analysis of both the predictive accuracy and the size of the rule models produced by the MOGGP-RIs reveals that, in general, the MOGGP-RIs' models are much more compact than the models produced by the baseline algorithms. However, the most interesting fact to notice about the MOGGP-RIs is the impact that the inclusion of a measure of rule model size in the fitness function had in their design.

For instance, 14 out of the 15 algorithms produced by the MOGGP (5 runs

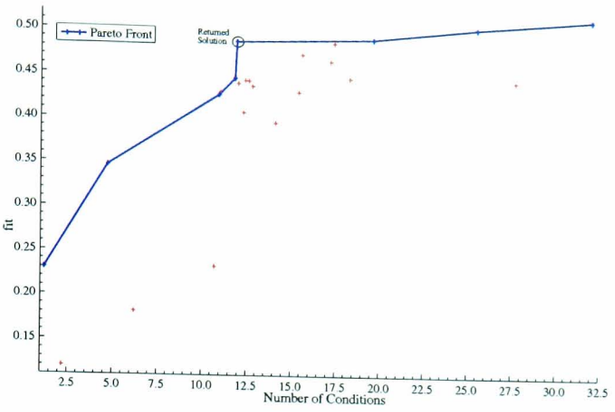

Figure 5.6: Objective values for the last population of individuals evolved by the MOGGP using both nominal and numerical data sets in the meta-training set

with different random seeds in 3 experiment settings considering different meta-training sets) use both a pre- and a post-pruning method. In contrast, none of the algorithms produced by the single-objective GGP used post-pruning, and only one out of the 15 algorithms pre-pruned rules by changing the final produced rule. Of course other forms of pre-pruning were used by these algorithms, such as considering as candidate rules only statistically significant rules, or rules with an accuracy greater than a predefined threshold.

Alg. 5.6 shows an example of a MOGGP-RI produced when the MOGGP was trained with only nominal data sets. The first feature to notice in this algorithm is that it works with three different sets of data. It first divides the training set into build and post-prune sets, and subsequently divides the build set into grow and pre-prune sets. We recognize that this data division might be problematic if few examples are available from the training set. However, this algorithm obtained accuracies statistically competitive with the ones provided by the well-known human-designed algorithms, together with much simpler rule models.

Having three sets of data, Alg. 5.6 works in three phases. First, it greedily builds rules and evaluates them using the Laplace estimation criterion in the grow set. The two best candidate rules are selected to undergo further refinements, and once the best rule is found, it is pre-pruned. In this second phase, the algorithm removes, one at-a-time, all the conditions present in the best rule. The new generated rules are again evaluated using the Laplace estimation criterion in the pre-prune set.

Rules are produced until 99% of the examples in the original grow set are not covered. Once the rule list is complete, the third phase starts. The rule list is

Algorithm 5.6: Example of a rule list algorithm created by the MOGGP using nominal data sets

```
Divide the training data in Build and PostPrune
RuleList = ∅
repeat
    Divide the Build data in Grow and PrePrune
    bestRule = an empty rule
    candidateRules = ∅
    candidateRules = candidateRules ∪ bestRule
    while candidateRules ≠ ∅ do
        for each candidateRule CR do
            newCandidateRules = ∅
            Add 1 condition-at-a-time to CR
            Evaluate CR using the Laplace estimation in Grow
            newCandidateRules = newCandidateRules ∪ CR
        candidateRules = 2 best rules selected from newCandidateRules
    for each condition c in bestRule do
        bestRule' = Rule produced by removing condition c from bestRule
        Evaluate bestRule' into PrePrune using Laplace estimation
        if bestRule' better than bestRule then
            bestRule = bestRule'
    RuleList = RuleList ∪ bestRule
until at least 99% of the examples in Grow are covered
notImproving = false
repeat
    RuleList' = Remove last rule from RuleList
    if accuracy(RuleList') ≥ accuracy(RuleList) in PostPrune then
        RuleList = RuleList'
    else notImproving = true
until notImproving
```

post-pruned by removing, from the last to the first inserted rule, one rule at-a-time. Rules are removed from the rule list while the accuracy of the system in the post-prune data set is not reduced.

The dynamics of Alg. 5.6 can be compared to Ripper's. Both algorithms produce, prune and "optimize" the entire rule list, with the following differences. Ripper works with rules ordered per class, produces only rules which do not cover any negative examples and evaluates them using information gain, and uses the MDL (minimum description length) criterion when optimizing the set of discovered rules. Alg. 5.6 produces rule lists, evaluates rules using the Laplace estimation and "optimizes" them by removing one condition-at-a-time. We do not know any algorithm in the literature which works in this way, so Alg. 5.6 is considered a

Algorithm 5.7: Example of a rule list algorithm created by the MOGGP using mostly numerical data sets

```
Divide the training data in Build and PostPrune
RuleList = ∅
repeat
    Divide the Build data in Grow and PrePrune
    bestRule = an empty rule
    candidateRules = ∅
    candidateRules = candidateRules ∪ bestRule
    while candidateRules ≠ ∅ do
        for each candidateRule CR do
            newCandidateRules = ∅
            Add 1 condition-at-a-time to CR
            Evaluate CR using the Laplace estimation in Grow
            if CR is significant with 99% significance level then
                newCandidateRules = newCandidateRules ∪ CR
        candidateRules = best rule selected from newCandidateRules
    notImproving = false
    repeat
        bestRule' = Rule produced by removing the last condition from
                    bestRule
        Evaluate bestRule' into PrePrune using Laplace estimation
        if bestRule' better than bestRule then
            bestRule = bestRule'
        else notImproving = true
    until notImproving
    RuleList = RuleList ∪ bestRule
until there is a maximum of 20 examples in Grow not covered
notImproving = false
repeat
    RuleList' = Remove 1 condition-at-a-time from the last rule in RuleList
    if accuracy(RuleList') ≥ accuracy(RuleList) in PostPrune then
        RuleList = RuleList'
    else notImproving = true
until notImproving
```

novel algorithm.

Algorithm 5.7 shows another example of a MOGGP-RI. It is similar to the algorithm presented in Algorithm 5.6, with the following differences. First, it tests the statistical significance of the rules before considering them as possible candidate rules, and only selects one candidate rule to undergo further refinements. Second, it pre-prunes a rule by removing only the last condition added to it, instead of removing each of the possible conditions-at-a-time, as Algorithm 5.6

does. At last, the criterion used to stop producing rules takes into account the number of examples left in the grow data subset; more precisely, it stops producing rules when the number of examples in the grow set is reduced to 20 or less.

These two algorithms reflect how the addition of a measure of rule set complexity to the fitness of the GGP changed the GGP-RIs evolved in order to find a good balance of the accuracy/complexity trade-off. They also show that the MOGGP is flexible enough to produce very different kinds of algorithms.

5.9 A Note on the GGP System's Execution Time

Formal analyses of the computational time and time complexity of evolutionary algorithms are not very often performed. Although theoretical studies [78] suggest the use of Markov chains models in order to perform these analyses, this is not a simple task.

Performing this kind of theoretical analysis is out of the scope of this thesis. Instead, we discuss the factors which tend to define the best and worst case scenarios in a run of the proposed GGP system.

The most time consuming operation in the GGP system is the evaluation of the individuals. As each individual represents a complete rule induction algorithm, the GGP overall runtime depends on the rule induction algorithms' overall runtime.

More precisely, the runtime of the fitness function of the GGP (by far the most time consuming part of the GGP) depends on three main factors:

- The number of data sets in the meta-training set.
- The number of attributes (and attributes values, in the case of nominal attributes) and examples in each data set in the meta-training set.
- The rule induction components (incorporated in the grammar) used by the GGP to produce a complete rule induction algorithm.

This third factor, in special, considerably changes the rule induction algorithms' runtimes. This is because, when analyzing the symbols of the grammar, we notice that a few of them perform operations with a time complexity higher than linear. *Add2*, for instance, considers all the combinations of attribute/value pairs two by two, so its time complexity is quadratic with respect to the number

Table 5.51: GGP runtime for different experiments' configurations when training the GGP with many data sets in the meta-training set

Meta-training set		Time (hr:min)	
Att. Type	# Data sets	Best	Worst
Nominal	3	2:04	3:47
	4	2:21	4:48
	5	3:26	5:43
	6	5:03	6:49
	7	7:59	12:45
Numerical	3	10:36	34:11
	4	15:22	26:06
	5	21:50	64:21
	6	15:16	28:09
	7	13:41	61:06
Nominal and Numerical	6	33:21	48:29
	8	25:39	32:48
	10	18:29	34:02
	12	33:12	53:30
	14	40:31	64:21

of attributes. As another example, *RemoveCondRule* post-prunes rule models by removing one/two conditions-at-a-time from each rule in the rule model.

Since it is so difficult to estimate a best/worst runtime for the GGP, we report here the best/worst runtimes empirically obtained for each of the experiments with the GGP reported in this chapter. All the experiments were performed on pentium 4 duo processor machines with 1GB RAM and running Linux.

Table 5.51 shows the results of the best and worst case scenarios when running the proposed GGP with different numbers of data sets in the meta-training set. The first column presents the type of data sets (according to the data type of the data set attributes) used during the experiments, followed by the number of data sets in the meta-training set and the best and worst run times, reported in the format hours:minutes.

It is important to remark that the execution time of the rule induction algorithms can still be improved, and their code is not optimized. As the GGP considers many combinations of non-terminals/terminals, it is not simple to write the code for all the grammar symbols in a way that all the combinations run in the best possible time.

5.10 Summary

This chapter presented the results obtained by the GGP system proposed in Chapter 4 when evolving robust rule induction algorithms. We reported the results of preliminary experiments executed to optimize the crossover and mutation parameters of the GGP, and then presented the results obtained by the GGP system in three phases.

In the first phase, we considered GGP-RIs evolved when using only data sets with nominal attributes in the meta-training set. In the second phase, we considered data sets with at least one numerical attribute in the GGP's meta-training set. In the third phase, we merged the data sets used in the previous two phases, and executed experiments with a total of 20 data sets. At last, experiments analyzing the impact of the number of data sets used in the meta-training set on the predictive accuracy of the evolved GGP-RIs were also reported.

For all the experiments reported in this chapter, the predictive accuracies obtained by the GGP-RIs were compared to the predictive accuracies of 4 well-known human-designed rule induction algorithms, namely Ordered-CN2, Unordered-CN2, Ripper and C4.5Rules. In general, the results showed that the GGP-RIs were competitive with these baseline methods. We also showed some of the evolved GGP-RIs, and highlighted their innovative procedures.

In order to evaluate the effectiveness of the GGP system in evolving GGP-RIs, we compared it with a grammar-based hill climber (GHC) system. The comparisons showed that, in many cases, the GGP-RIs obtained by the GGP are superior to the GHC-RIs obtained by the GHC, and the reverse situation is rarely observed. Hence, we can clearly conclude that the GGP is a considerably more effective method for evolving rule induction algorithms than the GHC.

Experiments with a multi-objective version of the GGP (MOGGP) were also performed, and showed that besides being competitive with the human-designed rule induction algorithms, many of the MOGGP-RIs also produced a more compact classification model.

The next chapter presents the computational results obtained when using the GGP in the second framework described in Section 5.1: to automatically evolve rule induction algorithms tailored to one specific data set.

Chapter 6

Evaluating the Proposed System for Evolving Algorithms Tailored to One Data Set

6.1 Introduction

As explained before, the grammar-based genetic programming system proposed in this thesis can be used into two different frameworks: to evolve robust rule induction algorithms, as reported in Chapter 5, or to evolve data set tailored rule induction algorithms. This latter approach is the one studied in this chapter.

There are two main differences in the set up of the experiments in this chapter when compared to the experiments in Chapter 5. First, while in Chapter 5 a set of data sets was used in the GGP's meta-training set, in this chapter only one data set is used in the GGP meta-training set. Secondly, while in Chapter 5 the meta-test set was composed by data sets coming from different application domains than the data sets present in the meta-training set, in this chapter the meta-test set works with a data set coming from the same application domain than the data set in the meta-training set. For example, if we want to find a rule induction algorithm tailored to the data set *promoters*, both the data subsets in the meta-training and meta-test sets will use data subsets from the data set *promoters*. Note that, in this context, the use of the terms meta-training and meta-test sets is less justifiable, once we have only one data set in the meta-training and one data set in the meta-test set. However, as we are performing some kind of meta-learning anyway, we keep this terminology.

For all the experiments run in this chapter, each data set was divided in three subsets: a training, a validation and a test set. The training and validation sets were inserted into the GGP's meta-training set, and used to evaluate the candidate rule induction algorithms found by the GGP. The meta-test set was not used during the evolution of the GGP, being reserved only for evaluating the predictive accuracy of the evolved rule induction algorithm. In order to evaluate this predictive accuracy, after the GGP was run, the training and validation sets used in the meta-training set during evolution were merged to create a new training set. The evolved rule induction algorithm used this new training set to create a rule model, which was then evaluated on the unseen test set.

For this set of experiments, optimizing all the GGP parameters for each specific data set was not a practical option, since it would take a very long time. Hence, we used the same parameters used in the experiments presented in Chapter 5: population size of 100, evolved in 30 generations and tournament size 2. Regarding the crossover and mutation probabilities, we used the value 0.7 for crossover and 0.25 for mutation, being the reproduction probability 0.05. These values were chosen because they were considered the best ones in the sets of experiments involving numerical and both nominal and numerical data sets (see Section 5.2).

For each data set, the GGP was run 25 times. As before, we had 5 different random seeds, but in order to obtain more statistical support for the results, for each random seed, we evolved 5 GGP-RIs. In each of these 5 runs, we varied the data in the meta-training and meta-test sets. In order to do that, the entire data set was divided in 5 partitions and, as in a conventional 5-fold cross validation procedure, each time three data partitions were used for training, one for validation and one for test.

Regarding the data sets used in the experiments, they were divided in two groups. First, we run experiments for 20 data sets used in the experiments in Chapter 5, as reported in Section 6.2. Second, we run experiments for 5 bioinformatics data sets, as reported in Section 6.3.

6.2 Experiments with UCI Data Sets

Initially, we run experiments using the GGP to evolve rule induction algorithms tailored to a specific UCI [104] data set. Table 6.1 shows the predictive accuracies obtained by the GGP-RIs followed by the values of the predictive accuracies of the baseline methods, namely Ordered-CN2 and Unordered-CN2, Ripper and

C4.5Rules. Results were compared using a two-tailed Student's t-test with significance level 0.05. Cells in dark gray represent statistically significant wins of the GGP-RIs over the respective baseline method, while cells in light gray represent statistically significant wins of the respective baseline method against the GGP-RIs. Note that, for the data set *splice*, an attribute selection method was applied before given the data set to the GGP. The reason for that was the computational time required to build rule induction algorithms for this data set. *splice* has 63 attributes and 3190 examples, and training the GGP with it was extremely slow.

The attribute selection method applied to *splice* is not optimal, as feature selection is out of the scope of this thesis. It followed the filter approach, and was based on the attributes' gain ratios [145]. In this approach, the attributes were ranked according to their gain ratio, and then removed, 10 by 10, from the worst to the best (according to their rank), from the original data set. Every time 10 attributes were removed from the training set, the new data set was then mined by the four baseline methods in a 5-fold cross-validation procedure (i.e, both versions of CN2, Ripper and C4.5Rules). This process of removing 10 attributes from the training set was repeated until the predictive accuracy of at least one of baseline methods dropped. In this case, the last 10 removed attributes were added again to the training set and then removed, one by one, from the worst to the best, while the predictive accuracy of the classifiers did not decrease. As a result of the attribute selection method, 14 out of 63 attributes were used.

Results in Table 6.1 show that the GGP-RIs obtain predictive accuracies significantly better than the baseline methods in 10 out of 80 cases, and accuracies significantly worse than the baseline methods in 5 cases. In the remaining 65 cases, the GGP-RIs' accuracies are considered statistically competitive with the ones generated by the baseline human-designed rule induction algorithms.

Table 6.1 presents 6 data sets that have not appeared before in the set of meta-test sets, namely *balance-scale*, *monks-2*, *vehicle*, *vowel*, *glass* and *lymphs*. 7 out of the 10 cases where the GGP-RIs obtain significantly better accuracies than a baseline method occur in these data sets. The other 3 cases reflect the results of previous experiments with a set of data sets in the meta-training set, and appear in *mushroom*, *segment* and *splice*. In these cases the accuracies of C4.5Rules and Unordered-CN2 are too low when compared to the other methods. The five significant losses of the GGP-RIs occur in the data sets *hepatitis*, *lymph*, *vehicle*, *vowel* and *splice*.

Table 6.1: Predictive accuracy rates (%) for GGP-RIs tailored to a specific data set in experiments using data sub-sets of the same application domain the meta-training and meta-test sets

Data Set	GGP-RIs	OrdCN2	UnordCN2	Ripper	C45Rules
balance-scale	79.77±1.11	81.62 ± 1.53	77.1 ± 1.18	74.04 ± 1.27	76 ± 1.58
crx	82.22±1.51	80.16 ± 1.27	80.6 ± 0.93	84.37 ± 1.21	84.82 ± 1.53
glass	65.36±1.89	68.44 ± 4.58	69.42 ± 2.27	66.13 ± 3.32	67.72 ± 4.23
heart-c	78.67±1.37	77.9 ± 1.96	77.54 ± 2.85	77.53 ± 1.1	74.2 ± 5.43
hepatitis	80.02±1.07	81.94 ± 5.02	83.34 ± 1.83	86.03 ± 1.14	83.36 ± 0.9
ionosphere	85.65±2.85	87.6 ± 2.76	90.52 ± 2.03	89.61 ± 1.75	89.06 ± 2.71
lymph	75.85±2.04	82.44 ± 1.7	80.36 ± 4.06	79.03 ± 4.62	81.42 ± 4.05
monks-1	100±0	100 ± 0	100 ± 0	93.84 ± 2.93	100 ± 0
monks-2	89.67±1.22	87.26 ± 1.09	76.5 ± 0.83	64.1 ± 0.8	73.78 ± 2.25
monks-3	98.38±0.6	97.46 ± 0.74	99.1 ± 0.4	98.54 ± 0.46	94 ± 4.89
mushroom	100±0	100 ± 0	100 ± 0	99.96 ± 0.04	98.8 ± 0.06
pima	73.57±0.53	69.34 ± 2.13	74.6 ± 0.38	73.91 ± 1.65	71.04 ± 1.67
promoters	76.07±3.36	81.9 ± 4.65	74.72 ± 4.86	78.18 ± 3.62	83.74 ± 3.46
segment	95.55±0.25	95.38 ± 0.28	85.26 ± 0.87	95.44 ± 0.32	88.16 ± 7.72
splice	89.53±0.65	90.32 ± 0.74	74.82 ± 2.94	93.88 ± 0.41	89.66 ± 0.78
sonar	70.9±2.27	70.42 ± 2.66	72.42 ± 1.4	72.88 ± 4.83	72.4 ± 2.68
vehicle	68.34±0.95	70.16 ± 1.68	59.68 ± 2.01	66.46 ± 1.94	71.94 ± 1.29
vowel	73.66±1.11	76.64 ± 0.93	62.02 ± 1.63	68.93 ± 2.64	58.68 ± 6.24
wisconsin	94.27±1.1	94.58 ± 0.68	94.16 ± 0.93	93.99 ± 0.63	95.9 ± 0.56
zoo	91.85±1.87	92.64 ± 1.33	92.52 ± 2.21	89.47 ± 1.66	92.56 ± 1.45

These results show that apparently the GGP was not able to find Ordered-CN2 – the algorithm with best performance among the four manually-designed algorithms in Table 6.1 for the data set *lymph*, although Ordered-CN2 is included in the GGP search space. However, when analyzing the 25 GGP-RIs produced for *lymph*, we observe that six of them were actually instances of Ordered-CN2.

As explained before, apart from measuring the predictive accuracy obtained by the GGP-RIs, it is also important to know if the GGP is capable of producing innovative rule induction algorithms, which are not simply a reproduction of the human-designed ones. The experiments showed in Table 6.1 generated nothing less than 500 algorithms (20 data sets × 25 algorithms). The first thing easily noticed is that the algorithms produced with single data sets in the meta-training set are much more original (different from manually-designed algorithms) than the ones produced when using a set of data sets in the meta-training set. We can say that approximately 25% of the algorithms produced were instances of

Ordered-CN2, or just changed one or two of its main components. Nevertheless, there were plenty of algorithms using a more creative way of finding rules.

From that, we can conclude that when evolving rule induction algorithms for a set of data sets the GGP is more cautious, and gives preference to simpler algorithms, which are more likely to perform well in a set of very different data sets. However, when tailoring the algorithm to the data, more "specialized" algorithms were generated. Another point to be noticed is that, for the same random seed, the 5 evolved algorithms generated with different sets of data during a cross-validation process usually follow the same broad strategy. For instance, in the data set *balance-scale*, 4 out of the 5 algorithms use a bottom-up approach instead of a top-down one. In addition, 3 out of these 4 create the first rule using the typicality measure, while the fourth one starts the search with a random example.

As mentioned earlier, the experiments whose results are reported in Table 6.1 generated 500 GGP-RIs. Here we are going to show two GGP-RIs, selected out or those 500 GGP-RIs. The selection procedure was based on both the originality of the algorithm (which is inevitably, to some extent, a subjective criterion) and its accuracy in the test set, although the second criteria does not have much statistical support, as it is based on only one run of the algorithm (corresponding to a single test set, i.e., a single fold of the cross-validation procedure). Alg. 6.1 was evolved for the data set *crx*. It produces the initial rule based on a typical example extracted from the training set (details on how to select the typical example can be found in Section 4.2), and remove two conditions-at-a-time from it. The generated candidate rules are evaluated using their information content, and rules are refined until the best rule found so far does not cover any negative examples. Rules can be pre-pruned before being inserted into the rule list by removing the last condition inserted to it. The rule production process is carried out until 99% of the examples in the training set are covered. A post-pruning phase also allows the algorithm to remove two conditions-at-a-time from each rule in the final rule list, as long as the value of the Laplace estimation in the post-pruned rule set is higher than the value of the Laplace estimation in the original rule list. As the MOGGP-RIs presented in Section 5.8, this algorithm also works with three sets of data, one to grow, one to pre-prune and a third one to post-prune the rules.

This algorithm is innovative in the sense that it starts the search with a typical example (instead of a random one) and removes two conditions-at-a-time from it, accounting for some attribute interaction. Recall that the selection of a typical example, based on the principles used in instance-based learning algorithms, is

Algorithm 6.1: Example of a decision list algorithm created by the GGP specifically for the data set *crx*

```
Divide the training data in Build and PostPrune
RuleList = ∅
repeat
    Divide the Build data in Grow and PrePrune
    bestRule = rule created from a typical example
    candidateRules = ∅
    candidateRules = candidateRules ∪ bestRule
    while negative examples covered by bestRule ≠ ∅ do
        for each candidateRule CR do
            newCandidateRules = ∅
            Remove 2 conditions-at-a-time from CR
            Evaluate CR using its information content in Grow
            newCandidateRules = newCandidateRules ∪ CR
        candidateRules = best 10 rules selected from newCandidateRules
    bestRule' = Remove the last condition from bestRule
    Evaluate bestRule' into PrePrune using Laplace estimation
    if bestRule' better than bestRule then
        bestRule = bestRule'
    RuleList = RuleList ∪ bestRule
until at least 99% of the examples in the training set are covered
for each rule R in the RuleList do
    notImproving = false
    repeat
        Remove 2 conditions-at-a-time from R
        Evaluate RuleList' in PostPrune
        if laplaceEstimation(RuleList') ≥ laplaceEstimation(RuleList) then
            RuleList = RuleList'
        else notImproving = true
    until notImproving
```

as innovative feature of the grammar that is not found in any manually-designed rule induction algorithm, to the best of our knowledge. The fact that the rules are refined until no negative examples are covered could lead to over-fitting. However, the generalization capability of the classifier is secured by both pre- and post-prune phases. First the rules are pre-pruned by removing the last condition added to it, and evaluated using the Laplace estimation this time. Later, a post-process phase helps to ensure a compact rule list.

Alg. 6.2 shows a GGP-RI produced for the data set *ionosphere*. The algorithm produces a rule set using a top-down strategy. The novelty of this algorithm lies in the way rules are refined, depending on how many conditions they have. Two

Algorithm 6.2: Example of a rule set algorithm created by the GGP specially for the data set *ionosphere*

```
RuleSet = ∅
for each class C in the training set do
    repeat
        bestRule = an empty rule
        candidateRules = ∅
        candidateRules = candidateRules ∪ bestRule
        while candidateRules ≠ ∅ do
            for each candidateRule CR do
                newCandidateRules = ∅
                if number of conditions in CR < 5 then
                    Add 2 conditions-at-a-time to CR
                else
                    Add 1 condition-at-a-time to CR
                Evaluate CR using accuracy
                if accuracy(CR) > 80% then
                    newCandidateRules = newCandidateRules ∪ CR
            candidateRules = 3 best rules selected from newCandidateRules
        RuleSet = RuleSet ∪ bestRule
    until at least 97% of the examples of class C in the training set are
          covered
Class clashes when classifying new examples are solved using the ls-content
criterion
```

conditions-at-a-time are added to rules with less than 5 conditions, while only one condition-at-a-time is added to rules with more than 5 conditions. A beam search of size 3 is performed, and rules are evaluated using accuracy. The accuracy of the rules also excludes rules with less than 80% accuracy from the set of candidates. Rules are produced for a given class until 97% of the examples of the respective class are covered.

6.2.1 GGP-RIs *versus* GHC-RIs

As we did for the experiments in Section 5.7, we compared the predictive accuracies obtained by the GGP-RIs with the ones obtained by the rule induction algorithms evolved by a grammar-based hill climbing method, i.e. GHC-RIs. Table 6.2 shows the results of these comparisons. Cells in dark gray represent cases where the GGP-RIs obtained predictive accuracies significantly better than the GHC-RIs according to a 2-tailed t-test with significance level 0.01.

Table 6.2: Comparing the predictive accuracies of the GGP-RIs and the GHC-RIs tailored to a specific data set for experiments using a data sub-sets of the same application domain in the meta-training and meta-test sets

Data Set	GGP-RIs	GHC-RIs
balance-scale	79.77±1.11	75.65±1.18
crx	82.22±1.51	80.5±0.87
glass	65.36±1.89	54.42±1.27
heart-c	78.67±1.37	78.72±0.88
hepatitis	80.02±1.07	78.66±0.8
ionosphere	85.65±2.85	83.43±1.32
lymph	75.85±2.04	73.47±1.71
monks-1	100±0	95.62±1.75
monks-2	89.67±1.22	69.53±1.2
monks-3	98.38±0.6	84.12±2.07
mushroom	100±0	97.31±1.43
pima	73.57±0.53	71.74±0.93
promoters	76.07±3.36	67.48±2.31
segment	95.55±0.25	87.96±0.71
splice	89.53±0.65	77.63±1.32
sonar	70.9±2.27	63.19±2.61
vehicle	68.34±0.95	56.65±4.25
vowel	73.66±1.11	42.88±3.35
wisconsin	94.27±1.1	92.32±0.78
zoo	91.85±1.87	89.6±0.83

As illustrated in Table 6.2, the GGP-RIs obtained significantly better accuracies than the GHC-RIs in 10 cases. In 3 out of these 10 cases, namely *vowel*, *vehicle* and *glass*, the accuracies found by the GHC-RIs were surprisingly low. In the case of *glass*, the type of algorithms generated explains the poor accuracy. It was caused by the GHC-RIs which used a pre-pruning method requiring a different set of data, and also a post pruning method (also requiring a different set of data). As *glass* was trained with only 129 examples, the data sets reserved for pre- and post-pruning were not big enough to give statistical support to the results. We recognized earlier on that this division of data might cause problems in small data sets. However, the GGP was able to overcome this problem, and only 4 out of the 25 GGP-RIs generated use a pre-pruning method which requires a different set of data, while none of those 25 algorithms use a post-pruning method. By contrast, the GHC was not able to detect the problem.

Algorithm 6.3: Example of a decision list algorithm created by the GHC specially for the data set *crx*

```
Divide the training data in Build and PostPrune
RuleList = ∅
repeat
    Divide the Build data in Grow and PrePrune
    bestRule = rule created from a typical example
    candidateRules = ∅
    candidateRules = candidateRules ∪ bestRule
    while candidateRules ≠ ∅ do
        for each candidateRule CR do
            newCandidateRules = ∅
            Remove 2 conditions-at-a-time from CR
            Evaluate CR using its confidence in Grow
            if accuracy(CR)> 70% then
                newCandidateRules = newCandidateRules ∪ CR
        candidateRules = 2 best rules selected from newCandidateRules
    bestRule' = bestRule
    notImproving = false
    repeat
        bestRule' = Remove the last condition from bestRule
        Evaluate bestRule' in PrePrune using Laplace Correction
        if bestRule' better than bestRule then
            bestRule = bestRule'
        else notImproving = true
    until notImproving
    RuleList = RuleList ∪ bestRule
until all the examples in the training set are covered
notImproving = false
repeat
    RuleList' = Remove last rule from RuleList
    if accuracy(RuleList') ≥ accuracy(RuleList) in PostPrune then
        RuleList = RuleList'
    else notImproving = true
until notImproving
```

Here we show the pseudo-codes of two GHC-RIs, selected out of the 500 GHC-RIs generated in the experiments whose results are reported in Table 6.2. To be consistent and allow comparisons among the GGP and GHC results, we present the GHC-RIs found when running the GHC with the same random seeds and data sets used by the GGP to produce Alg. 6.1 for *crx* and Alg. 6.2 for *ionosphere*. These GHC-RIs are described in Algs. 6.3 and 6.4. In general, the algorithms found by the GHC when using one data set in the meta-training set (in order to

evolve a rule induction algorithm tailored to a single application domain) were also more original than the ones found by the GHC when using a variety of data sets in the meta-training set (in order to evolve a more robust rule induction algorithm).

Alg. 6.3 presents a similar algorithm to the one presented in Alg. 6.1 (GGP-RI for *crx*). The main differences between these two algorithms lie on the rule evaluation heuristic, the beam width in the beam search and the post-processing method. Besides, the GGP-RI requires 99% of the examples in the training set to be covered, while the GHC-RI requires all. Note that Alg. 6.3 also filters the candidate rules according to their accuracy, which has to be higher than 70%.

In Alg. 6.3, the confidence of the rules is used both as the rules' evaluation heuristic and pre-pruning method, since the accuracy of the rules has to be greater than 70% (recall that in this context accuracy is a synonym for confidence). The beam width also changes from 10 to 2 from Alg. 6.1 to Alg. 6.3, as does the post-processing method. While Alg. 6.1 post-prunes rules by removing two conditions-at-a-time from them, Alg. 6.3 removes entire rules from the rule list. In conclusion, for this particular data set, the GHC was able to generate a solution similar to the one generated by GGP, with a competitive predictive accuracy.

When comparing the rule induction algorithms generated for *ionosphere* by the GGP (Alg. 6.2) and the GHC (Alg. 6.4), they are not similar at all. Alg. 6.2 creates a rule set, uses a top-down approach and refines rules according to their size, while Alg. 6.4 creates a rule list, works in a bottom-up fashion and removes two conditions-at-a-time from an initial rule, created from a random example. Alg. 6.4 also refines rules until they do not cover any negative examples, and uses the confidence to evaluate the candidate rules and the accuracy as a stopping criterion. Alg. 6.4 also post-processes rules by removing the last rule inserted into the rule model.

In terms of predictive accuracy, considering 10 runs of Alg. 6.4 (as the algorithm is not deterministic, due to the random selection of a seed example to form the first rule), it could never obtain predictive accuracies equal or higher than the ones obtained by Alg. 6.2.

Algorithm 6.4: Example of a decision list algorithm created by the GHC specially for the data set *ionosphere*

```
RuleList = ∅
Divide the training data in Build and PostPrune
repeat
    bestRule = rule created from an example randomly chose from the
               training set
    candidateRules = ∅
    candidateRules = candidateRules ∪ bestRule
    while candidateRules ≠ ∅ do
        for each candidateRule CR do
            newCandidateRules = ∅
            Remove 2 conditions-at-a-time from CR
            Evaluate CR using the confidence in Build
            if accuracy(CR) > 70% then
                newCandidateRules = newCandidateRules ∪ CR
        candidateRules = 10 best rules selected from newCandidateRules
    RuleList = RuleList ∪ bestRule
until all examples in the training set are covered
notImproving = false
repeat
    RuleList' = Remove last rule from RuleList
    if accuracy(RuleList') ≥ accuracy(RuleList) in PostPrune then
        RuleList = RuleList'
    else notImproving = true
until notImproving
```

6.3 Experiments with Bioinformatics Data Sets

In Section 6.2 we showed the results obtained by the GGP when evolving rule induction algorithms tailored to specific data sets from the well-known UCI repository. In this section, we apply the GGP system to evolve rule induction algorithms tailored to each of the 5 bioinformatics data sets described in Table 6.3. In all of these data sets, each example represents a protein. Proteins are the main elements of the cell, and perform almost all the functions related to cell activity. Their primary structure is formed from a sequence of amino acids, which are held together by covalent ("strong") bonds.

Each of the proteins in these data sets is described mainly by a set of motifs. A motif is a pattern or a "signature" typically found in some proteins. It is basically a sequence/partial sequence of amino acids which can be used to identify the function and/or the family of a protein [68]. Each motif is represented by a binary

Table 6.3: Bioinformatics data sets used by the GGP

Data set	Examples	Attributes		Classes	Def. Acc.
		Nomin.	Numer.		(%)
Postsynaptic	4303	10(444)	0(2)	2	93.96
GPCR-Prosite	6261	30(128)	2(2)	9	75.16
GPCR-Prosite-L2	6162	25(128)	2(2)	50	33.7
GPCR-Prints	5422	24(282)	2(2)	8	77.8
GPCR-Interpro	7461	30(449)	2(2)	12	64.4

attribute, which indicates the presence of absence of the motif in the protein.

Table 6.3 shows the number of examples, number of attributes and number of classes for each bioinformatics data set, followed by the default accuracy – i.e., the percentage of examples (proteins) present in the class of the majority of examples. Observe that, in the column "Attributes", the numbers in parentheses show the original number of attributes the data sets had before an attribute selection method was applied during a pre-process step.

The data sets showed in Table 6.3 were pre-processed by using an attribute selection method for two reasons. First, due to the large number of predictive attributes in the original data set, intuitively there are many attributes that are (to a large extent) irrelevant or redundant. The objective of attribute selection is to simplify a data set by reducing its dimensionality and identifying relevant attributes without sacrificing predictive accuracy [92]. Second, attribute selection usually significantly reduces the size of the data sets. This makes the application of the GGP algorithm much more efficient.

For the data set postsynaptic, we used the 10 top attributes selected by a particle swarm optimization algorithm [34], since these selected attributes are already available in the literature. Note that in [34] a wrapper approach is used to select a set of optimal attributes for a naive-bayes classifier. We recognize that these attributes are likely to be not optimized for the algorithms considered in this work. Nevertheless, the issue of optimal attribute selection is out of the scope of this work and, as the GGP-RI is produced during the GGP run, is not possible to perform a customized attribute selection to it beforehand. For the other four data sets, an attribute selection method based on the attributes' gain ratios [145], as applied to *splice* in Section 6.2, was used. Recall that, in this approach, the attributes were ranked according to their gain ratio, and then removed, 10 by 10, from the worst to the best (according to their rank), from the original data set.

Every time 10 attributes were removed from the training set, the new data set was then mined by the three baseline methods following the sequential covering approach (i.e, both versions of CN2 and Ripper). Note that C4.5Rules was not considered during this process because, for most of the data sets, it was not able to extract a set of decision rules from the built decision tree. This process of removing 10 attributes from the training set was repeated until the predictive accuracy of at least one of baseline methods dropped. In this case, the last 10 removed attributes were added again to the training set and then removed, one by one, from the worst to the best, while the predictive accuracy of the classifiers did not decrease.

The 5 data sets listed in Table 6.3 are related to two different application domains (corresponding to two types of proteins): protein postsynaptic activity and G-Protein-Coupled-Receptors (GPCR) families. The creation of the postsynaptic data set is described in detail in [111] – previous work of the author of this thesis. Identifying proteins with postsynaptic activities is of great intrinsic interest because they are connected with functioning of the nervous system. In turn, identifying GPCR proteins and their families is particularly important for medical applications, since it is believed that 40%-50% of current drugs target GPCR activity. The creation of the GPCR data sets is described in [73].

While one data set targets the application domain of postsynaptic proteins, the other four were built to characterize GPCR proteins. Each GPCR data set uses a different type of motif, extracted from a different biological database (i.e, Prosite, Interpro and Prints), to describe the data [68].

An important point to notice in the GPCR data sets is that the classes of GPCR proteins are organized in a hierarchical manner. Actually, these data sets were used to evaluate a hierarchical classification algorithm in [73]. Since hierarchical classification is out of the scope of this thesis, these data sets were "flattened". All the GPCR data sets, except GPCR-Prosite-L2, consider just the classes in the first top level of the class hierarchy, ignoring lower-level classes. GPCR-Prosite-L2, in turn, considers the classes in the second level of the class hierarchy. Note that GPCR-Prosite-L2 is composed essentially by the same proteins and attributes found in GPCR-Prosite. However, GPCR-Prosite considers, as class values, only the 9 classes present in the first hierarchical class level (ignoring lower-level classes), while GPCR-Prosite-L2 considers only the 50 classes in the second hierarchical class level (ignoring both first- and lower-level classes). The differences in the number of examples when comparing these 2 data sets are due

Table 6.4: Comparing the predictive accuracies (%) obtained by the GGP-RIs in the bioinformatics data sets with selected attributes against the predictive accuracies (%) obtained by the baseline methods when using the complete data set

Data Set	GGP-RIs	OrdCN2	UnordCN2	Ripper	C45Rules
Postsynaptic	98.32±0.24	98.7±0.22	98.42±0.12	98.3±0.22	97.82±0.32
GPCR-Prosite	86.44±0.13	87.34±0.32	75±0.2	87.66±0.39	82.64±2.44
GPCR-Prosite-L2	63.81±0.96	68.56±0.29	48.9±0.5	62.56±0.75	44±1.07
GPCR-Prints	90.71±0.69	91.04±0.25	83.84±1.52	91.06±0.27	88.94±2.76
GPCR-Interpro	89.34±0.29	90.88±0.45	90.56±0.41	90.26±0.32	80.36±6.35

to proteins where only the first class in the hierarchy are known. Of course, these proteins are used as examples in GPCR-Prosite but not in GPCR-Prosite-L2.

Experiments with the GGP were performed using the bioinformatics data sets with selected attributes only. Each data set was divided into three subsets, and inserted in the meta-training and meta-test sets of the GGP, as described in Section 6.1. For these experiments, the comparisons of the GGP-RIs and the baseline methods were executed in two phases. First, the GGP-RIs' predictive accuracies were compared to the accuracies obtained by the baseline methods when run with the complete data sets (before any attribute selection process was applied). The purpose of this analysis was to show that the attribute selection process did not have significantly reduce the predictive accuracies obtained by the baseline methods in the full data sets. These results are reported in Table 6.4. In a second phase, comparisons among the GGP-RIs and the baseline methods run in the data sets with selected attributes were carried out. The results obtained are shown in Table 6.5.

Both Tables 6.4 and 6.5 show the names of the data sets where the experiments were executed, followed by the predictive accuracies obtained by the GGP-RIs in the data sets with selected attributes. The following columns show the predictive accuracies obtained by Ordered-CN2, Unordered-CN2, Ripper and C4.5Rules for the complete data sets in Table 6.4; and for the pre-processed data sets (with selected attributes) in Table 6.5. Note that, for the bioinformatics data sets, the C4.5Rules algorithm was not able to extract rules from the original C4.5 tree in most of the experiments performed. In these cases, for the sake of completeness, the accuracy reported is the one obtained by the C4.5 tree model.

In Tables 6.4 and 6.5, cells in dark gray represent statistically significant wins of the GGP-RIs over the respective baseline method, whereas cells in light gray

Table 6.5: Comparing the predictive accuracies (%) obtained by the GGP-RIs and the baseline methods when using the bioinformatics data sets with selected attributes

Data Set	GGP-RIs	OrdCN2	UnordCN2	Ripper	C45Rules
Postsynaptic	98.32±0.24	98.4 ± 0.2	98.4 ± 0.2	98.21 ± 0.22	98.4 ± 0.2
GPCR-Prosite	86.44±0.13	86.86±0.28	75.68±0.31	86.64±0.31	84.04±2.04
GPCR-Prosite-L2	63.81±0.96	68.38±0.65	47.98±0.1	62.36±0.42	NA
GPCR-Prints	90.71±0.69	90.78±0.32	90.64±0.37	91.05±0.14	86.76±2.58
GPCR-Interpro	89.34±0.29	90.2±0.25	88.46±0.55	89.83±0.32	84.02±3.73

represent statistically significant wins of the baseline method over the GGP-RIs according to a 2-tailed t-test with significance level 0.05.

As showed in Table 6.4, the results obtained by the GGP-RIs with selected attributes are significantly better than the results obtained by baseline methods with the complete data set in four cases, and significantly worse in three cases. The GGP-RIs are significantly better than Unordered-CN2 in three out of four cases, and significantly better than C4.5Rules in the data set GPCR-Prosite-L2. At the same time, the GGP-RIs are significantly worse than Ordered-CN2 in GPCR-Prosite and GPCR-Prosite-L2, and significantly worse than Ripper in GPCR-Prosite.

If we compare these results with the ones reported in Table 6.5, we notice that now the GGP-RIs are significantly better than the baseline methods in two occasions, and statistically worse in only one. This is because the attribute selection process improved the accuracy of Unordered-CN2 in GPCR-Prints, but also decreased the accuracy of Ordered-CN2 and Ripper in GPCR-Prosite, making the corresponding previous wins/loss of the GGP-RIs not statistically significant anymore. Note that the predictive accuracy for C4.5Rules in the data set GPCR-Prosite-L2 with selected attributes is not available. This is because C4.5 presented an error when it was being executed in this data set and was not able to generate any classification model.

These results show us that the GGP is able to obtain statistically better results than Unordered-CN2 in two data sets, and competitive ones in the other three data sets. At the same time, for the data set GPCR-Prosite-L2, the results show that apparently the GGP was not able to find Ordered-CN2 – the algorithm with best performance among the 4 manually-designed algorithms in Table 6.5, although Ordered-CN2 is included in the GGP search space. However, when analyzing the 25 GGP-RIs produced for GPCR-Prosite-L2, we observe that 6 of them were

Table 6.6: Sensitivity x Specificity for the data set postsynaptic

Algorithm	Sensitivity	Specificity	Sensit×specif
GGP-RIs	0.74±0.004	0.99±0.0001	0.736±0.004
Ordered-Cn2	0.76±0.02	0.99±0.001	0.758±0.02
Unordered-Cn2	0.76±0.02	0.99±0.001	0.758±0.02
C45Rules	0.75±0.03	0.99±0.001	0.748±0.03
Ripper	0.7±0.04	0.99±0.001	0.702±0.04

actually instances of Ordered-CN2. Results obtained by the GGP-RIs were always as good as the results obtained by Ripper and C4.5Rules for all the data sets.

Again, based on these results, we can claim that the GGP is able to produce competitive rule induction algorithms tailored to a specific real world data set. Nonetheless, for one of these data sets, one could argue that the analysis of the GGP-RIs results based on accuracy is not a very effective one.

Notice that, as shown in Table 6.3, the class distribution of the postsynaptic data set is very unbalanced: only 6.04% of the examples have the positive class. This means that, as a baseline solution for this classification problem, the "majority classifier" - which predicts the majority (negative) class for all examples - would trivially obtain an accuracy rate of 93.96%. This value could be obtained without providing any insight about the relationship between the predictor attributes and the classes.

For data sets in which the class distribution is very unbalanced, an analysis based on the true positive rate (sensitivity) and true negative rate (specificity) is more effective [66]. Another alternative would be to use a measure based on ROC curves, such as the area under the ROC curve. We preferred a measure based on sensitivity and specificity to be consistent with the results for this data set reported in [111]. The sensitivity × specificity measure, introduced in Section 4.5.2 (Eq. 4.3), calculates the product of the true positive and true negative rates. We used this measure to reevaluate the results obtained by the GGP-RIs for the data set postsynaptic.

Table 6.6 shows the sensitivity, specificity and the product sensitivity × specificity obtained by the GGP-RIs and the other baseline methods for the postsynaptic data set. All the results showed in Table 6.6 present no significant difference according to a 2-tailed t-test with 0.01 significance level.

The analysis of the GGP-RIs results for the data set postsynaptic using a

sensitivity × specificity approach confirmed that the GGP-RIs produced are competitive with the other baseline methods. Both the GGP-RIs and the baseline methods not only obtain a high accuracy, but also have a good ability to separate objects from the positive and negative classes – rather than simply predicting the majority class for all the test examples – as shown by the relatively high values of sensitivity. After this new analysis, a new question came up. For data sets like postsynaptic, where the class distribution is very unbalanced, would it be worth to actually evolve the GGP with a different fitness function? That is, would it be worth to consider the sensitivity × specificity measure, for instance, as the GGP evaluation function during the evolution? This would make sense, once we know that, in the context of very unbalanced class distributions, predictive accuracy is not a very effective measure to evaluate the predictive power of classification models.

In Section 4.5.2 we reported that, based on the results of preliminary experiments with the GGP when evolving robust rule induction algorithms, a fitness function based on the sensitivity × specificity measure was not as effective as a fitness function based on the normalized value of accuracy. But would a GGP using only the postsynaptic data set in the meta-training set, and a fitness function based on sensitivity × specificity, generate better results than the GGP with the normalized accuracy fitness?

In order to find answers for this question, we run a set of experiments almost identical to the ones described so far in this section. However, we replaced the current fitness of the GGP by the sensitivity × specificity measure, and used the postsynaptic data set in the meta-training and meta-test sets. Surprisingly, the average predictive accuracy obtained over the 25 runs of the GGP was 92.85±0.03. This predictive accuracy value is slightly smaller than the default accuracy provided by the classification model using the class of the majority of the examples (93.96%), and it is significantly worse than the values of accuracy obtained by any of the baseline methods (and the GGP-RIs) presented in Tables 6.4 and 6.5.

However, as explained before, in the case of the postsynaptic data set, an analysis based on the sensitivity × specificity measure is more appropriate than one based on predictive accuracy. The sensitivity × specificity value obtained (on the test set) for these experiments using sensitivity × specificity as the fitness function of the GGP was 0.789±0.04. This value is statistically the same as all the values in the last column of Table 6.6. On the other hand, a more detailed analysis of the sensitivity and specificity values (on the test set) separately showed

a specificity (the proportion of negative – majority class – examples that are correctly predicted as negative) of 0.93±0.026 and a sensitivity (the proportion of positive – minority class – examples that are correctly predicted as positive) of 0.84±0.008. If we compare these values to the ones presented at Table 6.6, we notice that the specificity dropped from 0.99 to 0.93 and the sensitivity increased from 0.75 to 0.84. According to a 2 tailed t-test with 0.05 significance level, the sensitivity of the GGP-RIs is significantly better than the sensitivity of all the baseline methods presented in Table 6.6, while the specificity of the GGP-RIs is significantly worse than the specificity of all the baseline methods.

From this we conclude that the GGP-RIs found with the sensitivity × specificity fitness produced algorithms which are better when predicting the class of the minority of the examples, which is more difficult to predict and tends to be a prediction more useful to the user, by comparison with a prediction of the majority class [37, 105]. At the same time, these GGP-RIs are not able to preserve a high specificity – true negative rate for the majority class.

This last experiment just confirmed that the fitness of the current GGP is robust enough to produce robust algorithms even for data sets with very unbalanced classes. But what was so different in the GGP-RIs produced by these two versions of the GGP using different fitness functions?

An analysis of the rule induction algorithms produced by the GGP when using the normalized accuracy or the sensitivity × specificity measures as the fitness function revealed algorithms following two completely different approaches (we emphasize that this analysis refers only to the postsynaptic data set).

On one hand, the majority of the GGP-RIs produced with the normalized accuracy fitness follows one of the following 2 main approaches: (1) they create an initial empty rule and add conditions to it, or (2) they build an initial rule using 3 or 4 of the most frequent attribute/value pairs found in the training data, and remove conditions from it. Algorithms following any of these 2 approaches produce very compact and general rules, usually with a maximum of 3 or 4 conditions each. Alg. 6.5 shows an example of one of these algorithms produced by the GGP. It creates the first rule with the 3 most frequent attribute/values pairs in the data, and starts by removing one condition-at-a-time from it. As soon as the number of examples covered by the rule list is greater than 95%, it changes its refinement strategy by removing two conditions-at-a-time from the candidate rules. Rules are evaluated using the rule confidence measure, which is required to be at least equal to 60% to enable the candidate rule to undergo further refinements (recall

Algorithm 6.5: Example of a decision list algorithm created by the GGP – with a normalized fitness function – tailored to the data set *postsynaptic*

```
RuleList = ∅
repeat
    bestRule = rule created using the 3 most frequent attribute/values in
               the training data
    candidateRules = ∅
    candidateRules = candidateRules ∪ bestRule
    while candidateRules ≠ ∅ do
        for each candidateRule CR do
            newCandidateRules = ∅
            if number of covered examples in RuleList > 95% then
                Remove 2 conditions-at-a-time from CR
            else
                Remove 1 condition-at-a-time from CR
            Evaluate CR using confidence
            if accuracy(CR) > 60% then
                newCandidateRules = newCandidateRules ∪ CR
        candidateRules = best rule selected from newCandidateRules
    RuleList = RuleList ∪ bestRule
until all examples in the training set are covered
```

that in this context rule confidence and rule accuracy are synonyms). Only the best rule is selected to be refined. Rules are produced until all the examples in the training set are covered.

In contrast with the types of GGP-RIs produced when using the normalized accuracy measure as the fitness function, most of the GGP-RIs produced when using the sensitivity $\times$ specificity measure as fitness followed a bottom-up approach. Rules were initialized using a random example or a typical example, and most of the actual rule models had very specific rules (rules with many conditions). Looking at the predictive accuracies obtained by these algorithms we notice that most of them seemed to be over-fitting the training data. Regardless of that, as reflected by the sensitivity (true positive rate) obtained by these algorithms on the test set, they were able to generate significantly better rules to predict the minority class. Alg. 6.6 shows an example of a GGP-RI produced by the GGP using the sensitivity $\times$ specificity fitness function.

Alg. 6.6 creates a set of rules for each class in turn. It chooses a typical example from the training set and removes one condition-at-a-time from it. When 95% of the examples belonging to the current class are covered, the algorithm starts to remove 2 conditions-at-a-time from the candidate rules. The 4 best rules are

Algorithm 6.6: Example of a rule set algorithm created by the GGP – with a sensitivity × specificity fitness function – tailored to the data set *postsynaptic*

```
RuleSet = ∅
for each class C in the training set do
    repeat
        Divide the training data in Grow and Prune
        bestRule = rule created from a typical example
        candidateRules = ∅
        candidateRules = candidateRules ∪ bestRule
        while negative examples covered by bestRule ≠ ∅ do
            for each candidateRule CR do
                newCandidateRules = ∅
                if number of covered examples in class C > 95% then
                    Remove 2 conditions-at-a-time from CR
                else
                    Remove 1 condition-at-a-time from CR
                Evaluate CR using information content in Grow
                if accuracy(CR) > 80% then
                    newCandidateRules = newCandidateRules ∪ CR
            candidateRules = 4 best rules selected from newCandidateRules
        notImproving = false
        repeat
            bestRule' = Rule obtained by removing the last condition from
                        bestRule
            if laplace(bestRule') ≥ laplace(bestRule) in Prune then
                bestRule = bestRule'
            else notImproving = true
        until notImproving
        RuleSet = RuleSet ∪ bestRule
    until at least 97% of the examples of class C in the training set are
          covered
Class clashes when classifying new examples are solved using the ls-content
criterion
```

selected to undergo further refinements, which are carried out until the best rule found so far covers no negative examples. Candidate rules are evaluated using the information content, and are also required to have an accuracy of at least 60% to be considered as candidate rules.

Rules are pre-pruned before being inserted into the rule set by removing a final list of conditions from them. Conditions are removed from the best produced rule until the Laplace correction value of the new pruned rule is worse than the Laplace correction value of the best rule in the prune set. Rules are produced until at least

97% of the examples in the current class are covered.

In summary, this section showed that the GGP can produce GGP-RIs tailored to a specific real world data set which are competitive with well-known human-designed rule induction algorithms. It also showed that the system coped well with the problem of unbalanced classes in the postsynaptic data set, in particular when using the sensitivity $\times$ specificity fitness function, which leads to a better prediction of the minority class (whose prediction tends to be more important to the user than the prediction of the majority class).

6.4 A Note on the GGP System's Execution Time

In Section 5.9, we discussed the factors which influence the execution time of the GGP, and explained why is so difficult to estimate a best/worst runtime for the GGP. We then reported results for the best and worst case scenarios empirically obtained when running the proposed GGP with different numbers of data sets in the meta-training set.

Similarly, in this section, Table 6.7 reports that same kind of information when running the GGP with a single data set in the meta-training set. The first column shows the name of the data set in the meta-training set, followed by the best and worst run times, reported in the format hours:minutes.

Out of the 25 data sets used in those experiments (i.e. 20 UCI plus 5 bioinformatics data sets), where each data set was used in a separate experiment, we report the computational times for 5 data sets, as follows: *balance-scale* and *post-synaptic* present the fastest run among the UCI and the bioinformatics data set, respectively, while *promoters* and *GPCR-Interpro* present the slowest. The times reported for *lymphs* represent an average time for middle size UCI data sets.

6.5 Summary

This chapter presented the results obtained by the GGP system proposed in Chapter 4 when evolving rule induction algorithms tailored to one specific data set. Experiments were first performed in 20 UCI data sets, and later in a set of 5 real-world bioinformatics data sets.

For all the experiments reported in this chapter, the predictive accuracies obtained by the GGP-RIs were compared to the predictive accuracies of four

Table 6.7: GGP runtime for experiments targeting one data set

Meta-training set	Time (hr:min)	
	Best	Worst
Balance-scale	0:21	0:45
Postsynaptic	1:41	4:56
Lymphs	1:20	3:05
Promoters	15:37	43:18
GPCR-Interpro	63:26	169:30

well-known human-designed rule induction algorithms, namely Ordered-CN2, Unordered-CN2, Ripper and C4.5Rules. In general, the results showed that the GGP-RIs were competitive with these baseline methods. They have also showed some of the evolved GGP-RIs, and highlighted their innovative features.

At last, a case study in the postsynaptic data set showed that the proposed GGP is also robust when evolving GGP-RIs for data sets with special characteristics, like very unbalanced class distributions.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

This thesis has presented a new Grammar-based Genetic Programming (GGP) system which automatically evolves rule induction algorithms. The GGP system works with a grammar, which contains previous knowledge about how human experts design rule induction algorithms, and some other interesting components that, to the best of our knowledge, were not used by human-designed rule induction algorithms so far.

The results of extensive computational experiments showed that the present GGP system can effectively evolve rule induction algorithms under two different frameworks:

- The evolution of *robust* rule induction algorithms, which perform well in a variety of data sets different from the data sets used during the GGP's run.
- The evolution of rule induction algorithms tailored to a single specific application domain, where the data subset used during the GGP's run belongs to the same application domain to which the evolved rule induction algorithm will be applied.

The predictive accuracies obtained by the rule induction algorithms evolved by the GGP (GGP-RIs) were shown to be, overall, slightly better than the predictive accuracies obtained by well-known manually designed (and refined over decades of research) rule induction algorithms.

The experimental results also showed that overall the GGP-RIs obtained significantly higher predictive accuracies than the rule induction algorithms produced

by a Grammar-based Hill Climbing (GHC) method. These results were produced in controlled experiments where both the GGP and the GHC used the same grammar, the same fitness function, the same individual representation and evaluated the same number of candidate rule induction algorithms during their search. Hence, one can conclude that the GGP system is considerably more effective than the GHC system.

An analysis of the evolved GGP-RIs showed that, besides being competitive with human-designed algorithms, many of them present some innovative way of refining rules and/or integrating pre and post-pruning techniques. Experiments also revealed that, when evolving *robust* rule induction algorithms, the GGP takes a more conservative approach, and builds simpler algorithms. In contrast, when generating GGP-RIs tailored to a specific application domain, the GGP evolves more different and innovative algorithms.

Moreover, the multi-objective version of the GGP (MOGGP), presented in Section 5.8, seems to be a promising one. Besides generating MOGGP-RIs with overall predictive accuracies as good as the ones obtained by the human-designed rule induction algorithms, the evolved MOGGP-RIs also generate simpler (smaller) classification models. Generating more compact classification rule models is an important issue in application domains in which the model has to be interpreted by a human expert before any decision making process takes place. This is in general recognized as an important issue in data mining [49, 145].

In summary, this thesis has presented a new approach to automatically construct rule induction algorithms. These algorithms are specially promising when tailored to a specific data set. In this context, the proposed method is a new and promising alternative to traditional meta-learning methods that are limited to algorithm selection, rather than algorithm construction.

Although this thesis has presented an effective GGP system and reported extensive experimental results, it still leaves room for several improvements and future research directions. These new ideas are presented in Section 7.2.

7.2 Future Work

This section discusses five possible future directions for this research, namely: an implementation of the system using a GGP based on the production-rule-sequence-encoding individual representation, a modified GGP fitness, improvements of the

grammar, the use of GGP-RI ensembles, and a new approach to build rule induction algorithms targeting a group of data sets with similar characteristics.

7.2.1 Solution-Encoding Individual Representation *versus* Production-Rule-Sequence-Encoding Individual Representation

When choosing which type of grammar-based GP to use in this research, we did not find any evidence which showed that one of the individual representations in the title of this subsection was superior to the other. Hence, we decided to use the solution encoding individual representation because we considered it simpler (see Section 4.1 for detailed explanation).

However, it would be interesting to combine the grammar developed into this work with a grammatical evolution system, following the production-rule-sequence-encoding individual representation, and compare both systems' behavior and results.

7.2.2 Modifying the GGP Fitness Function

After the grammar, the element which has more impact in the search mechanism of the GGP is its fitness function. During the development of the system, we compared the results obtained when using three types of fitness functions: accuracy, sensitivity $\times$ specificity (see Eq. 4.3) and the current fitness of the system, based on a variation of accuracy adjusted to take into account the frequency of the majority class (see Eq. 4.4).

Nevertheless, in the last few years, another approach for evaluating the effectiveness of a classifier has been widely disseminated: the area under the ROC curve (AUC) [48, 50]. A comparison of the results of the GGP with the current fitness function and the AUC would be interesting to study.

7.2.3 Improvements of the Grammar

The grammar is the element in the GGP system which determines the GGP's search space. We believe that there are some extensions that, if inserted into the current grammar, could lead the GGP to evolve even more original and innovative rule induction algorithms. There are particulary three extensions which could be introduced:

- Include in the non-terminal *EvaluateRule* the grammar production rules proposed in Wong [147], which generate the evaluation function of rule induction algorithms. The new grammar would then be able to generate new evaluation heuristics apart from the current rule confidence, Laplace estimation, information content and information gain.

- Insert into the grammar more complex and/or innovative components of rule induction algorithms, such as the minimum description length [123] heuristic, used by Ripper, or measures of rule interestingness, as described in Section 2.2.3.

- Extend the grammar to be adaptable, based on the performance of the best individuals at each generation, in a similar way to the work of [144].

7.2.4 Ensembles of Evolved Rule Induction Algorithms

In the experiments presented in Chapter 5, at each generation 100 individuals are evaluated. After 30 generations, one single rule induction algorithm is chosen out of 100. We might then think: is not it a waste of computational resources to evolve 100 different algorithms and ignore 99 of them?

Research in the area of ensembles of classifiers, i.e., a set of classifiers whose individual predictions are combined to classify new examples, has demonstrated that combining classifiers can really obtain better prediction accuracies than using a single classifier selected out of all classifiers in the ensemble [45]. We could combine the GGP-RIs produced in the GGP's last generation into a voting [40] or a stacking framework [146], and check if the results obtained would be superior to the ones obtained by single GGP-RIs.

However, it should be noted that, although an ensemble of GGP-RIs has a good potential to improve the predictive accuracy with respect to a single GGP-RI, an ensemble has the disadvantage of producing a much more complex classification model, therefore hindering the interpretability of the model by the user.

7.2.5 Constructing Rule Induction Algorithms Targeted to a Group of Data Sets with Similar Characteristics: a New Approach

Recall that this thesis presented two approaches in which the GGP was able to evolve rule induction algorithms. In the first one, we generated *robust* rule induction algorithms, which were designed to be effectively applied to any classification data set, regardless of the application domain. In the second one, rule induction algorithms tailored to a specific application domain were generated.

A third approach which could be considered is to use the proposed GGP to produce rule induction algorithms for a group of data sets with similar characteristics, though the data sets come from different application domains. In this approach, data sets would be grouped according to some common properties, and only data sets belonging to a given group allowed in the GGP's meta-training set.

For instance, in the bioinformatics field, there are a lot of data sets which – although derived from different application domains – share several important characteristics from a data mining viewpoint. In particular, there are many bioinformatics data sets that contain a very large number of binary attributes, very sparse data and very unbalanced classes. The postsynaptic data set used in this thesis (see Section 6.3) is an example of such data sets. Other examples are found in [72, 73].

It is possible to target a group of data sets with similar characteristics when producing rule induction algorithms, although in practice this is a difficult problem. The main difficulty is how to measure the degree of similarity between different data sets from different application domains. Data set characterization [91, 98] is in general an open problem in the meta-learning literature, but we think that this is an interesting research direction.

References

[1] H. Abe and T. Yamaguchi. Comparing the parallel automatic composition of inductive applications with stacking methods. In R. Camacho and A. Srinivasan, editors, *Proc. of ECML/PKDD'03 - Workshop on Parallel and Distributed Computing for Machine Learning*, pages 1–12, Cavtat-Dubrovnik, Croatia, September 2003.

[2] A. Abraham. Meta learning evolutionary artificial neural networks. *Neurocomputing*, 56:1–38, 2004.

[3] R. Agarwal and M. V. Joshi. Pnrule: A new framework for learning classifier models in data mining. In *Proc. of the 1^{st} SIAM Int. Conf. in Data Mining*, pages 1–17, 2001.

[4] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1^{st} edition, 1986.

[5] K. M. Ali and M. J. Pazzani. Hydra: A noise-tolerant relational concept learning algorithm. In R. Bajcsy, editor, *Proc. of the 13^{th} Int. Joint Conf. on Artificial Intelligence (IJCAI-93)*, pages 1064–1071, 1993.

[6] A. An and N. Cercone. Rule quality measures for rule induction systems: Description and evaluation. *Computational Intelligence*, 17(3):409–424, 2001.

[7] R. Andrews, J. Diederich, and A. Tickle. A survey and critique of techniques for extracting rules from trained artificial neural networks. *Knowledge-Based Systems*, 8(6):373–389, 1995.

[8] P. J. Angeline. Subtree crossover causes bloat. In J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo, editors, *Proc. of the 3^{rd} Annual Conf. on Genetic Programming (GP-98)*, pages 745–752. Morgan Kaufmann, 22-25 July 1998.

[9] R. M. A. Azad. *A Position Independent Representation for Evolutionary Automatic Programming Algorithms - The Chorus System.* PhD thesis, University of Limerick, Ireland, December 2003.

[10] T. Baeck, D. B. Fogel, and Z. Michalewicz. *Evolutionary Computation 1 Basic Algorithms and Operators.* Institute of Physics Publishing, 2000.

[11] W. Banzhaf. Genotype-phenotype-mapping and neutral variation – A case study in genetic programming. In Y. Davidor, H. Schwefel, and R. Männer, editors, *Parallel Problem Solving from Nature III*, volume 866 of *LNCS*, pages 322–332, Jerusalem, 9-14 October 1994. Springer-Verlag.

[12] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications.* Morgan Kaufmann, January 1998.

[13] M. Berthold and D. J. Hand, editors. *Intelligent Data Analysis: An Introduction.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[14] S. Bhattacharyya. Direct marketing response models using genetic algorithms. In *Proc. of 4^{th} Int. Conf. on Knowledge Discovery and Data Mining (KDD-98)*, pages 144–148, 1998.

[15] H. Boström and L. Asker. Combining divide-and-conquer and separate-and-conquer for efficient and effective rule induction. In S. Džeroski and P. Flach, editors, *Proc. of the 9^{th} Int. Workshop on Inductive Logic Programming (ILP-99)*, pages 33–43, 1999.

[16] C. Brodley and M. Friedl. Identifying mislabeled training data. *Journal of Artificial Intelligence Research*, 11:131–167, 1999.

[17] C. A. Brunk and M. J. Pazzani. An investigation of noise-tolerant relational concept learning algorithms. In L. Birnbaum and G. Collins, editors, *Proc. of the 8^{th} International Workshop on Machine Learning*, pages 389–393. Morgan Kaufmann, 1991.

[18] R. Caruana and A. Niculescu-Mizil. Data mining in metric space: an empirical analysis of supervised learning performance criteria. In *Proc. of the 10^{th} ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining (KDD-04)*, pages 69–78. ACM Press, 2004.

[19] D. R. Carvalho and A. A. Freitas. A hybrid decision tree/genetic algorithm for coping with the problem of small disjuncts in data mining. In D. Whitley, D. Goldberg, E. Cantu-Paz, L. Spector, I. Parmee, and H. Beyer, editors, *Proc. of the Genetic and Evolutionary Computation Conference (GECCO-00)*, pages 1061–1068, Las Vegas, Nevada, USA, 10-12 2000. Morgan Kaufmann.

[20] D. R. Carvalho, A. A. Freitas, and N. Ebecken. Evaluating the correlation between objective rule interestingness measures and real human interest. In A. Jorge, L. Torgo, P. Brazdil, R. Camacho, and J. Gama, editors, *Proc. of the 9^{th} European Conf. on Principles and Practice of Knowledge Discovery in Databases (PKDD-05)*, pages 453–461. Springer Verlag, 2005.

[21] M. J. Cavaretta and K. Chellapilla. Data mining using genetic programming: The implications of parsimony on generalization error. In P. J. Angeline, Z. Michalewicz, M. Schoenauer, X. Yao, and A. Zalzala, editors, *Proc. of the Congress on Evolutionary Computation (CEC-99)*, volume 2, pages 1330–1337. IEEE Press, 6-9 July 1999.

[22] J. Cendrowska. Prism: An algorithm for inducing modular rules. *International Journal of Man-Machine Studies*, 27:349–370, 1987.

[23] A Chandra and X. Yao. Ensemble learning using multi-objective evolutionary algorithms. *Journal of Mathematical Modeling and Algorithms*, 5(4):417–445, 2006.

[24] M. Chisholm and P. Tadepalli. Learning decision rules by randomized iterative local search. In L. Birnbaum and G. Collins, editors, *Proc. of the 19^{th} Int. Conf. on Machine Learning (ICML-02)*, pages 75–82. Morgan Kaufmann, 2002.

[25] S. B. Cho and K. Shimohara. Modular neural networks evolved by genetic programming. In *Proc. of the IEEE Int. Conf. on Evolutionary Computation*, pages 681–684. IEEE Press, 1996.

[26] P. Clark and R. Boswell. Rule induction with CN2: some recent improvements. In Y. Kodratoff, editor, *EWSL-91: Proc. of the European Working Session on Learning on Machine Learning*, pages 151–163, New York, NY, USA, 1991. Springer-Verlag.

[27] P. Clark and T. Niblett. The CN2 induction algorithm. *Machine Learning*, 3:261–283, 1989.

[28] R. Cleary. Extending grammar evolution with attribute grammars: An application to knapsack problems. Master's thesis, University of Limerick, Canberra, Australia, 2005.

[29] J.C. Cleaveland and R. C. Uzgalis. *Grammars for Programming Languages.* Elsevier Computer Science Library, New York, USA, 1977.

[30] C. A. Coello Coello and G.B. Lamont, editors. *Multi-Objective Algorithms for Attribute Selection in Data Mining.* World Scientific, 2004.

[31] C.A. Coello Coello, D.A. Van Veldhuizen, and G.B. Lamont. *Algorithms for Solving Multi-Objective Problems.* Kluwer Academic Publishers, New York, USA, 2002.

[32] W. W. Cohen. Efficient pruning methods for separate-and-conquer rule learning systems. In *Proc. of the* 13^{th} *Int. Joint Conf. on Artificial Intelligence (IJCAI-93)*, pages 988–994, France, 1993.

[33] W. W. Cohen. Fast effective rule induction. In A. Prieditis and S. Russell, editors, *Proc. of the* 12^{th} *Int. Conf. on Machine Learning (ICML-95)*, pages 115–123, Tahoe City, CA, jul 1995. Morgan Kaufmann.

[34] E. S. Correa, A. A. Freitas, and C. G. Johnson. A new discrete particle swarm algorithm applied to attribute selection in a bioinformatics data set. In M. Keijzer et al. (Eds.), editor, *Proc. of the Genetic and Evolutionary Computation Conference (GECCO-06)*, pages 35–42. ACM Press, July 2006.

[35] I. De Falco, A. Della Cioppa, and E. Tarantino. Discovering interesting classification rules with genetic programming. *Applied Soft Computing*, 1(4):257–269, May 2002.

[36] I. De Falco, E. Tarantino, A. Della Cioppa, and F. Fontanella. A novel grammar-based genetic programming approach to clustering. In *Proc. of the 2005 ACM Symposium on Applied Computing (SAC-05)*, pages 928–932, New York, NY, USA, 2005. ACM Press.

[37] B. de la Iglesia, Justin C. W. Debuse, and Victor J. Rayward-Smith. Discovering knowledge in commercial databases using modern heuristic techniques.

In *Proc. of the 2nd ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining (KDD-96)*, pages 44–49, 1996.

[38] K. Deb. *Multi-objective Optimization using Evolutionary Algorithms.* Wiley Interscience series in Systems and Optimization, Berlin, 2001.

[39] V. Dhar, D. Chou, and F. J. Provost. Discovering interesting patterns for investment decision making with GLOWER - a genetic learner overlaid with entropy reduction. *Data Mining and Knowledge Discovery*, 4(4):251–280, 2000.

[40] T. G. Dietterich. Ensemble methods in machine learning. In *Proc. of 1st Int. Workshop on Multiple Classifier Systems*, volume 1857 of *Lecture Notes in Computer Science*, pages 1–15, 2000.

[41] P. Domingos. Rule induction and instance-based learning: A unified approach. In *Proc. of the 14th Int. Joint Conf. on Artificial Intelligence (IJCAI-95)*, pages 1226–1232, 1995.

[42] M. Dong and R. Kothari. Look-ahead based fuzzy decision tree induction. *IEEE Transactions on Fuzzy Systems*, 9(3):461–468, June 2001.

[43] G. Dounias, H. Axer, B. Bjerregaard, D. Graf von Keyserlingk, J. Jantzen, and A. Tsakonas. Genetic programming for the generation of crisp and fuzzy rule bases in classification and diagnosis of medical data. In *Proc. of 1st Int. NAISO Congress on Neuro Fuzzy Technologies*, Havana, Cuba, 16-19 January 2002.

[44] S. Dŝeroski, B. Cestnik, and I. Petrovski. Using the m-estimate in rule induction. *Journal of Computing and Information Technology*, 1(1):37–46, 1993.

[45] S. Dŝeroski and B. Zenko. Is combining classifiers better than selecting the best one. In *Proc. of the 19th Int. Conf. on Machine Learning (ICML-02)*, pages 123–130, San Francisco, CA, USA, 2002. Morgan Kaufmann.

[46] A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computation.* Springer-Verlag, 2003.

[47] S. Esmeir and S. Markovitch. Lookahead-based algorithms for anytime induction of decision trees. In *Proc. of the 21th Int. Conf. on Machine Learning (ICML-04)*, 2004.

[48] T. Fawcett. Roc graphs: Notes and practical considerations for data mining researchers. Technical Report HPL-2003-4, HP Labs, 2003.

[49] U. M. Fayyad, G. Piatetsky-Shapiro, and P. Smyth. From data mining to knowledge discovery: an overview. In U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*. AAAI/MIT Press, 1996.

[50] P.A. Flach. The geometry of ROC space: understanding machine learning metrics through roc isometrics. In *Proc. 20th International Conference on Machine Learning (ICML-03)*, pages 194–201. AAAI Press, January 2003.

[51] J. J. Freeman. A linear representation for GP using context free grammars. In J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo, editors, *Proc. of the 3rd Annual Conf. on Genetic Programming (GP-98)*, pages 72–77. Morgan Kaufmann, 22-25 July 1998.

[52] A. A. Freitas. *Data Mining and Knowledge Discovery with Evolutionary Algorithms*. Springer-Verlag, 2002.

[53] A. A. Freitas. A critical review of multi-objective optimization in data mining: a position paper. *SIGKDD Explorations Newsletter*, 6(2):77–86, 2004.

[54] F. Friedrichs and C. Igel. Evolutionary tuning of multiple svm parameters. *Neurocomputing*, 64:107–117, 2005.

[55] G. Fung, S. Sandilya, and R. Bharat Rao. Rule extraction from linear support vector machines. In *Proc. of the 11th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining (KDD-05)*, pages 32–40, New York, NY, USA, 2005. ACM Press.

[56] J. Fürnkranz. Pruning algorithms for rule learning. *Machine Learning*, 27(2):139–171, 1997.

[57] J. Fürnkranz. Separate-and-conquer rule learning. *Artificial Intelligence Review*, 13(1):3–54, 1999.

[58] J. Fürnkranz. A pathology of bottom-up hill-climbing in inductive rule learning. In *Proc. of the 13th Int. Conf. on Algorithmic Learning Theory (ALT-02)*, pages 263–277, London, UK, 2002. Springer-Verlag.

[59] J. Fürnkranz and P. A. Flach. ROC 'n' rule learning: towards a better understanding of covering algorithms. *Machine Learning*, 58(1):39–77, 2005.

[60] J. Fürnkranz and P.A. Flach. An analysis of rule evaluation metrics. In *Proc. 20^{th} Int. Conf. on Machine Learning (ICML-03)*, pages 202–209. AAAI Press, 2003.

[61] J. Fürnkranz and G. Widmer. Incremental reduced error pruning. In *Proc. the 11_{th} Int. Conf. on Machine Learning (ICML-94)*, pages 70–77, New Brunswick, NJ, 1994.

[62] *Genetic Programming.* http://www.genetic-programming.org/, Visited in May, 2007.

[63] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, Reading, MA, 1989.

[64] G. Goos and J. Hartmanis, editors. *Attribute Grammar: Definition, Systems and Bibliography.* Lecture Notes in Computer Science, 1988.

[65] F. Gruau. On using syntactic constraints with genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 19, pages 377–394. MIT Press, Cambridge, MA, USA, 1996.

[66] D. J. Hand. *Construction and Assessment of Classification Rules.* Wiley, 1997.

[67] J. Hekanaho. Background knowledge in GA-based concept learning. In T. Fogarty and G. Venturini, editors, 13^{th} *Int. Conf. on Machine Learning (ICML-96)*, pages 234–242, 1996.

[68] P. G. Higgs and T. K. Attwood. *Bioinformatics and Molecular Evolution.* Blackwell, 2005.

[69] R. J. Hilderman and H. J. Hamilton. *Knowledge Discovery and Measures of Interest.* Kluwer Academic Publishers, Norwell, MA, USA, 2001.

[70] N. X. Hoai, R. I. McKay, and H. A. Abbass. Tree adjoining grammars, language bias, and genetic programming. In C. Ryan, T. Soule, M. Keijzer, E. Tsang, R. Poli, and E. Costa, editors, *Proc. of the 6^{th} European Conf.*

on Genetic Programming (EuroGP-03), volume 2610 of *Lecure Notes in Computer Science*, pages 335–344, Essex, 14-16 April 2003. Springer-Verlag.

[71] N. X. Hoai, R. I. McKay, and D. Essam. Some experimental results with tree adjunct grammar guided genetic programming. In J. A. Foster, E. Lutton, J. Miller, C. Ryan, and A. G. B. Tettamanzi, editors, *Proc. of the 5th European Conf. on Genetic Programming (EuroGP-02)*, volume 2278 of *LNCS*, pages 228–237, Kinsale, Ireland, 3-5 April 2002. Springer-Verlag.

[72] N. Holden and A. A. Freitas. A hybrid particle swarm/ant colony algorithm for the classification of hierarchical biological data. In P. Arabshahi and A. Martinoli, editors, *Proc. of the 2005 IEEE Swarm Intelligence Symposium (SIS-05)*, pages 100–107. IEEE, June 2005.

[73] N. Holden and A.A. Freitas. Hierarchical classification of G-protein-coupled receptors with a PSO/ACO algorithm. In *Proc. of the IEEE Swarm Intelligence Symposium (SIS-06)*, pages 77–84. IEEE Press, June 2006.

[74] H. Horner. A C++ class library for genetic programming: The Vienna University of Economics genetic programming kernel. citeseer, citeseer.nj.nec.com/horner96class.html, 29 May 1996.

[75] T. Howley and M. G. Madden. The genetic kernel support vector machine: Description and evaluation. *Artificial Intelligence Review*, 24(3-4):379–395, 2005.

[76] T. Hussain and R. Browse. Network generating attribute grammar encoding. In *Proc. of IEEE Int. Joint Conf. on Neural Networks*, pages 431–436, 1998.

[77] H. Jacobsson. Rule extraction from recurrent neural networks: A taxonomy and review. *Neural Computation*, 17:1223–1263, 2005.

[78] J.He and X. Yao. Towards an analytic framework for analyzing the computation time of evolutionary algorithms. *Artificial Intelligence*, 145(1-2):59–97, 2003.

[79] Y. Jin, editor. *Multi-Objective Machine Learning*. Springer, Berlin, 2006.

[80] K. A. De Jong, W. M. Spears, and D. F. Gordon. Using genetic algorithms for concept learning. *Machine Learning*, 13(2-3):161–188, 1993.

[81] A. K. Joshi and Y. Schabes. Tree-adjoining grammars. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3, pages 69–124. Springer, Berlin, New York, 1997.

[82] R. E. Keller and W. Banzhaf. Genetic programming using genotype-phenotype mapping from linear genomes into linear phenotypes. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors, *Proc. of the 1st Annual Conf. on Genetic Programming (GP-96)*, pages 116–122, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[83] R. D. King, K. E. Whelan, F. M. Jones, P. G. K. Reiser, C. H. Bryant, S. H. Muggleton, D. B. Kell, and S. G. Oliver. Functional genomic hypothesis generation and experimentation by a robot scientist. *Nature*, 427:247–252, 2004.

[84] J. R. Koza. *Genetic Programming: On the Programming of Computers by the means of natural selection.* The MIT Press, Massachusetts, 1992.

[85] J. R. Koza, M. A. Keane, M. J. Streeter, W. Mydlowec, J. Yu, and G. Lanza. *Genetic Programming IV: Routine Human-Competitive Machine Intelligence.* Kluwer Academic Publishers, 2003.

[86] W. B. Langdon, S. J. Barrett, and B. F. Buxton. Combining decision trees and neural networks for drug discovery. In *Proc. of the 5th European Conf. on Genetic Programming (EuroGP-02)*, pages 60–70, London, UK, 2002. Springer-Verlag.

[87] W. B. Langdon and B. F. Buxton. Genetic programming for combining classifiers. In L. Spector, E. D. Goodman, A. Wu, W. B. Langdon, H.M. Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M. H. Garzon, and E. Burke, editors, *Proc. of the Genetic and Evolutionary Computation Conference (GECCO-01)*, pages 66–73, San Francisco, California, USA, 7-11 July 2001. Morgan Kaufmann.

[88] N. Lavrac and S. Dzeroski. *Inductive Logic Programming: Techniques and Applications.* Routledge, New York, NY, 10001, 1993.

[89] N. Lavrac and S. Dzeroski, editors. *Relational Data Mining.* Springer-Verlag, Berlin, 2001.

[90] T. Lim, W. Loh, and Y. Shih. A comparison of prediction accuracy, complexity, and training time of thirty-three old and new classification algorithms. *Machine Learning*, 40(3):203–228, 2000.

[91] G. Lindner and R. Studer. Ast: Support for algorithm selection with a CBR approach. In *Proc. of the 3rd European Conf. on Principles and Practice of Knowledge Discovery in Databases (PKDD-99)*, pages 418–423, London, UK, 1999. Springer-Verlag.

[92] H. Liu and H. Motoda, editors. *Feature Selection for Knowledge Discovery and Data Mining.* Kluwer, 1998.

[93] J. J. Liu and J. Tin-Yau Kwok. An extended genetic rule induction algorithm. In *Proc. of the Congress on Evolutionary Computation (CEC-00)*, pages 458–463. IEEE Press, 6-9 2000.

[94] X. Llorà and J. M. Garrell. Prototype induction and attribute selection via evolutionary algorithms. *Intelligent Data Analysis*, 7(3):193–208, 2003.

[95] T. McConaghy and G. Gielen. Canonical form functions as a simple means for genetic programming to evolve human-interpretable functions. In *Proc. of the 8th annual Conf. on Genetic and Evolutionary Computation (GECCO-06)*, pages 855–862, New York, NY, USA, 2006. ACM Press.

[96] R. S. Michalski. On the quasi-minimal solution of the general covering problem. In *Proc. of the 5th Int. Symposium on Information Processing*, pages 125–128, Bled, Yugoslavia, 1969.

[97] R. S. Michalski, T. J. Carbonell, and T. M. Mitchell, editors. *Machine Learning: An Artificial Intelligence Approach.* TIOGA Publishing Co., Palo Alto, USA, 1983.

[98] D. Michie, D. J. Spiegelhalter, C. C. Taylor, and J. Campbell, editors. *Machine learning, neural and statistical classification.* Ellis Horwood, Upper Saddle River, NJ, USA, 1994.

[99] T. Mitchell. *Machine Learning.* Mc Graw Hill, 1997.

[100] D. J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230, 1995.

[101] D. P. Muni, N. R. Pal, and J. Das. A novel approach to design classifier using genetic programming. *IEEE Transactions on Evolutionary Computation*, 8(2):183–196, April 2004.

[102] S. K. Murthy and S. Salzberg. Lookahead and pathology in decision tree induction. In *Proc. of the Int. Joint Conf. on Artificial Intelligence (IJCAI-95)*, pages 1025–1033, 1995.

[103] P. Naur. Revised report on the algorithmic language algol-60. *Communications ACM*, 6(1):1–17, 1963.

[104] D. J. Newman, S. Hettich, C. L. Blake, and C. J. Merz. *UCI Repository of machine learning databases.* University of California, Irvine, http://www.ics.uci.edu/~mlearn/MLRepository.html, 1998.

[105] C. Nguyen and T. B. Ho. An imbalanced data rule learner. In *Proc. of the 9^{th} European Conf. on Principles and Practice of Knowledge Discovery in Databases (PKDD-05)*, pages 617–624, 2005.

[106] H. Nuñez, C. Angulo, and A. Catala. Rule extraction from support vector machines. In *Proc. of the European Symposium on Artificial Neural Networks (ESANN-02)*, pages 107–112, 2002.

[107] M. Ohsaki, S. Kitaguchi, K. Okamoto, H. Yokoi, and T. Yamaguchi. Evaluation of rule interestingness measures with a clinical dataset on hepatitis. In *Proc. of the 8^{th} European Conf. on Principles and Practice of Knowledge Discovery in Databases (PKDD-04)*, pages 362–373. Springer-Verlag New York, Inc., 2004.

[108] M. Oltean. Evolving evolutionary algorithms using linear genetic programming. *Evolutionary Computation*, 13(3):387–410, 2005.

[109] M. O'Neill, A. Brabazon, C. Ryan, and J. J. Collins. Evolving market index trading rules using grammatical evolution. In E. J. W. Boers, S. Cagnoni, J. Gottlieb, E. Hart, Pier L. Lanzi, G. R. Raidl, Robert E. Smith, and H. Tijink, editors, *Applications of Evolutionary Computing*, volume 2037 of *LNCS*, pages 343–352, Lake Como, Italy, 18-19 April 2001. Springer-Verlag.

[110] M. O'Neill and C. Ryan. *Grammatical Evolution : Evolutionary Automatic Programming in an Arbitrary Language.* Morgan Kaufmann, 2003.

[111] G. L. Pappa, A. J. Baines, and A. A. Freitas. Predicting post-synaptic activity in proteins with data mining. *Bioinformatics*, 21(Suppl. 2):ii19–ii25, September 2005.

[112] G. L. Pappa and A. A. Freitas. Towards a genetic programming algorithm for automatically evolving rule induction algorithms. In J. Fürnkranz, editor, *Proc. of the ECML/PKDD-04 Workshop on Advances in Inductive Learning*, pages 93–108, Pisa, 2004.

[113] G. L. Pappa and A. A. Freitas. Automatically evolving rule induction algorithms. In J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, editors, *Proc. of the 17th European Conf. on Machine Learning (ECML-06)*, volume 4212 of *Lecture Notes in Computer Science*, pages 341–352. Springer Berlin/Heidelberg, September 2006.

[114] G. L. Pappa, A. A. Freitas, and C. A. A. Kaestner. Multi-objective algorithms for attribute selection in data mining. In C. A. Coello Coello and G.B. Lamont, editors, *Applications of Multi-Objective Evolutionary Algorithms*, pages 603–626. World Scientific, 2004.

[115] N. R. Paterson and M. Livesey. Distinguishing genotype and phenotype in genetic programming. In J. R. Koza, editor, *Late Breaking Papers at the Genetic Programming 1996 Conference*, pages 141–150, Stanford University, CA, USA, 28–31 1996. Stanford Bookstore.

[116] N. R. Paterson and M. Livesey. Evolving caching algorithms in C by genetic programming. In J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, editors, *Proc. of the 2nd Annual Conference on Genetic Programming (GP-97)*, pages 262–267, Stanford University, CA, USA, 1997. Morgan Kaufmann.

[117] M. J. Pazzani. Knowledge discovery from data? *IEEE Intelligent Systems*, 15(2):10–13, 2000.

[118] Y. H. Peng, P. A. Flach, C. Soares, and P. Brazdil. Improved dataset characterization for meta-learning. In *Proc. of the 5th Int. Conf. on Discovery Science*, pages 141–152. Springer-Verlag, January 2002.

[119] B. Pfahringer, H. Bensusan, and C. Giraud-Carrier. Meta-learning by landmarking various learning algorithms. In *Proc.of the* 17^{th} *Int. Conf. on Machine Learning, (ICML-00)*, pages 743–750, San Francisco, California, 2000. Morgan Kaufmann.

[120] V. W. Porto, D. B. Fogel, and L. J. Fogel. Alternative neural network training methods. *IEEE Expert: Intelligent Systems and Their Applications*, 10(3):16–22, 1995.

[121] F. Provost, T. Fawcett, and R. Kohavi. The case against accuracy estimation for comparing induction algorithms. In *Proc. of the* 15^{th} *Int. Conf. on Machine Learning (ICML-98)*, pages 445–453, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[122] F. Provost and V. Kolluri. A survey of methods for scaling up inductive algorithms. *Data Mining Knowledge Discovery*, 3(2):131–169, 1999.

[123] J. R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.

[124] J. R. Quinlan. *C4.5: programs for machine learning.* Morgan Kaufmann, 1993.

[125] A. Ratle and M. Sebag. Genetic programming and domain knowledge: Beyond the limitations of grammar-guided machine discovery. In M. Schoenauer, K. Deb, G. Rudolph, X. Yao, E. Lutton, J. J. Merelo, and H. Schwefel, editors, *Proc. of the* 6^{th} *Int. Conf. on Parallel Problem Solving from Nature (PPSN)*, pages 211–220, Paris, France, 16-20 2000. Springer Verlag.

[126] D. Rivero, J. Dorado, J. R. Rabuñal, A. Pazos, and J. Pereira. Artificial neural network development by means of genetic programming with graph codification. *Transactions on Engineering, Computing and Technology*, 16:209–214, 2006.

[127] W. Romao, A. A. Freitas, and I. M. S. Gimenes. Discovering interesting knowledge from a science & technology database with a genetic algorithm. *Applied Soft Computing*, 4:121–137, 2004.

[128] A. Rozsypal and M. Kubat. Selecting representative examples and attributes by a genetic algorithm. *Intelligent Data Analysis*, 7(4):291–304, 2003.

[129] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 2002.

[130] C. Ryan, J. J. Collins, and M. O'Neill. Grammatical evolution: Evolving programs for an arbitrary language. In W. Banzhaf, R. Poli, M. Schoenauer, and T. C. Fogarty, editors, *Proc. of the 1st European Workshop on Genetic Programming*, volume 1391 of *Lecture Notes in Computer Science*, pages 83–95, Paris, 14-15 1998. Springer-Verlag.

[131] C. Schaffer. Overfitting avoidance as bias. *Machine Learning*, 10(2):153–178, 1993.

[132] B. Schölkopf and A. J. Smola. *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond.* The MIT Press, 2002.

[133] P. Smyth and R. M. Goodman. An information theoretic approach to rule induction from databases. *IEEE Transactions on Knowledge and Data Engineering*, 4(4):301–316, 1992.

[134] A. Suyama, N. Negishi, and T. Yamaguchi. CAMLET: A platform for automatic composition of inductive learning systems using ontologies. In *Proc. of the Pacific Rim Int. Conf. on Artificial Intelligence*, pages 205–215, 1998.

[135] H. Theron and I. Cloete. BEXA: A covering algorithm for learning propositional concept descriptions. *Machine Learning*, 24(1):5–40, 1996.

[136] A. B. Tickle, R. Andrews, M. Golea, and J. Diederich. The truth will come to light: directions and challenges in extracting knowledge embedded within trained artificial neural networks. *IEEE Transactions on Neural Networks*, 9(6):1057–1068, 1998.

[137] A. Tsakonas, G. Dounias, J. Jantzen, H. Axer, B. Bjerregaard, and D. G. von Keyserlingk. Evolving rule-based systems in two medical domains using genetic programming. *Artificial Intelligence in Medicine*, 32(3):195–216, 2004.

[138] S. Tsumoto. Clinical knowledge discovery in hospital information systems: Two case studies. In *Proc. of the 4th European Conf. on Principles of Data Mining and Knowledge Discovery (PKDD-00)*, pages 652–656, London, UK, 2000. Springer-Verlag.

[139] R. Vilalta and Y. Drissi. A perspective view and survey of meta-learning. *Artificial Intelligence Review*, 18(2):77–95, 2002.

[140] G. I. Webb and N. Brkic. Learning decision lists by prepending inferred rules. In *Proc. of the AI-93 Workshop on Machine Learning and Hybrid Systems*, pages 6–10. World Scientific, 1993.

[141] G. M. Weiss. Timeweaver: a genetic algorithm for identifying predictive patterns in sequences of events. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *Proc. of the Genetic and Evolutionary Computation Conference (GECCO-99)*, volume 1, pages 718–725, Orlando, Florida, USA, 13-17 1999. Morgan Kaufmann.

[142] S. M. Weiss and N. Indurkhya. Optimized rule induction. *IEEE Expert: Intelligent Systems and Their Applications*, 8(6):61–69, 1993.

[143] P. A. Whigham. Grammatically-based genetic programming. In J. P. Rosca, editor, *Proc. of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 33–41, Tahoe City, California, USA, 9 July 1995.

[144] P. A. Whigham. *Grammatical Bias for Evolutionary Learning.* PhD thesis, School of Computer Science, University College, University of New South Wales, Australian Defence Force Academy, Canberra, Australia, 14 October 1996.

[145] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations.* Morgan Kaufmann, 2^{nd} edition, 2005.

[146] D. Wolpert. Stacked generalization. *Neural Networks*, 5:241–259, 1992.

[147] M. L. Wong. An adaptive knowledge-acquisition system using generic genetic programming. *Expert Systems with Applications*, 15(1):47–58, 1998.

[148] M. L. Wong and K. S. Leung. *Data Mining Using Grammar-Based Genetic Programming and Applications.* Kluwer, Norwell, MA, USA, 2000.

[149] J. Woodward. GA or GP? That is not the question. In R. Sarker, R. Reynolds, H. Abbass, K. C. Tan, B. McKay, D. Essam, and T. Gedeon,

editors, *Proc. of the 2003 Congress on Evolutionary Computation (CEC-03)*, pages 1056–1063, Canberra, 2003. IEEE Press.

[150] X. Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447, 1999.

[151] J. Zhang. Selecting typical instances in instance-based learning. In *Proc. of the 9^{th} Int. Workshop on Machine learning (ML-92)*, pages 470–479, San Francisco, CA, USA, 1992. Morgan Kaufmann.

[152] J. Van Zyl and I. Cloete. Fuzzconri - a fuzzy conjunctive rule inducer. In J. Fürnkranz, editor, *Proc. of the ECML/PKDD-2004 Workshop on Advances in Inductive Learning*, pages 548–559, Pisa, 2004.

Appendix A

Computing the Size of the GGP Search Space

The size of the search space available for the GGP to search for new rule induction algorithms is determined by the GGP grammar. The actual number of solutions the grammar can produce is given by all the possible derivation trees built by applying a set of derivation steps starting from the *Start* symbol. The grammar presented in Section 4.2, and showed again in Table A.1 for convenience, does not have any recursive production rules, i.e. a non-terminal appearing in both the left-hand side and the right-hand side of a rule. Therefore, calculating the number of solutions the grammar can generate is not too complicated.

Alg. A.1 presents an algorithm which calculates how many derivations (different derivation trees/subtrees) a non-terminal can produce. Alg. A.1 works as follows. For each non-terminal of the grammar, it considers all the production rule in which that specific non-terminal NT appears in the left-hand side of the rule. The total number of derivations produced by a non-terminal is equivalent to the sum of the number of derivations for all of its production rules.

Note that, in Table A.1, the production rules generated by each non-terminal are written in a compact form, using the notation introduced in Section 3.3.1. Hence,

```
<condWhile>::= uncoveredNotEmpty |uncoveredGreater
               (10| 20| 90%| 95%| 97%| 99%) trainEx.
```

is equivalent to the 7 production rules:

```
<condWhile>::= uncoveredNotEmpty.
<condWhile>::= uncoveredGreater 10 trainEx.
```

```
<condWhile>::= uncoveredGreater 20 trainEx.
<condWhile>::= uncoveredGreater 90% trainEx.
<condWhile>::= uncoveredGreater 95% trainEx.
<condWhile>::= uncoveredGreater 97% trainEx.
<condWhile>::= uncoveredGreater 99% trainEx.
```

For each production rule, the number of derivations is given by the the product of the number of derivations of each of the symbols in the right-hand side of the production rule. The number of derivations of a terminal symbol is equal to 1, as a terminal cannot be expanded. The number of derivations of a non-terminal is recursively calculated. When a non-terminal is marked as optional, as in the last two symbols (enclosed by "[", "]") in the following production rule:

```
<CreateOneRule>::= <InitializeRule> <innerWhile> [<PrePruneRule>]
                   [<RuleStoppingCriterion>]
```

the algorithm can automatically calculate the number of derivations without expanding the compact rule representation to its 4 equivalent production rules, just by adding one to the number of derivations which can be created by each optional symbol (i.e., we have to consider all the derivations produced by the optional symbol plus the absence of the symbol).

Algorithm A.1: Derivations(NT)

```
derivNT = 0 ;
for each production rule PR generated by NT do
    derivPR = 1 ;
    for each symbol S in the right-hand side of PR do
        deriv = 0;
        if S is a terminal then
            deriv = 1 ;
        else if S is a non-terminal NT' then
            deriv = Derivations(NT');
        if S is optional then
            deriv = deriv + 1;
        derivPR = derivPR × deriv;
    derivNT = derivNT + derivPR;
return derivNT ;
```

Table A.2 lists the number of derivations for all the non-terminals of the grammar. In the first column of this table the NT number between brackets refers to the identification number of the corresponding NT symbol in the grammar of Table A.1. We started by calculating the number of possible derivations for the

Table A.1: The grammar used by the GGP

```
1-  <Start> ::= (<CreateRuleSet>|<CreateRuleList>) [<PostProcess>].
2-  <CreateRuleSet> ::= forEachClass <whileLoop> endFor
                          <RuleSetTest>.
3-  <CreateRuleList> ::= <whileLoop> <RuleListTest>.
4-  <whileLoop>::= while <condWhile> <CreateOneRule> endWhile.
5-  <condWhile>::= uncoveredNotEmpty |uncoveredGreater
                   (10| 20| 90%| 95%| 97%| 99%) trainEx.
6-  <RuleSetTest> ::= lsContent |confidenceLaplace.
7-  <RuleListTest>::= appendRule | prependRule.
8-  <CreateOneRule>::= <InitializeRule> <innerWhile> [<PrePruneRule>]
                        [<RuleStoppingCriterion>].
9-  <InitializeRule> ::= emptyRule| randomExample| typicalExample |
                          <MakeFirstRule>.
10- <MakeFirstRule> ::= NumCond1| NumCond2| NumCond3| NumCond4.
11- <innerWhile> ::= while (candNotEmpty| negNotCovered)
                      <FindRule> endWhile.
12- <FindRule> ::= <RefineRule> <EvaluateRule>
                    [<StoppingCriterion>] <SelectCandidateRules> |
                    <innerIf> <EvaluateRule>
                    [<StoppingCriterion>] <SelectCandidateRules>.
13- <innerIf> ::= if <condIf> then <RefineRule> else <RefineRule>.
14- <condIf> ::=  <condIfExamples> |<condIfRule>.
15- <condIfRule> ::= ruleSizeSmaller (2| 3| 5| 7).
16- <condIfExamples> ::= numCovExp ( >| <)(90%| 95%| 99%).
17- <RefineRule> ::= <AddCond>| <RemoveCond>.
18- <AddCond> ::= Add1| Add2.
19- <RemoveCond>::= Remove1| Remove2.
20- <EvaluateRule>::= confidence | Laplace| infoContent| infoGain.
21- <StoppingCriterion> ::= MinAccuracy (0.6| 0.7| 0.8)|
                             SignificanceTest (0.1| 0.05| 0.025| 0.01).
22- <SelectCandidateRules> ::= 1CR| 2CR| 3CR| 4CR| 5CR| 8CR| 10CR.
23- <PrePruneRule> ::= (1Cond| LastCond| FinalSeqCond) <EvaluateRule>.
24- <RuleStoppingCriterion> ::= accuracyStop (0.5| 0.6| 0.7).
25- <PostProcess> ::= RemoveRule EvaluateModel| <RemoveCondRule>.
26- <RemoveCondRule> ::= (1Cond| 2Cond| FinalSeq) <EvaluateRule>.
```

non-terminals which generate only terminal symbols. The non-terminal *AddCond*, for example, generates only the terminals *Add1* or *Add2*, and consequently its number of derivations is 2.

We then calculated the number of derivations of the non-terminals which have production rules that generate the non-terminals for which the corresponding number of derivations is known. For instance, the non-terminal *InitializeRule* generates 4 production rules. The first three are terminals (so each of them has the number of derivations equals to 1), while the fourth is the non-terminal *MakeFirstRule*. *MakeFirstRule*'s number of derivations is known as 4, and therefore the number of derivations for *InitializeRule* is 7.

The number of derivations for all non-terminals but two was calculated following Alg. A.1. The two exceptions were the non-terminals *innerIf* and *CreateOneRule*. This was because, as explained in Section 4.4, some constraints were imposed to the population initialization process. These constraints refer to the non-terminals *innerIf* and *CreateOneRule*, and reduce the number of derivations they can produce.

In the case of the non-terminal *innerIf*, in order to avoid introns, situations where both the *if* and *else* part of the conditional statement performed the same operation were avoided. As observed in Table A.1, NT13, *innerIf* combines two instances of the non-terminal *RefineRule*, whose number of derivations is 4. Following Alg. A.1, the number of derivations of *innerIf* would be 160 (the condition part of the *if* statement produces 10 derivations $\times$ 16 derivations generated by the body of the conditional statement, given by the combination of 2 instances of the non-terminal *RefineRule*). However, as combining *Add1* with *Add1*, *Add2* with *Add2*, *Remove1* with *Remove1* and *Remove2* with *Remove2* is not allowed, this number of combinations of 2 instances of *RefineRule* is reduced to 12, and the number of derivations of *innerIf* is reduced to 120.

The calculation of the number of derivations of the non-terminal *CreateOneRule* (NT8 in Table A.1) also has to account for some constraints. As explained in Section 4.4, it does not make sense to use *AddCond* with the production rules *randomExample* and *typicalExample* (generated by *InitializeRule*) nor *RemoveCond* with *emptyRule* (generated by *InitializeRule*), and *NumCond1* and *NumCond2* (both generated by *MakeFirstRule*). From now on, for the sake of simplicity, we refer to the terminals *randomExample* and *typicalExample* as "specific initialization terminals", and to the terminals *emptyRule*, *NumCond1* and *NumCond2* as "general initialization terminals".

Table A.2: Number of derivations generated by the non-terminals of the grammar

Non-terminal	Number of Derivations
<condWhile> (NT 5)	7
<RuleSetTest> (NT 6)	2
<RuleListTest> (NT 7)	2
<MakeFirstRule> (NT 10)	4
<condIfRule> (NT 15)	4
<condIfExamples> (NT 16)	6
<AddCond> (NT 18)	2
<RemoveCond> (NT 19)	2
<EvaluateRule> (NT 20)	4
<StoppingCriterion> (NT 21)	7
<SelectCandidateRules> (NT 22)	7
<RuleStoppingCriterion> (NT 24)	3
<InitializeRule> (NT 9)	3+4 = 7
<condIf> (NT 14)	6+4 = 10
<RefineRule> (NT 17)	2×2 = 4
<PrePruneRule> (NT 23)	3×4 = 12
<RemoveCondRule> (NT 26)	3×4 = 12
<innerIf> (NT 13)	10×(16-4) = 120
<PostProcess> (NT 25)	1+12 = 13
<FindRule> (NT 12)	(4+120)×4×(7+1)×7 = 27,776
<innerWhile> (NT 11)	2×27,776 = 55,552
<CreateOneRule> (NT 8)	12,999,168
<whileLoop> (NT 4)	7×12,999,168 = 90,994,176
<CreateRuleList> (NT 3)	90,994,176×2 = 181,988,352
<CreateRuleSet> (NT 2)	90,994,176×2 = 181,988,352
<Start> (NT 1)	2×181,988,352×14 = 5,095,683,856

If we had to calculate the number of derivations of *CreateOneRule* regardless of these exceptions, its value would be 7×55,552×13×4 = 20,220,928. However, out of the 7 ways of initializing a rule, only 2 are not subject to exceptions, namely the terminals *NumCond3* and *NumCond4* (generated by *MakeFirstRule*). For the other 5, we actually have to recalculate the number of derivations of *CreateOneRule* (see Eq. (A.1)) and, consequently, the number of derivations of *innerWhile*, *FindRule* (see Eq. (A.3)), *RefineRule* and *innerIf* (see Eq. (A.2)).

In Eq. (A.1) we show the calculation of the number of derivations of *CreateOneRule)*, denoted *d(CreateRule)*. The third line in Eq. (A.1) separates the number of derivations for the 2 terminals with no constraints from the other 5. For these 5 terminals, the number of derivations of *FindRule* has to be recalculated,

as shown in Eq. (A.3). In Eq. (A.3), the number of derivations of *RefineRule* (d(RefRule') is 2 instead of 4, because "specific initialization terminals" can only be used with *AddCond* and "general initialization terminals" can only be used with *RemoveCond*, each of them with number of derivations equals to 2.

We then recalculate the number of derivations of *innerIf* in Eq. (A.2). In this case, *RefineRule* has also to be recalculated, and this has to take many constraints into account at once. Recall that there is a first constraint which states that from the 16 possible combinations of 2 instances of *RefineRule* in the *if* and *else* parts of a conditional statement, only 12 are valid (to avoid introns). For these 12 *RefineRule* instances, individuals presenting a *condIfRule* and a "specific initialization terminal" cannot use a *RemoveCond* non-terminal in the *if* part of the conditional statement. At the same time, individuals having a "general initialization terminal" cannot be used with the non-terminal *AddCond* in the *else* part of the conditional statement. In summary, only half of the possible 12 combinations (the ones with *Add* in the *if* part or *Remove* in the *else* part of the conditional statement) are valid. Hence, the number of combination for both instances of *RefineRule* is equal to 6 for *condIfRule*.

$$
\begin{aligned}
d(CreateRule) &= d(InitRule) \times d(inWhile) \times d(PrePrunRule) \times d(RuleStopCrit) \\
d(CreateRule) &= d(InitRule) \times d(inWhile) \times 13 \times 4 \\
d(CreateRule) &= 2 \times (55,552 \times 52) + 5 \times (2 \times d(FindRule)) \times 52 \\
d(CreateRule) &= 5,777,408 + 5 \times (2 \times d(FindRule)) \times 52 \\
d(CreateRule) &= 5,777,408 + 5 \times (2 \times 13,888) \times 52 \\
d(CreateRule) &= 7,221,760
\end{aligned}
\tag{A.1}
$$

$$
\begin{aligned}
d(innerIf) &= d(condIf) \times d(RefRule) \times d(RefRule) \\
d(innerIf) &= (d(condIfExamples) \times d(RefRule) \times d(RefRule)) \\
&\quad + (d(condIfRule) \times d(RefRule) \times d(RefRule)) \\
d(innerIf) &= (6 \times 6) + (4 \times 6) = 60
\end{aligned}
\tag{A.2}
$$

$$
\begin{aligned}
d(FindRule) &= d(RefRule) \times d(EvalRule) \times d(StopCrit) \times d(SelectCand) \\
&\quad + d(innerIf) \times d(EvalRule) \times d(StopCrit) \times d(SelectCand) \\
d(FindRule) &= (d(RefRule') + d(innerIf)) \times 224 \\
d(FindRule) &= (2 + 60) \times 224 = 13,888
\end{aligned} \tag{A.3}
$$

In the case of *condIfExamples*, individuals with a "specific initialization terminal" cannot use a *RemoveCond* non-terminal in the *if* part of the conditional statement if they have a "<" operator in the condition. When using a ">" operator, *RemoveCond* cannot be in the *else* part of the conditional statement. In contrast, individuals having a "general initialization terminal" cannot use the non-terminal *AddCond* in the *else* part of the conditional statement if the condition used the operator ">". When using the operator "<", the same is true for the *if* part of the conditional statement. Once again, the number of combinations for both instances of *RefineRule* is equal to 6.

As showed in the last line of Table A.2, the total size of the grammar search space given by the *Start* symbol – i.e., the total number of distinct rule induction algorithms that can be derived from the grammar – is 5,095,683,856.