Strongly Typed Evolutionary Programming

Claire Julia Kennedy

A thesis submitted to the University of Bristol in accordance with the requirements for the degree of Doctor of Philosophy in the Faculty of Engineering, Department of Computer Science.

December 1999

Abstract

As the potential of applying machine learning techniques to perplexing problems is realised, increasingly complex problems are being tackled, requiring intricate explanations to be induced. Escher is a functional logic language whose higher-order constructs allow arbitrarily complex observations to be captured and highly expressive generalisations to be conveyed.

The work presented in this thesis alleviates the challenging problem of identifying an underlying structure normally required to search the resulting hypothesis space efficiently. This is achieved through STEPS, an evolutionary based system that allows the vast space of highly expressive Escher programs to be explored. STEPS provides a natural upgrade of the evolution of concept descriptions to the higher-order level.

In particular STEPS uses the individual-as-terms approach to knowledge representation where all the information provided by an example is localised as a single closed term so that examples of arbitrary complexity can be treated in a uniform manner. STEPS also supports λ-abstractions as arguments to higher-order functions thus enabling the invention of new functions not contained in the original alphabet. Finally, STEPS provides a number of specialised genetic operators for the design of specific concept learning strategies.

STEPS has been successfully applied to a number of complex real world problems, including the international PTE2 challenge. This problem involves the prediction of the Carcinogenic activity of a test set of 30 chemical compounds. The results produced by STEPS rank joint second if the hypothesis must be interpretable and joint first if interpretability is sacrificed for increased accuracy.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or institution of learning.

C. Kennedy

Acknowledgements

Firstly I would like to thank my supervisor Christophe Giraud-Carrier for his support and guidance throughout this work. His constant enthusiasm has been an inspiration to me.

I would like to thank the members of the Machine Learning Group, both past and present, with whom I have had many interesting discussions, in particular Peter Flach, Tony Bowers, John Lloyd, Rene McKinney-Romero, Torbjorn Dahl, Nicholas Lachiche, Kerstin Eder and Simon Butterworth. I would also like to thank Tony for his implementation of the Escher interpreter, without which, this work would not have been possible. I am grateful to EPSRC who funded my work and travel for the last three years on grant GR/L21884.

I would like to thank the many friends that I have made during my time here in the department especially Ann for always knowing the right thing to say, Katerina for being Greek and lovely, Mark for being sarcastic and lovely and Angus for being lovely. I am especially grateful to Mark and Henk for their constructive feedback after proof reading this thesis.

I would also like to thank my friends who have both supported and encouraged me over the last few years. Especially Suzanne and Ben (for putting a roof over my head) Sutt, Claire-Louise, Graeme and Simon and the girls from home: Tina, Lorraine, Alexie, Sarah and Emma, my shopping buddies.

I am indebted to my family Hilary, Jim, Paul, John, and Nanny who have supported me both emotionally and financially for more than just the last few years. You never know, I may even get a “real” job now. Last but not least I would like to say a special thankyou to Andrew for his encouragement and support and for putting up with me when the going got tough.

Dedication

To

Mum, Dad, Nanny,

John, Paul,

and Andrew.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The field of Machine Learning is concerned with enriching machines with the ability to learn, i.e., machine learning is the ability for a machine to

> *improve its performance at a particular task in response to experience without being explicitly programmed.*

Machine Learning was born from the fields of Artificial Intelligence and Cognitive Science, inspired by the potential impact it would have on our every day uses of computers, and by a desire to understand the complex procedures by which humans and other *intelligent* beings learn. Machine Learning can be viewed as *automatic programming*. It can offer a practical alternative to the solution of a wide range of difficult or ambiguous problems such as the autonomous driving of a vehicle, the prediction of the structure of a protein, and the design of an analogue circuit, to name but a few. The surge of interest into the field in recent years is due to its successful contribution to significant industrial applications. In particular, machine learning algorithms have been used to extract useful knowledge from commercial databases in the context of *data mining*.

There are several ways in which a computer can learn (Langley 1996). This thesis focuses on inductive learning, which is perhaps the most studied and therefore most understood form of machine learning.

1.1 Inductive Machine Learning

Inductive machine learning involves inducing *generalisations* from *specific examples.* For example, after carrying out a survey of birds in Britain, one could generalise that *all birds have feathers and fly.* The induced generalisations are often expressed as classification rules. The experience provided to an inductive learner typically takes the form of *observations* from the real world. These observations are often represented by attribute value pairs, where attributes correspond to the properties of the entity observed and their connected values indicate the value that the particular entity takes for that property. These observations are presented to a learner in conjunction with labels corresponding to their classification. Examples labelled in this manner are called a *training set* as they are used to *train* the learner.

The learner is required to extract some critical features, or rules, from these observations, compressing the information that they contain into a generalisation, or *hypothesis*, so that correct inferences can be made on unseen observations. The learner achieves this by identifying a hypothesis that correctly classifies, or *fits* the labelled information with which it has been provided. One trivial method to achieve this would be to construct a hypothesis as the disjunction of all examples presented to the learner. However this hypothesis can not be considered a generalisation, as it is unable to make inferences on unseen examples. This resultant hypothesis is not desirable so some additional constraints are necessary in order that its *value* is ranked below that of alternative hypotheses that also fit the data. An inductive learner achieves this by employing some form of *inductive bias* (Mitchell 1982). Representational biases force a learning algorithm to make generalisations by restricting the language in which the hypotheses are expressed. For example the language is often restricted to express hypotheses in the form of conjunctions of attribute value-pairs. An alternative form of bias, search bias, allows the inductive algorithm to examine some more preferable hypotheses earlier than others. For example the learner may prefer to examine simpler hypotheses first. However, the solution(s) to the learning problem to be solved are contained in a subset of the space of hypotheses. Therefore when restricting this space through biases, the effect of these biases on the solution space should be considered. If the effect of the biases is to restrict the solution space or to delay access to part of it, then this may reduce the chances of finding an optimal solution.

In order to find an appropriate hypothesis the learner searches through all the hypotheses, called the *Hypothesis Space* to find one that fits the training examples and that will, due to the use of inductive bias, generalise to unseen examples. Therefore, much of machine learning research involves discovering strategies in which this search can be made more efficient. The hypothesis space is defined by the language in which the generalisations or hypotheses are expressed. Often the language involves some combination of the attribute values that describe an observation. However for some problems this is insufficient and a more sophisticated language is required. For example it is possible to represent relationships between the attributes by using a subset of first-order logic. However extra sophistication incurs a cost: with increased expressiveness of the hypothesis language comes increased complexity (in size and search), of the hypothesis space.

All machine learning algorithms perform a search through a hypothesis space, but their fundamental differences lie with the particular language they choose to express the possible generalisations and hence the manner in which they choose to explore the hypothesis space created by this chosen language. Often the particular language chosen can provide a useful underlying structure that organises the search of the hypothesis space in such a way that its huge expanse can be explored without explicitly enumerating all of its members. One such means for doing this is to arrange the hypotheses according to a *general-to-specific* ordering (Mitchell 1977). A hypothesis can be considered to be *more general than* another hypothesis in the same hypothesis space if it covers at least the same examples as the second hypothesis. The reverse is true of the *more special than* relationship. This arrangement of the hypotheses provides a partial order on the hypothesis space exacting a lattice structure upon it. The algorithm traverses the lattice by using operators that specialise or generalise the current hypothesis as necessary. In the first-order setting, where hypotheses are represented in a first-order language such as Prolog, the hypothesis space has an order imposed on it by θ - subsumption (Nienhuys-Cheng and de Wolf 1997). Operators based on this ordering, such as Inverse Resolution (Muggleton and Buntine 1988) and Inverse Entailment (Muggleton 1995) are used.

An alternative view of a learning problem can be obtained by mapping the chosen hypothesis space to its fitness landscape. The fitness landscape expresses the fitness or value of each hypothesis in the hypothesis space at solving the current learning problem. Areas of high fitness or value are expressed by peaks in the landscape, while areas of low fitness or value are expressed by troughs. Algorithms can be designed to move around the fitness landscape in order to find areas of high fitness in the search for an optimal solution (Goldberg 1989, Koza 1992). When a search of the fitness landscape is carried out, there is no need to impose an ordering on the hypothesis space as the hypothesis space is not explicitly being searched and is therefore no longer useful.

Inductive machine learning has been successfully applied to a number of applications. (e.g. see (Mitchell n.d.)). A particular subset of these applications fall under the umbrella of *Concept Learning* where the inductive learner is provided with positive and negative examples of a particular concept, and asked to induce a general description of the concept. Concept learning can be viewed as the task of generating classification rules that determine whether or not a particular example belongs to the concept / class or not. The particular applications considered in this thesis are concept learning problems. When designing a learning algorithm to solve a particular class of problems, in this case concept learning, it is common to maximise the algorithm's performance by *optimising* the algorithm to the particular features of that class of problems. However the *No Free Lunch* theorem states that by optimising an algorithm to a particular class of problems, there exist other classes of problems on which the algorithm performs suboptimally (Wolpert and Macready 1995). Therefore, as the particular algorithm described in this thesis is optimised to concept learning, its performance on other classes of problems may be inferior to that of other learning algorithms biased towards those classes of problems.

1.2 Thesis Statement

As the potential of applying machine learning techniques to perplexing problems is realised, increasingly complex problems are being tackled, requiring intricate explanations to be induced. However, the complexity of the knowledge expressed by an induced hypothesis is bounded by the constraints of the hypothesis language that it employs.

Escher is a functional logic language whose higher-order constructs allow arbitrarily complex observations to be captured and highly expressive generalisations to be conveyed. However attempts at identifying an underlying structure that allows the organisation of the hypothesis space, in such a way that it may be searched efficiently, have so far proved challenging. This thesis proposes to alleviate this problem by extending an evolutionary algorithm, from a family of techniques well known for their powerful search ability, to allow the vast space of highly expressive Escher programs to be explored.

1.3 Outline of Thesis

This thesis is organised as follows:

Chapter 2 presents a discussion of knowledge representation in the context of machine learning. Various forms of knowledge representation commonly used in machine learning are examined in terms of the contrasting hypothesis languages which they employ. The *individuals-as-terms* approach to knowledge representation used by the Strongly Typed Evolutionary Programming System, STEPS, as described in this thesis is presented.

Chapter 3 first presents the evolutionary technique, Genetic Programming, and examines its advantages and shortcomings when applied to particular problem types. This is followed by a description of the enhancements necessary to integrate the Strongly Typed Genetic Programming approach with the Escher individuals-as-terms setting resulting in the design of the evolutionary learning system STEPS.

In chapter 4 the application of STEPS to the problem of concept learning is discussed. Some specialised genetic operators used by STEPS during the learning of concept descriptions are defined. The benefits of the use of these operators are examined through their application to a number of simple concept learning problems. These specialised genetic operators allow the implementation of specific *strategies* for learning that may be employed by STEPS. The strategies are evaluated using the simple concept learning problems. Some of the work presented in this chapter has appeared in (Kennedy 1998, Kennedy and Giraud-Carrier 1999*a*, Kennedy and Giraud-Carrier 1999*b*).

Chapter 5 identifies that both automatically defined functions in GP and reformulation approaches to predicate invention in ILP are motivated by a need to compress the program code that results from their corresponding learning processes. In this chapter the connection between reformulation, predicate invention and automatically defined functions is established. A special case of automatically defined functions, λ-abstractions in the context of strongly typed genetic programming, is discussed and the particular approach to predicate invention adopted by STEPS is introduced.

Chapter 6 describes the application of STEPS to two real world concept learning problems from the molecular biology domain. The inherent structure of the graphical representation of molecules is naturally captured by the individuals-as-terms representation used in STEPS. These problems illustrate the capacity of STEPS to tackle more substantial, complicated problems. Some of the work presented in this chapter has appeared in (Kennedy *et al.* 1999, Kennedy 1999).

The final chapter concludes the thesis. The work presented in the thesis and its main contributions are summarised. This is then followed by some suggestions for future work.

Chapter 2

Knowledge Representation

2.1 Introduction

Knowledge is fundamental to all forms of intelligence, and in particular to learning. Indeed, although there are quite different approaches to learning, they can all be viewed as manipulation, transformation and discovery of knowledge. For a machine to perform these tasks, knowledge must be encoded in a suitable way. The encoding of knowledge so that it can be manipulated by a computer is the subject of *Knowledge Representation*.

Knowledge representation has long been a prevalent area of research in artificial intelligence. This chapter is restricted to discussing knowledge representation in the context of machine learning as this is the focus of this thesis. Various forms of knowledge representation commonly used in machine learning are examined. Then, the *individuals-as-terms* approach to knowledge representation used by STEPS as described in this thesis is presented.

2.2 Knowledge Representation in Machine Learning

One of the main uses of machine learning is to aid in the solution of hard problems that are difficult to solve with traditional programming techniques. The knowledge and experience of the problem must be encoded into a form that can be *understood* by the computer. This information and experience is manipulated by the learning algorithm and a solution is obtained. This solution may then be translated so that the knowledge that it encapsulates is more accessible to a human user.

The learning (or generalisation) problem, can be summarised as follows (Mitchell 1982):

> "*Given*: (1) A language in which to describe instances.
> (2) A language in which to describe generalisations.
> (3) A matching predicate that matches generalisations to instances.
> (4) A set of positive and negative training instances of a target generalisation to be learned.
>
> *Determine*: Generalisations within the provided language that are consistent with the presented training instances (i.e., plausible descriptions of the target generalisation)."

The language in which the instances are described is called the *Observation Language* here. The representation provided to the learning algorithm should be flexible enough to abstract the essential features of observations for the particular problem. A more complex problem will therefore require a more flexible observation language so that its features may be adequately captured.

The solution derived from the information expressed in the observation language (i.e., generalisation) is expressed in the *Hypothesis Language.* For the solution to be of most use, the hypothesis language should be able to make the information that it contains available to the user. Therefore the solution ideally should be expressed in terms that are natural to the user providing a transparent elucidation to the problem.

In addition to a transparent solution, it is often desirable that the solution be found in an optimal manner. In this case the hypothesis language is required to be of a form that allows the search space to be explored efficiently.

The requirement for efficient, transparent knowledge representation strategies leads to a conflicting situation. Transparency is associated with expressiveness since most learning algorithm users are human. However expressive representations are typically inversely related to efficient representation. A trade-off between these two characteristics is therefore required.

The method of knowledge representation chosen to express a problem dictates what can and can not be learnt in terms of its expressivity, the speed of learning in terms of its efficiency, and the transparency or understandability of the knowledge that is learnt in terms of its explanatory power. However these are not the only aspects of learning which it can affect. Obviously the type of knowledge representation chosen can affect the types of problems that can be learnt, and in this respect it can affect the generality of a learner in terms of the number of different types of problems that it may tackle. Particular forms of knowledge representation can allow a learner to obtain solutions from very sparse or very large datasets. Different problems require different learner characteristics, hence there exists a number of methods for representing knowledge in machine learning, some of which we will examine in the next section.

2.3 Knowledge Representation Approaches Commonly used in Machine Learning

2.3.1 Attribute Value Observation Language

Most current inductive machine learning systems make use of an attribute value observation language, where the observations are represented as attribute value pairs, or symbolic feature vectors. This relatively simple observation language is perfectly adequate for representing problems that do not contain any complex relationships.

In order to illustrate this form of knowledge representation, consider the toy problem of characterising the weather in which it is suitable to play tennis (Mitchell 1997). The weather on a particular day is described by four features (or attributes): Outlook, which can take on the values Sunny, Overcast or Rain; Temperature, which can take on the values Hot, Mild, Cool; Humidity which can take on the values High or Normal; and Wind, which can take on the values Strong or Weak.

The training examples consist of a pairs composed of feature vectors and their corresponding target values for the function *playTennis*:

((Outlook, Temperature, Humidity, Wind), (yes or no))

((sunny, mild, high, strong), yes)

((rain, hot, high, weak), yes)

((sunny, hot, high, weak), no)

⋮

This problem will be used as a running example throughout this section.

There are a number of different approaches to learning that may be used to identify the weather conditions in which it is suitable to play tennis. The primary characteristics that distinguish them are the manner in which this information is expressed, i.e., the hypothesis language employed, and the algorithm used to search the hypothesis space. A number of such hypothesis languages and their associated learner will be presented in the following sections.

2.3.1.1 Genetic Algorithms

Genetic algorithms (Goldberg 1989) encode potential solutions as a fixed length bit string called a *chromosome*, i.e. the hypothesis language consists of bit strings. During learning a population of these chromosomes is manipulated by genetic operators over successive generations. Genetic algorithms are particularly successful at solving optimisation problems, but they can also be applied to classification problems with complex hypotheses.

A potential solution to the "playTennis" problem, described above, evolved by a genetic algorithm can be represented by a bit string of length eleven: one bit for each value that each feature can take (set to 1 if the feature has that particular value) and one bit to indicate whether the weather is suitable (by being set to 1) or not suitable (by being set to 0) for playing tennis. For example the following bit string:

Outlook	Temperature	Humidity	Wind	playTennis
011	111	01	10	0

would represent the rule

> *if the Outlook is Overcast or Raining, and the Humidity is Normal, and the Wind is Strong, then I don't want to play tennis.*

Note that all the bits for Temperature have their value set to one, which is equivalent to describing the fact that *I don't care* what value Temperature has, i.e., it is a disjunction of all possible values for Temperature.

The bit string representation of genetic algorithms gives rise to a relatively efficient learning algorithm, and there exists a body of theory providing theoretical evidence to complement the empirical proof of the robustness of genetic algorithms (Goldberg 1989). However the fixed length chromosome representation is not very flexible (see section 2.3.1.2).

2.3.1.2 Genetic Programming

Genetic Programming (Koza 1992) is an extension of genetic algorithms where potential solutions, and therefore the hypothesis language, take the form of lisp programs represented as parse trees. For example the parse tree for the program $x^2 + ((y * 3) + z)$ is presented in Figure 2.1. In order to create these program trees an alphabet for the problem is specified. The alphabet can be split into functions (the set of possible internal nodes of the tree) and terminals (the set of possible leaf nodes of the tree). Traditional genetic programming requires closure of the function set. This means that every function in the function set must be able to handle any value or data type that is returned by any function in the function set (including itself). This effectively means that program trees are made up of functions that all return the same type. Genetic Programming has been found to work well on a large number of numerical problems (Koza 1992, Koza 1994), in particular regression tasks.

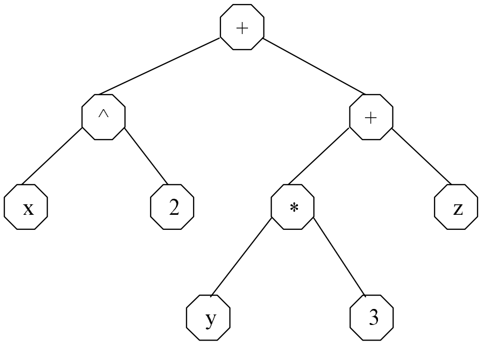


Figure 2.1: The simple program $x^2 + ((y * 3) + z)$ in Parse tree form

We illustrate the knowledge representation used by genetic programming on the play tennis problem. Koza uses this problem to compare genetic programming with decision trees in (Koza 1991). In order for genetic programming to be applied to this simple problem, the four features or attributes that constitute the weather are converted into functions taking the possible values that they may take on as their arguments and returning the appropriate argument for the current example. E.g. if the current example has a sunny outlook, then the function *Outlook* will return its first argument *Sunny*. This gives us the following function set:

$$F = \{Outlook, Temperature, Humidity, Wind\}$$

The terminal set for this problem is:

$$T = \{0, 1\}$$

where 0 is returned if the weather conditions are not suitable for playing tennis and 1 is returned if the reverse is true. In other words it is implicitly assumed that all features have the same type in order to enforce the closure requirement.

In (Koza 1991) the following result (expressed as a LISP S-expression) is obtained for this problem:

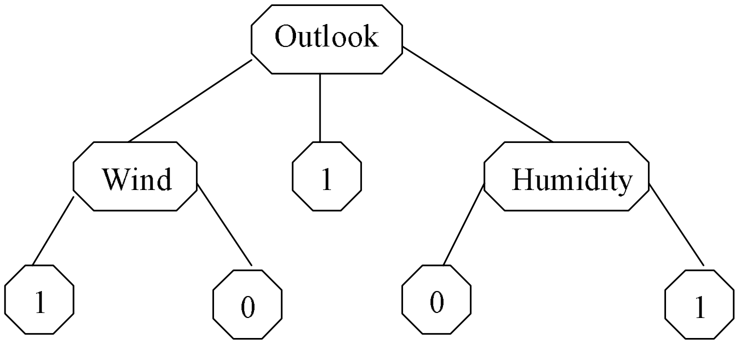


Figure 2.2: Koza's solution to the playTennis problem in parse tree form

```
(Outlook (Wind (1 0) 1 (Hum 0 1)))
```

It is expressed in parse tree form in figure 2.2. The expression's meaning in English is "*I want to play tennis if the outlook is sunny and the humidity is normal, or if the outlook is overcast, or if the outlook is rain and it is not windy*"

The flexibility of the representation employed by genetic programming allows for a general purpose learning system. The basic algorithm and representation stay the same, and just the fitness function and the genetic operators have to be tailored to the new domain. However the assumption of closure of the alphabet can lead to artificially formed solutions. Compare the solution to the play tennis problem presented above with the solution obtained by STEPS in section 4.4.1. The artefactual encoding of attributes as Boolean functions, in order for the simplifying requirement of closure to be enforced, allows the attribute *Outlook* to take the value *strong* returned by the attribute $Wind$ as its argument. This is not natural. In order to alleviate this problem a type system has been introduced to standard genetic programming in Strongly Typed Genetic Programming (Montana 1995). In addition the necessity to repeatedly interpret the lisp program trees can lead to inefficiency.

2.3.1.3 Feed Forward Neural Networks

Feed Forward Neural Networks consist of a network of nodes and their weighted connections. The input to the network consists of real or integer or boolean valued (or a mixture) feature vectors. The feature vectors are passed through the network over a number of iterations in order for the network to adjust its weights and learn the desired function. Neural Networks are effective at solving complex problems that require a complex discriminant mathematical function in order to obtain the answer (Rumelhart and McClelland 1986).

To solve the playTennis problem from above, then the neural network could be provided with boolean valued feature vectors representing the weather as above.

These feature vectors to be provided to a neural network are similar in form to the potential solutions evolved by the genetic algorithm. They differ in the number of 'bits' used to indicate the class of weather (i.e. suitable or not suitable). Two bits are used in this case so that the two output nodes can compete when a new instance is presented (the use of one bit to indicate the class is also possible of course).

For purposes of illustration the Stuttgart Neural Network Simulator (SNNS) (SNN 1996) was used, with the back propagation algorithm to train a neural network on the playing tennis problem.

The configuration of the network consisted of ten input nodes, two hidden nodes (the number of hidden nodes was chosen to be small so that the network actually generalised the patterns rather than just stored them all), two output nodes, and a bias node connected to the hidden and output nodes. The activation function in each node was the sigmoid function $\frac{1}{1+e^{-x}}$, where x is the sum of the nodes weighted inputs. The resultant trained network is in Figure 2.3.

The numbers on the connections between the nodes are the final weights obtained after training the network. These weights represent the knowledge that the network has gained through its training. Therefore the weights in conjunction with the constraints of the network architecture can be viewed as the Hypothesis Language for this particular learning technique.

Neural Networks can be very accurate at approximating discriminant functions and they

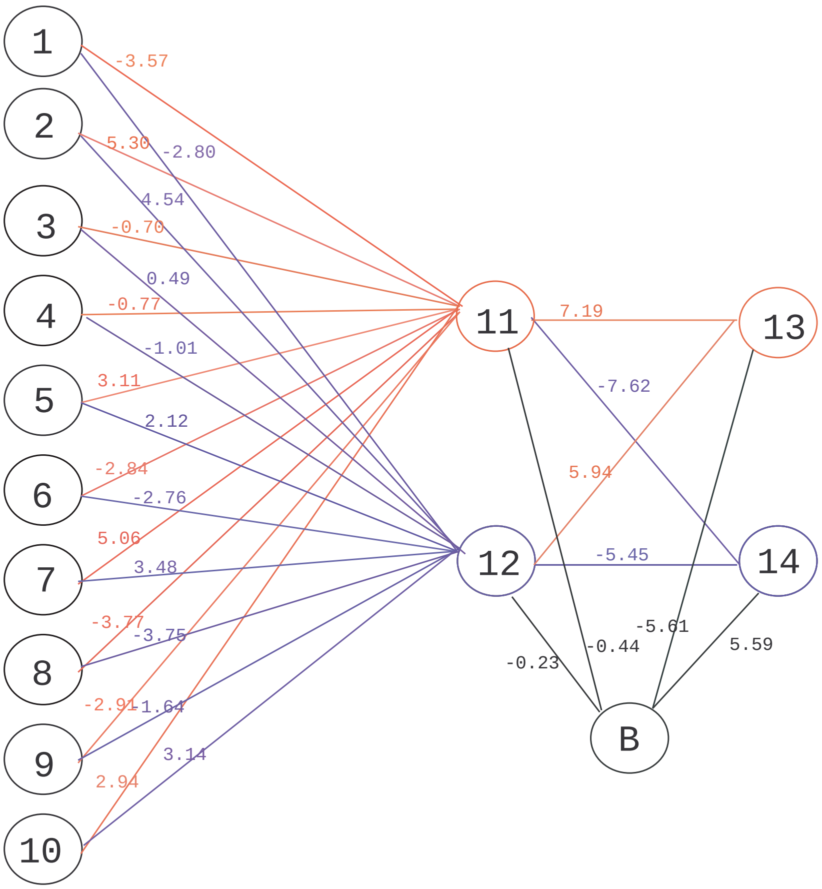


Figure 2.3: Neural network trained on the playTennis problem

have been found to be robust against noisy data making them particularly suitable for real world problems such as analysing sensor data. However the resulting knowledge is embedded in the weights of the network and can be viewed as a matrix of real numbers, which is not particularly intuitive to a human user, it is a so called black box or opaque approach.

2.3.1.4 Instance Based Learning

Instance based learning algorithms take a prototypical approach to learning (Aha *et al.* 1991). The stored internal representation of the examples for a prototyping algorithm is the examples themselves, and so the hypothesis language is equivalent to the observation language. A classification is obtained by presenting the new example to the algorithm which uses a distance metric to identify which of the stored examples is closest. The classification returned is a function of the categories of the closest examples (e.g. the majority category). Instance based learning algorithms are particularly good at solving classification problems where it is the classification and not a general classification rule that is of interest.

The k-Nearest Neighbour algorithm is a widely used instance based learning technique. Consider the playTennis problem again. Here all the available examples are stored in memory. The examples are feature vectors which are of the form $(x, f(x))$:

```
((1,0,0,0,0,1,1,0,0,1),1)
((0,0,1,0,1,0,1,0,0,1),1)
((1,0,0,0,1,0,1,0,1,0),1)
((0,1,0,0,1,0,0,1,1,0),1)
((1,0,0,1,0,0,0,1,0,1),0)
((0,0,1,0,1,0,0,1,1,0),0)
((1,0,0,1,0,0,0,1,1,0),0)
((0,0,1,0,0,1,1,0,1,0),0)
```

Once this has been achieved the learning phase is complete. The algorithm's work begins once a new individual is presented for classification where the distance metric calculation is required. For the nearest neighbour approach the distance metric is typically a simple Euclidean distance function of the form (Mitchell 1997):

$$d(x_i, x_j) = \sqrt{\sum_{r=1}^{n} (a_r(x_i) - a_r(x_j))^2}$$

where $a_r(x)$ denotes the value of the rth attribute of instance x. For example, the distance between

```
(0,1,0,1,0,0,0,1,0,1)
```

and

```
(1,0,0,0,0,1,1,0,0,1)
```

is 6. The closest neighbour of $(0,1,0,1,0,0,0,1,0,1)$ is in fact the example

```
((1,0,0,1,0,0,0,1,0,1),0)
```

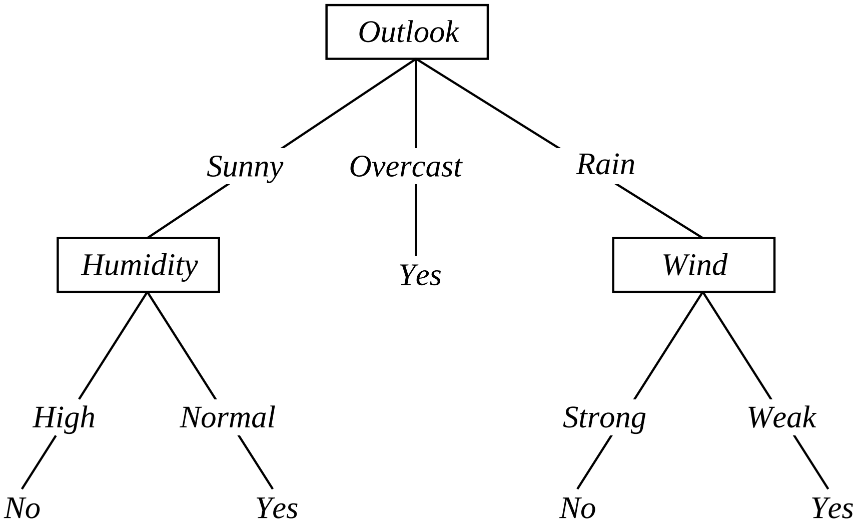
from which it has a distance of 2. Hence the new example's classification would be 0

Instance based learning approaches are a very efficient method for classification, especially if an efficient lookup method to identify the closest match is employed. They can also handle noisy data and are suitable for incremental learning problems. However the distance metric used to identify candidate matches can be difficult to devise (e.g., for non-numeric or mixed attribute values) (Giraud-Carrier and Martinez 1995, Wilson and Martinez 1995). The training phase can be a drain on memory as all examples have to be stored.

2.3.1.5 Decision Trees

Decision tree algorithms such as ID3 (Quinlan 1986) and its successor C4.5 (Quinlan 1993) represent their learned knowledge in a decision tree structure, i.e., their hypothesis language is a set of decision trees. The internal nodes of these structures are tests on the attributes or features of the examples which are provided to the algorithm. Each of these attribute tests has a descendent branch for each value that the particular attribute may take. The leaf nodes of the decision tree are classification labels that may be applied to an example. A classification for a new example is obtained by starting at the root and then traversing through the tree applying the attribute tests to the example and following the appropriate branches, until a leaf node is met and its value returned as the classification.

The decision tree obtained for the playTennis example problem using ID3look like (taken from (Mitchell 1997)):



Decision trees can also be represented as if-then-else rules in propositional logic. The equivalent expression for the above decision tree, in rule format, is:

$IF\ Outlook = overcast\ THEN\ yes\ \vee$

$IF\ Outlook = rain\ \wedge\ wind = weak\ THEN\ yes\ \vee$

$IF\ outlook = sunny\ \wedge\ humidity = normal\ THEN\ yes$

$ELSE\ no$

Decision tree algorithms are capable of representing transparent general solutions that have proved successful on a wide variety of problems.

2.3.1.6 Rule Induction

Representing learned knowledge as a set of *if-then-else* rules can help to convey the knowledge to a human as they are generally human intelligible. One method for learning if-then-else rules is to translate them from an induced decision tree. An alternative to this method is to directly learn a set of rules from the training data, e.g. CN2 (Clark and Niblett 1989). Therefore the hypothesis language is a set of rules.

Generally a *Sequential Covering* algorithm is used to learn a single rule, consisting of a conjunction of attribute tests, at a time. Each rule is required to be highly accurate, but only cover some of the training examples. Once a single rule has been induced the examples that it covers are removed from the set of training data and the process is repeated until all the examples in the training set have been covered. The set of these rules are disjunctively combined to form the induced hypothesis.

The set of rules learnt from the playTennis example would be of the form as those derived from the decision tree presented in section 2.3.1.5.

2.3.2 First Order Logic Observation Language

The First order logic representation is an extension of the propositional logic form of knowledge representation used for the Attribute value setting. Examples are presented as logical expressions and any available information about the problem to be solved is presented as background knowledge from which a hypothesis expressed in first order (typically Horn clause) logic is induced. First order logic allows the expression of relationships between attributes.

2.3.2.1 Inductive Logic Programming

Inductive Logic Programming (Muggleton 1992*b*, Muggleton and DeRaedt 1994, Lavrač and Džeroski 1994) is the area in machine learning which uses Horn clause logic as its knowledge representation strategy. The systems typically use the logic programming language Prolog to implement their knowledge representation facilities, i.e., Prolog is used for both the observation language and the hypothesis language.

Although propositional logic is entirely sufficient to represent the play tennis problem, the problem can also be expressed in Prolog and solved by an inductive logic programming system. In this case the examples are presented as ground facts:

```
playTennis(ex1).
playTennis(ex2).
playTennis(ex3).
         ⋮
```

and additional information is presented as background information:

```
    outlook(ex1,Sunny).
    temperature(ex1,Mild).
    humidity(ex1,High).
    wind(ex1,Strong).

    outlook(ex2,Rain).
    temperature(ex2,Hot).
    humidity(ex2,High).
    wind(ex2,Weak).

          ⋮
```

An induced hypothesis (in Prolog) would then be of the form:

```
    playtennis(Ex):-
       outlook(Ex,overcast).

    playtennis(Ex):-
       outlook(Ex,rain),
       wind(Ex,weak).

    playtennis(Ex):-
       outlook(Ex,sunny),
       humidity(Ex,normal).
```

First order logic provides an expressive knowledge representation strategy that allows the use of background knowledge to augment the examples during learning. It can describe interactions or relationships between components (e.g. `above(A,B)` specifies the relationship that `A` is above `B`). An example of a First-order problem expressed in Prolog is now presented. Here each example is a train, where a train consists of a number of cars and where a car has a number of features including its shape, its length, the number of wheels it has, the type of roof it has, and its load. The aim of the problem is to predict from the description of the train, whether it is travelling eastbound or westbound. As before each example is a ground fact:

```
eastbound(t1)
```

and the features that describe the train would typically be presented as background knowledge:

```
car(t1,c1). rectangle(c1). short(c1). none(c1). two_wheels(c1).
   load(c1,l1). circle(l1). one_load(l1).

car(t1,c2). rectangle(c2). long(c2). none(c2). three_wheels(c2).
   load(c2,l2). hexagon(l2). one_load(l2).

car(t1,c3). rectangle(c3). short(c3). peaked(c3). two_wheels(c2).
   load(c3,l3). triangle(l3). one_load(l3).

car(t1,c4). rectangle(c4). long(c4). none(c4). two_wheels(c4).
   load(c4,l4). rect(l4). three_loads(l4).
```

where `car(t1,c1).` represents the fact that train `t1` contains a car named `c1`, and `rectangle(c1).` represents the fact that car `c1` is of rectangular shape.

An induced hypothesis (in Prolog) would then be of the form:

```
eastbound(T):- car(T,C), short(C).
```

i.e., a train is travelling eastbound if it contains a short car.

As mentioned earlier, first order representations are typically implemented in Prolog which relies on the closed world assumption. The systems using this form of representation tend to flatten out all the structure in the example information to make their algorithms more efficient (as in the example above). This flattening of the information loses the inherent structure forfeiting some of the complex semantic meaning. It also becomes ambiguous as to what information should actually constitute an individual example and which related information should be placed as background information. However this is an artefact of the algorithms and not of the representation language chosen. The trains example could equally be expressed in a more structured manner in Prolog. In this case each example train could be represented as a list of cars as follows:

```
eastbound([c(rectangle,short,none,2,l(circle,1)),
           c(rectangle,long,none,3,l(hexagon,1)),
           c(rectangle,short,peaked,2,l(triangle,1)),
           c(rectangle,long,none,2,l(rectangle,3))]).
```

and an equivalent induced hypothesis could be of the form:

```
eastbound(T):-member(C,T),arg(2,C,short).
```

In this thesis, the above structured representation will be taken further using a higher-order, strongly typed langauge, as described in the next section.

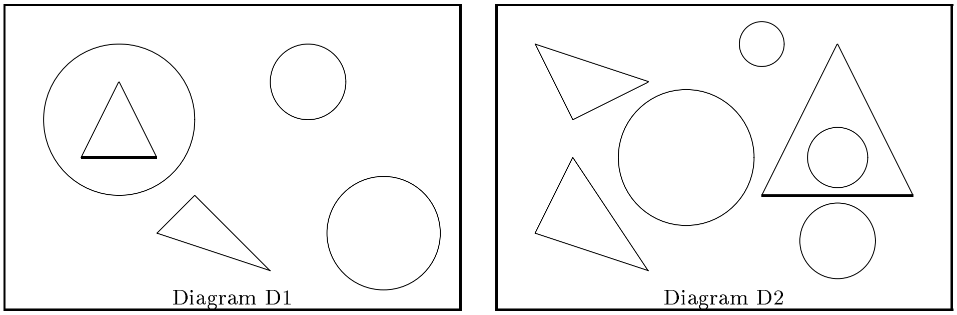
2.4 Individuals-As-Terms Representation

The choice of knowledge representation facilities depends on the nature of the problem to be solved. If the value of a classification is sufficient then a subsymbolic learner with opaque knowledge representation is perfectly adequate. However if a comprehensible reasoning is imperative to the solution then a symbolic learner must be used and the choice of knowledge representation becomes more critical.

Historically the propositional attribute value learning approach has been sufficient for the solution to the learning problem. A symbolic hypothesis is formed by restricting the possible values that the attributes may take. The search space is constrained by only allowing the attributes to accept their appropriate values. In this way the attributes can be thought of as types. The simplicity of this representation allows efficient learners to be implemented, however the representation can be restricted as the explicit representation of relations is not available, making some applications (e.g. the molecular problems in chapter 6) too rich to be easily represented.

The first-order representation, typically implemented in Prolog, is a more expressive representation which allows the expression of relations and the use of background knowledge. However this representation is restricted to Horn clauses where everything (including functions) is represented as predicates. In addition Prolog is essentially typeless so the inherent search space constraints available in the propositional setting are lost during this upgrade. This leads to a situation where ad-hoc substitutes for the type system (e.g. Mode declarations (Muggleton 1995) and determinacy (Quinlan 1990)) are implemented to regain efficiency.

Most recently, higher-order representations, based on the Escher language, have been proposed (Bowers *et al.* 1997, Flach *et al.* 1998). Escher is a strongly typed programming language that integrates the best features of functional and logic programming languages allowing the use of both functions and predicates so that items more naturally represented as a function do not have to be converted into a predicate. Its syntax is based on the functional programming language Haskell (Lloyd 1999) and it contains higher-order constructs such as sets (a useful facility for knowledge representation). This higher-order representation further improves the potential for symbolic inductive learning as the enhanced expressiveness increases applicability, allowing the awkward formulation of some of the problems in the other logic based representations to be alleviated by a more natural representation. This in turn allows for improved comprehensibility elucidating a greater understanding of the information extracted from the data. In addition to this as we will see in the remainder of this section, the Escher representation provides a way to "... deal with functions and predicates in a unified way." (Flach *et al.* 1998) and provides a seamless link between the extension from attribute value language through to the more complex higher orders of logic. For more information on the Escher programming language the reader is referred to (Lloyd 1999, Lloyd 1995).



```
data Shape = Circle | Triangle | Inside(Shape, Shape);
type Diagram = {(Shape, Int)};
d1 = {(Circle,2), (Triangle,1), (Inside(Triangle, Circle),1)};
d2 = {(Circle,3), (Triangle,2), (Inside(Circle, Triangle),1)};
```

Figure 2.4: Example diagrams and their corresponding Escher representations

2.4.1 Examples

When using the attribute value language as the observation language, all examples take the form of a tuple of constants. When using first order representation each example is flattened into a fact explicitly naming it and a relational database containing all its properties, hence all structure originally inherent in an example is collapsed out. The *individuals-as-terms* approach to knowledge representation involves representing each individual example as a single closed term. Therefore simple attribute value examples remain as tuples of constants, however more complex examples will have correspondingly more complex terms, where the inherent, structured information contained may be captured in a localised manner. Note that in the individuals-as-terms context, an individual refers to an 'individual' example, this conflicts with the evolutionary definition where an individual refers to a potential solution. Typically in this thesis, the word 'individual' will refer to the first meaning of the word.

This approach to representing examples can be compared to the *learning from interpretations* approach of Inductive Logic Programming (DeRaedt and Dehaspe 1997). However the learning from interpretations approach has no localised internal structure.

Figure 2.4 illustrates the closed term representation used here on two diagrams taken from Problem 47 of (Bongard 1970). Note the keyword `data` indicates a declaration of a type, and the keyword `type` introduces a type synonym. Each diagram contains a number of shapes, where each shape can be a circle, a triangle or a shape inside another shape. A diagram in the Escher closed term representation is a set of pairs, each consisting of a shape and a number indicating the number of times the shape appears in the diagram. Thus, d1 contains 2 circles, 1 triangle and 1 triangle inside a circle.

2.4.2 Background Knowledge

The individuals-as-terms representation also puts the issue of background knowledge into perspective. The availability of background knowledge is often stated as one of the major advances of the first-order representation over the propositional setting. However the background knowledge appears just to be an artifact of the flattening approach to representation taken by most ILP practitioners. When examples are represented as closed terms the properties of that example remain a part of its representation so there is no need for the majority of the background knowledge provided by the corresponding ILP representations. In fact in (Flach *et al.* 1998) it was found that the background knowledge disappeared altogether when the problems were translated from the Prolog flattened representation into the Escher individuals-as-terms representation. Even the functions associated with the data types used to represent the examples (see section 3.4) are deemed not to be part of the background knowledge as they are inherently attached to the type. It is only "...auxiliary functions that are neither useful outside the context of a particular learning task, nor easily to be found by the learner itself" (Flach *et al.* 1998) that are actually considered to be background knowledge. Flach *et al* therefore consider the advantage of the higher orders of logic over the propositional setting not to be the possibility to encode background knowledge, but the extra expressiveness allowed by the wider range of available types.

2.4.3 Induction

In order to induce descriptions, it is necessary to extract parts of the individual closed terms so as to make inferences about them. This is accomplished by selector functions, which "pull" individual components out of terms. Each structure (e.g. list, set) that is used in a term has it own set of associated selector functions. Figure 2.5 shows the selector functions for tuples, lists and sets.

```
v = Tuple Type
        proji(v)
v = List Type
        exists \v2 -> v2 'elem' v
        length(v)
v = Set Type
        exists \v2 -> v2 'in' v
        card(v)
```

Figure 2.5: Some data structures and their associated selector functions

For example, the number of occurrences of some shape in d1 above is obtained with

```
exists \x -> x 'in' d1 && proj2(x)
```

Once the components of the data structures have been extracted, conditions can be made on them or they can be compared to values or other data types. For example, the following expression tests whether the number of circles in d1 is equal to 2.

```
exists \x -> x 'in' d1 && (proj1(x) == Circle && proj2(x) == 2)
```

An algorithm, called Enumerate, has been designed to generate automatically the appropriate selector function associated with a set of types (Bowers *et al.* 1999). The *Enumerate* algorithm was originally designed to provide a set of conditions, based on the structure of the closed term examples, to be used by a decision tree algorithm. The Enumerate algorithm will be discussed in more detail in Section 3.4.1.

2.5 Summary

In this chapter we have discussed the key aspects of knowledge representation in the context of Machine Learning. The conflict between the desirable properties of knowledge representation, namely expressiveness and efficiency, has been stated in addition to some of the characteristics of learning that knowledge representation typically affects. This was

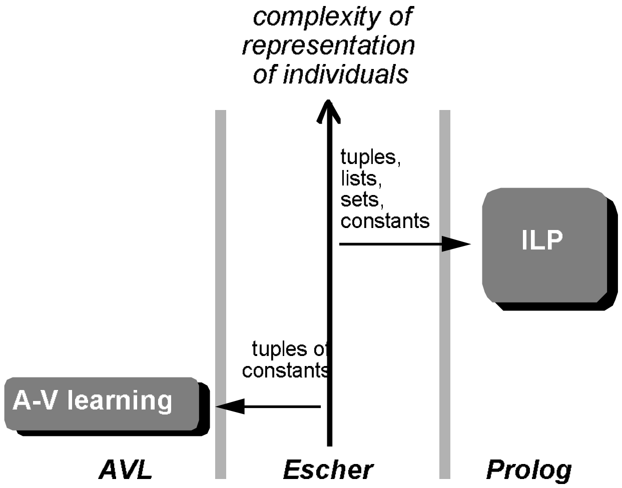

complexity of representation of individuals
tuples, lists, sets, constants
ILP
tuples of constants
A-V learning
AVL
Escher
Prolog

Figure 2.6: The relation between attribute-value learning and ILP is best understood through a strongly typed language such as Escher. One of the main differences lies in the complexity of the terms that are used to represent individuals. From (Flach *et al.* 1998)

subsequently followed by an examination of various forms of knowledge representation commonly used in machine learning.

The individuals-as-terms approach as used in this thesis has been presented and compared to the two widely used logical representations, the propositional and first order settings. The individuals-as-terms approach to knowledge representation localises all the information provided by an example as a single closed term to give a compact and self-contained description of each example. The illumination of the natural progression of knowledge representation in the attribute value language through to the higher orders of logic can be viewed diagrammatically in figure 2.6 taken from (Flach *et al.* 1998). Individuals are of arbitrary structure, tuples of constants for the attribute value case, tuples lists and constants in the ILP case, up through to the higher order case where in addition to the preceding types, more complex types such as sets occur.

Chapter 3

The STEPS Learning System

3.1 Introduction

As was seen in chapter 2 the *individuals-as-terms* representation presented in (Flach *et al.* 1998) provides a unified approach to learning in increasing orders of logic. The Escher language provides a highly expressive knowledge representation facility for both the *Observation* and *Hypothesis* languages. However, this highly expressive representation causes an explosion of the search space that is compounded by the challenging problem of identifying consistent extensions to the effective search techniques of the propositional and first-order settings.

Techniques from the field of Evolutionary Computing offer a powerful solution to such complex search problems. Evolutionary computing is comprised of a family of learning algorithms based on the Darwinian theory of evolution. The generic form of an evolutionary algorithm is found in Figure 3.1. This algorithm can be tailored to a particular problem domain through the adaptation of its fitness function, genetic operators and the form of the individuals that are evolved.

One increasingly popular technique belonging to the family of evolutionary computing is Genetic Programming (Koza 1992). Genetic Programming was introduced in section 2.3.1.2

```
Generation = 0
Generate initial population
Evaluate initial population according to fitness function
While termination criterion not met
    Use fitness evaluation to select subpopulation
    Generate new population from subpopulation using genetic operators
    Evaluate new population according to fitness function
    Increment Generation
Return designated result
```

Figure 3.1: A generic evolutionary algorithm

As the aim of learning in this research is to obtain a solution in the form of an expressive Escher program, the GP approach is a suitable general purpose technique to adapt. Escher is a strongly typed language, therefore an evolutionary paradigm that incorporates type information is necessary so that only type-correct programs are generated during learning. Traditional program tree based evolutionary paradigms, such as GP, assume the closure of all functions in the body of the program trees (Koza 1992). This means that every function in the function set must be able to take any value or data type that can be returned by any other function in the function set. While this characteristic simplifies the genetic operators, it limits the applicability of the learning technique and can lead to artificially formed solutions and prevents GP from being applied directly in the Escher individuals-as-terms setting. Strongly Typed Genetic Programming (Montana 1995), an extension to the GP approach, lifts the closure requirement by implementing mechanisms that allow only type correct programs to be considered. STEPS is a Strongly Typed Evolutionary Programming System that further extends this latter approach to allow the vast space of highly expressive Escher concept descriptions to be explored efficiently. The original Evolutionary Programming systems used mutation operators to evolve finite state machines in order to solve sequence prediction problems (Fogel *et al.* 1966). However, in recent years Evolutionary Programming has expanded to operate on vectors of real numbers, program trees and dynamic bayesian networks in order to address different problems (Chellapilla *et al.* 1998, Angeline 1997, Chellapilla 1997, Tucker and Liu 1999). One of the main characteristics that distinguishes Evolutionary Programming from other evolutionary paradigms, such as genetic algorithms and genetic programming, is its renouncement of recombination as the main genetic operator.

In this chapter the evolutionary technique of Genetic Programming is presented and its advantages and shortcomings examined in detail. This is followed by an exposition of Strongly Typed Genetic Programming. A description of the further enhancements necessary to integrate the Strongly Typed Genetic Programming approach with the Escher individuals-as-terms setting is then provided.

3.2 Genetic Programming

3.2.1 Representation

Genetic Programming like Genetic Algorithms, uses an evolutionary algorithm to search for a solution to the problem at hand, but differs from Genetic Algorithms in the representation of the individuals that constitute a population. Genetic Programming as the name suggests automatically evolves the solution in the form of a computer program. Users of the technique often call the approach *automatic programming* due to the nature of the representation. During the evolutionary process, the programs are represented by parse trees, so that the genetic operators may be easily applied. The parse tree representation of the program is a hierarchical representation where the arguments of a function are represented as its descendant nodes. Therefore a function of arity n in a parse tree will have n child nodes. These arguments may be constants, variables or other functions. The parse tree for the program $x^2 + ((y * 3) + z)$ is presented in figure 3.2. The parse tree representation can vary in its depth and size which can change dynamically during the evolutionary process to give a more flexible representation than the fixed length bit strings of the Genetic Algorithm.

3.2.2 Initialisation

Before a problem may be solved using the Genetic Programming technique, the alphabet from which the program trees are composed must be specified. The alphabet for a particular problem can be split into two sets. The *Function set* (F) which is made up of all the functions (i.e., the internal nodes of a tree) and the *Terminal set* (T) which is made

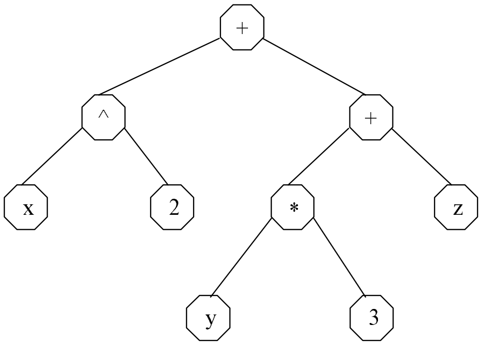


Figure 3.2: The simple program $x^2 + ((y * 3) + z)$ in Parse tree form

up of all the constants and variables for a particular problem (i.e., the leaf nodes with no descendants in a tree). It is often not clear which functions and terminals should be included in F and T in order for a problem to be solved. Their determination is clearly a very important point in the evolutionary process as an insufficient alphabet leads to a situation where the solution is impossible to find, and an over zealous alphabet will lead to inefficiency as the search space becomes unnecessarily large. Since it is almost always impossible to ascertain the exact alphabet for a problem, the user generally tries to make an intelligent guess.

In order to create individuals to be placed into the initial population, symbols from the alphabet are randomly selected until a complete tree is formed. The size and shape of a tree can be restricted by specifying whether a function or terminal should be randomly selected at a particular point in the recursive tree growing algorithm. In addition, in order that each tree be syntactically correct, the *closure* constraint is imposed on the alphabet. An alphabet is closed if every function in F can accept any value or data type in T that may be returned by any function in F. Some non-closed alphabets can be forced to become closed by defining protected versions of the functions that cause the violation. For example an alphabet consisting of the standard arithmetic functions (e.g., $+, -, \times, \div$) operating on real values can become closed by defining a protected division function which returns 1 if its divisor is 0.

Individuals are created and placed in the initial population in this manner until the pre-specified quota (i.e., maximum population size) is met. Once this has been achieved, an evolutionary algorithm very similar to that used by the Genetic Algorithm is employed to search through the space of programs that may be combined from the alphabet until either a solution is found or a preset maximum number of iterations has occurred.

3.2.3 Fitness Evaluation

The next step in an evolutionary algorithm is to evaluate the performance of the newly created individuals to identify how well they perform on the problem to be solved (see Figure 3.1). This fitness evaluation forms the basis of Darwinian *Natural Selection*. The performance of each individual is usually calculated as a function of its performance over a number of training examples called *fitness cases*. The design of this function is crucial as it defines how capable the algorithm is of solving the problem. The function may take into account a number of factors (e.g., program tree size, complexity, etc.) so that the solution may be tailored to the users requirements.

3.2.4 Selection

Once the individuals have been evaluated, they may be selected based on their fitness values, with a bias towards higher fitness values, to become parents for the individuals of the next generation. There are a number of possible selection schemes that allow individuals with a higher fitness value to have a higher probability of being selected, including the more popular *Tournament* and *Fitness Proportionate* selections.

The tournament selection scheme is one of the simpler fitness based selection schemes. A group of n individuals (where $n \geq 2$) are randomly selected from the population. The individual with the highest fitness value out of the group members is selected as a parent and the remaining individuals discarded. This process is repeated by sampling the population, with replacement.

The fitness proportionate technique for selection is the most popular selection technique used in Genetic Algorithms. It is also very popular among users of Genetic Programming. The probability that an individual in the population will be selected is proportional to its own fitness and is given by the ratio of its fitness to the fitness of the other members of the population. Formally the probability p that an individual ind_i will be selected from a population with n members is calculated as

$$p(ind_i) = \frac{fitness(ind_i)}{\sum_{j=1}^{n} fitness(ind_j)}$$

Fitness proportionate selection is often called *Roulette Wheel* selection, as the selection technique can be viewed as spinning a roulette wheel. In this analogy, each individual is allocated a portion of the wheel proportionate to its probability of selection. Therefore fitter individuals will have correspondingly larger portions of the wheel increasing the likelihood of the roulette landing in its portion.

3.2.5 Genetic Operators

As with genetic algorithms, genetic operators are used to navigate the search space. The standard operators of Genetic Algorithms have to be adapted to operate on the program tree structure of the individuals. The genetic operators are biologically inspired by the process of genetic recombination. The main genetic operators used by Genetic Programming algorithms are *Reproduction, Crossover* and *Mutation.* These operators are applied to parent individuals in order to produce new individuals with pre-specified probabilities.

The Reproduction operator is used to ensure that the fitter members of the population survive and populate the remaining generations. A single parent is required and a direct copy of it is made and is placed into the new generation. In the experiments presented in (Koza 1992) typically 10% of a new population is created through the reproduction operator.

The mutation operator is used to produce a random change in the genetic material of an individual in order to increase diversity in the population. Like the reproduction operator only one parent is required. A node is randomly selected from the parent program tree. This node is known as the *mutation point.* The subtree rooted at the mutation point is removed and replaced with a randomly grown subtree to create a new individual that is placed in the new population. The mutation operator is rarely used in standard Genetic

Programming as the increase in diversity that it provides is considered not to be necessary due to the larger population sizes used. The mutation operator is omitted from all the problems presented in (Koza 1992).

The principal genetic operator used by Genetic Programming algorithms is the crossover operator. This operator is a *sexual* operator as two parents are selected and their genetic material is combined to produce two new individuals. Crossover first randomly selects a node from the first parent program tree with uniform probability. Then a node is selected in the same manner from the second parent program tree. The subtrees rooted at these so called *crossover points* are then swapped (or crossed) over to produce two new individuals to be placed into the new generation as depicted in Figure 3.3. Note that the closure property ensures that arbitrary exchanges are possible and result in syntactically correct program trees. In (Koza 1992) it is recommended that the remaining 90% of a new population is generated using the crossover operator (the first 10% you may recall is generated through reproduction).

3.2.6 Example

In order to illustrate the process of applying Genetic Programming to a problem, the popular benchmark problem of obtaining a navigation strategy for an *artificial ant* will be used (Koza 1992). The problem involves navigating the artificial ant along a trail whose path contains food. There are many particular trails of varying difficulty that may be navigated, the trail that will be used in this example is the *Santa Fe* trail containing eighty-nine items of food. The trail goes through a thirty-two by thirty-two grid, and in order to untrivialise the problem, there are gaps in the trail where there is no food. During its attempted navigation of the trail, the ant only has the ability to see into the adjacent cell in the grid in the direction in which it is facing. The aim of the problem is to obtain a navigation strategy that allows the ant to find and eat all the food along the trail in a reasonable length of time.

The primary concern for this problem is for the ant to eat all of the food along the trail. Therefore the learning alphabet needs some functions that allow the ant to move in the

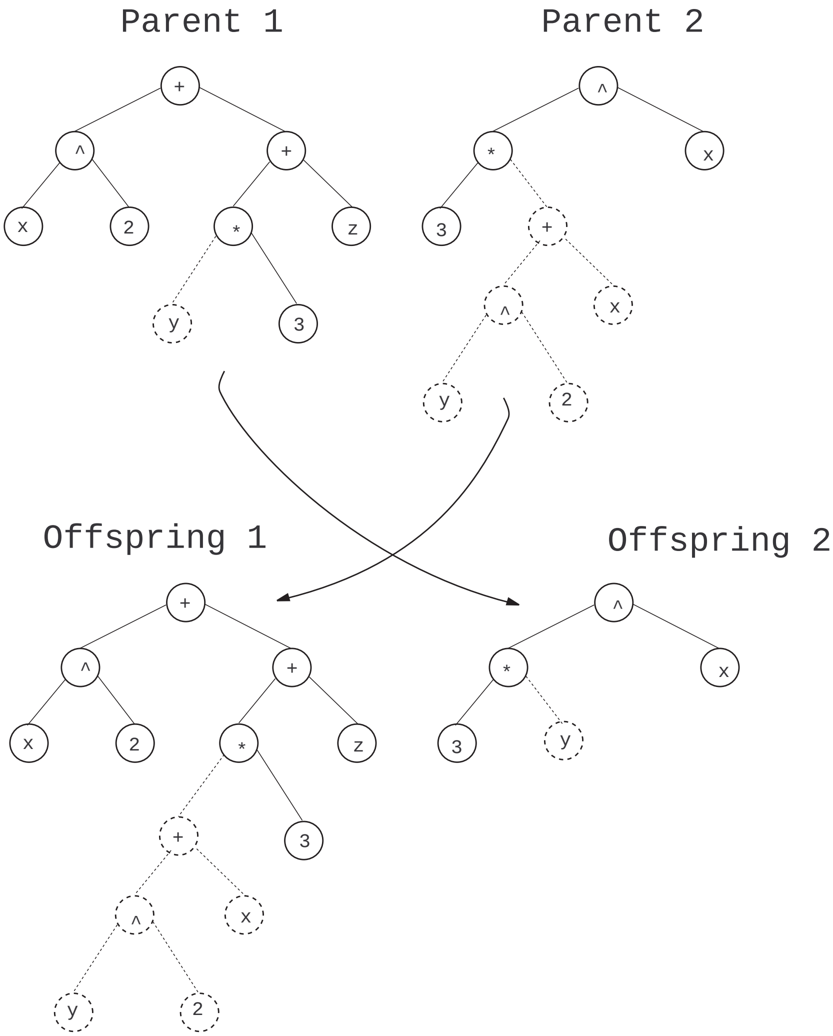


Figure 3.3: An example of the crossover operator

direction of the food. The alphabet could include such functions as *if_food_ahead*, *move*, *left*, *right*. Note that the latter three functions are used only for their side effects that allow the ant to move around the grid and change its facing direction. These functions are of 0-arity (i.e., have no arguments) and therefore would be placed in the *Terminal Set*, T. Whereas the function *if_food_ahead* is of arity two (i.e. requires two arguments, namely an action to carry out if there is food ahead and an action to carry out if there is not food ahead) and so is a member of the *Function Set*, F. These movements around the grid based on whether food is present or not need to be linked and iterated so that the entire trail may be navigated. This is achieved through the $progn_2$ and $progn_3$ functions that take 2 and 3 arguments respectively and carry out the actions specified by their arguments in sequence. For example, $progn_3(left, move, move)$ would cause the artificial ant to turn left and then move forward two squares on the grid. These functions require arguments and so are placed in the function set.

The food finding mission of the problem also has an influence on the design of the fitness function. A good indication of the performance of a particular strategy could be the amount of food eaten by the traversing ant, in which case the optimum fitness for a strategy would be the value eighty-nine (the number of pieces of food on the trail). However this could be achieved by chance by an unintelligent strategy that allows the ant to navigate the entire grid. In order to eliminate this possibility, a time limit is placed on the ant. Namely, if each move or turn in the strategy takes one time step then a limit is placed on the number or these moves or turns taken. In (Koza 1992) a time limit of 400 time steps is considered sufficient time to allow a non-random walk along the trail to achieve maximum fitness.

Once the alphabet and fitness function have been determined, the application of the evolutionary search may take place. An initial population can be created and successive generations evolved through crossover and reproduction. The iterative process continues until an optimal navigation strategy has been evolved, or the maximum number of generations exceeded.

Section 7.2 in (Koza 1992) reports on the results of applying Genetic Programming to the artificial ant problem with a population of 500 and a maximum of 50 generations. In one of the runs carried out a 100% correct navigation strategy (i.e., all eighty-nine pieces of food along the trail were found and eaten by the artificial ant within 400 time steps) was evolved by generation twenty-one.

3.2.7 Discussion

Genetic Programming has been used to solve a wide range of problems in its standard form, (e.g. see (Koza 1992)), and has become a major research area, as evidenced by the wide range of papers presented at the major conferences on evolutionary computing in recent years (including GECCO, CEC, GP, EuroGP, ICGA, and PPSN) and in a number of authored books (Koza 1992, Koza 1994, Koza *et al.* 1999, Banzhaf *et al.* 1998, Langdon 1998, Nordin 1997) and specially edited volumes on the subject (Kinnear, Jr. 1994, Angeline and Kinnear, Jr. 1996, Spector *et al.* 1999). A number of extensions to the original paradigm have been made to increase the applicability and efficiency of Genetic Programming. Some of these extensions include the encapsulation of building blocks during evolution through the use of Automatically Defined Functions (Koza 1994) which most notably has allowed Genetic Programming to excel in the field of analogue circuit design, where it has equalled and outperformed the existing human techniques (Koza *et al.* 1999). This technique will be further investigated in chapter 5. The introduction of a type system (Montana 1995) to Genetic Programming has opened up a number of more complex problems not conforming to the closure constraint as we will see in the next section. Finally the evolution of machine code programs (Nordin 1997) has increased the speed and therefore the efficiency of the Genetic Programming algorithm so that solutions may be obtained in a fraction of the previous times necessary.

3.3 Strongly Typed Genetic Programming

3.3.1 Closure

As mentioned in section 3.2, the standard Genetic Programming algorithm assumes the closure of the alphabet used to compose the individuals that are evolved. Closure can even be imposed on an alphabet by defining special *protected* functions. The closure constraint is enforced so that the genetic operators may be applied to arbitrary points within a program tree and not result in runtime errors.

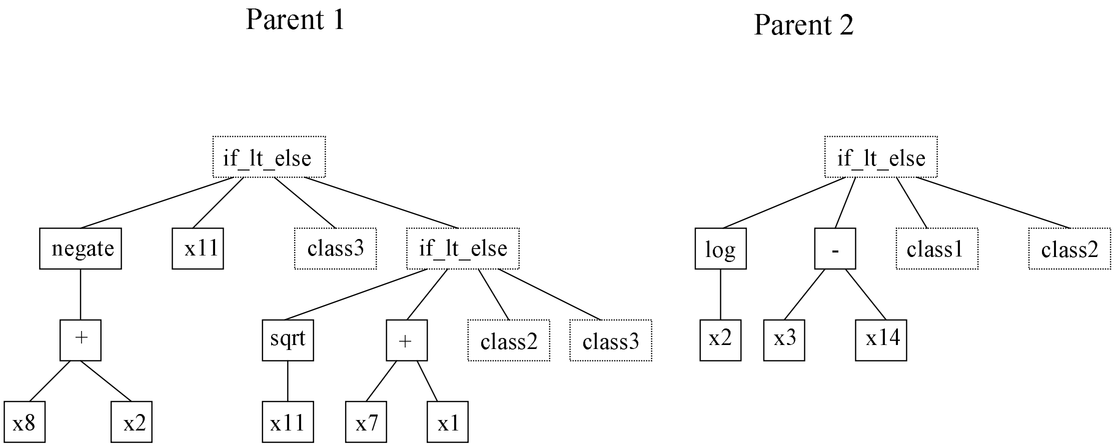
The closure constraint on the learning alphabet essentially forces the problem to be solved using a single data type. While this is fine for simple problems, it is unrealistic to expect complex problems to be solved by obtaining a computer program, when in reality computer programs routinely manipulate multiple data types. In addition enforcing closure leads to artificially formed solutions. For example it would be very difficult to impose closure on the alphabet used to solve a three-class classification problem as the use of boolean operators in the body of the program tree dictates that only a boolean value may be returned. In order to overcome these limitations, strong typing can be introduced to the Genetic Programming algorithm.

3.3.2 Syntactically Constrained Structures

Consider the situation where we have conflicting data types in the alphabet which is to be used to solve a problem, e.g., a three-class classification problem where boolean functions used in the body of the program leads to the requirement of a boolean value being returned due to the closure constraint - which value can be used to represent the third class? The syntactically constrained structure extension to Genetic Programming ((Koza 1992), chapter 19) attempts to deal with this problem by defining constraints on which nodes can appear in which position in a program tree.

Take, for example, the *Thyroid* problem (Wer 1992). This problem consists of 4000 training examples (and 3500 test examples) made up of twenty-one real-valued measurements. The aim is to generate a classifier that is capable of correctly classifying the test cases into one of three classes (two classes signifying variants of thyroid illness and the other signifying no thyroid illness). In order to apply standard Genetic Programming to this problem it was necessary to split the problem up into two subproblems (Gathercole 1998) as subdividing a program tree's real-valued output into ranges (i.e., output < 0 signifying class 1, output $=$ 0 signifying class 2, and output ≥ 0 signifying class 3) gave disappointing results. The two sub-problems consisted of learning a program tree to distinguish between the examples that belonged to class three and those that did not, and then a second tree to distinguish from the examples that were not in class 3, those that belonged to class 1 and those that belonged to class 2.

An alternative to tackling the problem in this manner would be to allow the program tree to return a class label (i.e., class1, class2, etc.). However this would destroy the closure of the learning alphabet that has been designed around the real-valued type of the constants in the problem. For instance, it would make no sense to attempt to add $class1$ to 23.5. However if we define some simple rules that specify which functions may be applied to which other members of the alphabet, then we may create program trees



The dotted lines indicate nodes of the class label return type, while the solid lines indicate nodes of the real-valued return type.

$$\begin{aligned} Parent_1 = \quad & if - (x_8 + x_2) < x_{11}\ then\ class_3 \\ & else\ if \sqrt{x_{11}} < (x_7 + x_1)\ then\ class_2 \\ & else\ class_3 \\ Parent_2 = \quad & if \log x_2 < (x_3 - x_{14})\ then\ class_1 \\ & else\ class_2 \end{aligned}$$

Figure 3.4: An example of the constraints to be preserved by the crossover operator when using Constrained Syntactic Structures

that do make sense. For example we could have the arity-4 function *if_lt_else*, which returns its third argument if its first argument is less than or equal to its second argument otherwise it returns its fourth argument, in the alphabet for this problem. Clearly the first two arguments must be numeric values, however we could specify in our constraints that the third and fourth arguments be class labels. Therefore the return type of the *if_lt_else* function would be a class label. If one of these function symbols was placed at the top level of our program trees, then it would allow the program tree to return a class label instead of a real-valued number (see Figure 3.4).

The constraints are defined before learning takes place and they have an impact on the evolutionary process of the algorithm. It is necessary for the constraints to be respected during the creation of the initial population so that no runtime errors due to functions taking inappropriate data types as arguments occur during evaluation. In addition it is necessary that the constraints be maintained during recombination for the same reason. In order to comply, the crossover and mutation operators have to be adapted. The adapted crossover operator selects any point in the first parent program tree, but then selects a node of the same type in the second parent program tree. The operator then continues as normal. Figure 3.4 shows two mixed type parent program trees that have been selected in order to carry out crossover. The dotted lines indicate nodes of one particular return type, while the solid lines indicate another return type. Only nodes of the same return type may be exchanged during crossover in order to create syntactically correct program trees. The mutation operator creates a new subtree that maintains the predefined constraints to replace the subtree being mutated.

3.3.3 Introducing a Type System

Montana's Strongly Typed Genetic Programming (Montana 1995) is a generalisation of Koza's Syntactically Constrained Structures. Montana introduced a type system to standard Genetic Programming in order to overcome the problems incurred by the closure constraint imposed on the learning alphabet. The type system is similar to the pre-specified constraints of Syntactically Constrained Structures as it specifies constraints indirectly *a priori* through the types of arguments of the functions and the return types of both the functions and terminals. However, the constraints are implicit and come with the type system, whereas in Syntactically Constrained Structures, they are explicit and hand crafted by the user. A program tree is required to return the expected type for the problem it is attempting to solve and each node within the program tree must return the type expected by its parent node. The type system therefore constrains the search space by only allowing a subset of the combinations of symbols from the alphabet to be combined.

As with constraints of Syntactically Constrained Structures, the type system has an impact on the evolutionary process. During the creation of the initial population, individuals have to be created according to the type system. A node selected to be inserted into a program tree must be of an appropriate type according to the type system. Type possibility tables (Montana 1995) can be used to generate the possible type consistent combinations of functions and terminals that may occur at the different depths of a program tree. Also during the recombination of individuals, the constraints of the type system have to be maintained. The algorithm for crossover is very similar to the crossover algorithm under Syntactically Constrained Structures. Any node is selected in the first parent and a node of the appropriate return type in the second parent. The subtrees rooted at these nodes are then exchanged. The mutation operator takes advantage of the algorithm used to generate the individuals in the initial population to generate a randomly grown subtree to replace the subtree being mutated.

In addition to the type constraints, Strongly Typed Genetic Programming allows the use of Generic functions and Generic data types. Generic functions will accept and return a variety of different types so that an individual function does not have to be defined for each type. For example the addition function is able to accept and return both floating point numbers and integers. When a generic function is placed into a program tree, it is instantiated with the argument and return types that it will take in that particular context. It then can be treated as a conventional typed function node. A Generic Data Type is a set of possible data types. They are treated as algebraic quantities until they are evaluated, when they are instantiated according to the context of the training examples. Generic data types make it possible for generic functions to be evolved.

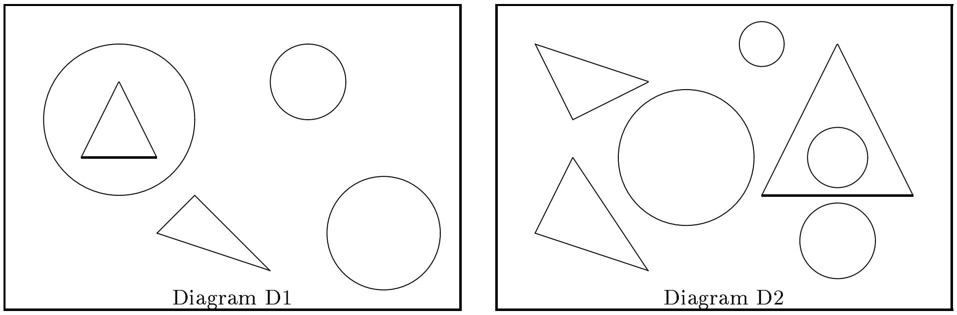
Strongly Typed Genetic Programming has been found to be very effective in obtaining solutions to problems involving multiple data types compared with standard Genetic Programming. Montana reports a number of experiments including matrix and vector manipulation and list processing functions where this is the case (Montana 1995). Haynes finds that the Genetic Programming solution to a problem, involving the evolution of cooperation strategies for predators to capture prey, to be *"significantly inferior"* to the solution obtained using Strongly Typed Genetic Programming (Haynes *et al.* 1995). In addition, some extensions to Strongly Typed Genetic Programming have further enhanced its performance on problems involving multiple data types. In (Haynes 1995, Haynes *et* *al.* 1996) Haynes extends Strongly Typed Genetic Programming to allow a hierarchy of types in an object orientated manner in order to solve a Clique detection problem. In (Clack and Yu 1997) the authors extend Montana's Strongly Typed Genetic Programming with a more compact expression-based program tree, a type unification algorithm instead of a type lookup table, and higher order functions. Crossover on partial application of functions provides more diversity in the population allowing smaller populations to be evolved. They evolve a solution to some list manipulation programs 6 to 7 times faster than Montana's. In (Yu and Clack 1998*a*, Yu and Clack 1998*b*) Yu and Clack present a polymorphic Genetic Programming system based on the principles of the strongly Typed Genetic Programming system presented in (Clack and Yu 1997) that evolves Haskell programs.

3.4 Evolving Escher Programs

3.4.1 Representation

STEPS is a Strongly Typed Evolutionary Programming System that also builds on the STGP approach to learning. The extensions have evolved due to the differences in the type of problems tackled, and the representations used. The main application of STEPS is Concept Learning (see Chapter 4). Therefore there is no need for the evolution of Generic and Polymorphic functions used when carrying out program synthesis. The examples for each concept learning problem are represented as individual closed terms which allows an algorithm to automatically generate an alphabet of function combinations for each problem, instantiating the types of polymorphic functions *a priori.* This alphabet is combined to form an Escher program where local variables are explicitly quantified, so that the evolved programs must be variable consistent in addition to type consistent.

The aim of STEPS is to evolve highly expressive Escher programs. The particular knowledge representation strategy employed by STEPS is the *individuals-as-terms* approach described in Section 2.4. In this approach the examples are provided as single closed terms encapsulating all the properties of the particular individual being represented. Therefore during learning these properties must be extracted from the terms so that comparisons



```
data Shape = Circle | Triangle | Inside(Shape, Shape);
type Diagram = {(Shape, Int)};
d1 = {(Circle,2), (Triangle,1), (Inside(Triangle, Circle),1)};
d2 = {(Circle,3), (Triangle,2), (Inside(Circle, Triangle),1)};
```

Figure 3.5: Example diagrams and their corresponding Escher representations

and inferences can be made on them. This is achieved by using special functions known as *selector functions*.

The selector functions generated for a particular problem constitute a major portion of the learning alphabet. These special functions are made up of a sequence of symbols. Recall the example given in section 2.4.3:

```
exists \x -> x ’in’ d1,
```

which obtains a component `(Shape, Int)` in d1 in Figure 2.4 reproduced here as Figure 3.5 for convenience. Note that the subtree version of the selector function has an extra `&&` node with an empty second branch. This is necessary so that conditions on the selected component can be *hooked* onto the subtree during learning. The semantics of a selector function could be destroyed if some of the symbols constituting the function were altered or removed. Therefore the combination of symbols making up each selector function must be fixed during learning. Hence they are placed into the learning alphabet in the form of a subtree instead of individual symbols, so they may be placed directly into a program tree as a single unit. For example, the above selector function appears as Figure 3.6(a) in subtree form.

Once the components of the data structures have been selected, conditions can be built on

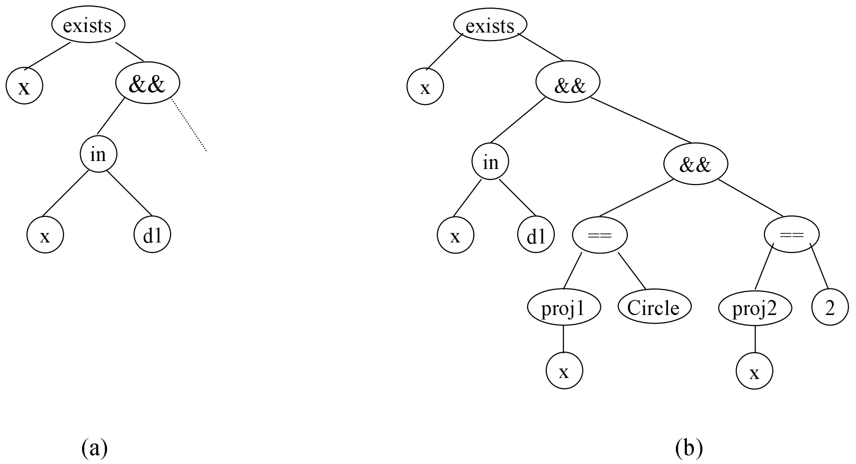


Figure 3.6: (a) A sample selector in tree form, (b) A sample condition in tree form

them or they can be compared to values or other data types. For example, the following expression tests whether the number of circles in d1, in Figure 3.5, is equal to 2.

```
exists \x -> x 'in' d1 && (proj1(x) == Circle && proj2(x) == 2)
```

The equivalent tree form appears as Figure 3.6(b). Once the selector functions have been obtained and placed into the learning alphabet, the conditions placed on them can be evolved from other symbols in the alphabet. This is achieved by filling the blank slots in the selector function subtrees (i.e., the empty second argument of the function `&&` in Figure 3.6 (a)) during the creation of a new program tree.

As mentioned in section 2.4.3, an algorithm has been designed that automatically generates the necessary selector functions for a problem from the data types provided in the training examples (Bowers *et al.* 1999). The *Enumerate* algorithm has been adapted to return selector functions in partially created subtree form suitable for the STEPS approach to learning instead of the full conditional statements as in its original design. The *AdaptedEnumerate* algorithm is given in Figure 3.7. In the current implementation of the AdaptedEnumerate algorithm n is restricted to 1 (i.e., only a single item can be extracted

```
**Input**: A set $S$ of terms of type $\mu$
**Output**: A set $P$ of partially created subtrees
$P = \emptyset$
case $\mu$ of

  • base data type (e.g. bool, Int, Real, etc.):
      $P_1 = (\{t\#*\})$, where $t \in S$ and $\#$ is one of the comparison operators $=$, $<$, $>$, $\leq$, $\geq$ (appropriately chosen according to whether $\mu$ has an ordering defined on it).
      $P_2 = \{\exists x_1 \ldots x_n.(t = F(x_1, \ldots, x_n) \wedge *)\} \cup (AdaptedEnumerate(\{x_1\}) \cup \ldots \cup (AdaptedEnumerate(\{x_n\})$, where $t \in S$, and $F$ is a constructor of arity $n$ having codomain of type $\mu$.
      $P = P \cup P_1 \cup P_2$.
      return $P$.

  • tuple type:
      $P_1 = AdaptedEnumerate(\{proj_{i_1}(t_1), \ldots, proj_{i_n}(t_n)\})$, where $t_1, \ldots, t_n \in S$, $proj_i$ is the $i$th projection, $1 \leq i \leq m$, $m$ is the arity of tuples of type $\mu$, and $proj_{i_1}(t_1), \ldots, proj_{i_n}(t_n)$ have the same type.
      $P = P \cup P_1$.

  • set type:
      $P_1 = \{\exists x_1 \ldots x_n.(x_1 \in t_1 \wedge \ldots \wedge x_n \in t_n \wedge *)\} \cup (AdaptedEnumerate(\{x_1, \ldots, x_n\})$, where $t_1, \ldots t_n \in S$.
      $P_2 = \{card(\{(x_1 \ldots x_n) \mid x_1 \in t_1 \wedge \ldots \wedge x_n \in t_n \wedge *\})\#*\} \cup AdaptedEnumerate(\{x_1 \ldots x_n\})$, where $t_1 \ldots t_n \in S$, $\#$ is one of the comparison operators $=$, $<$, $>$, $\leq$, $\geq$.
      $P = P \cup P_1 \cup P_2$.

  • list type:
      $P_1 = \{\exists x_1 \ldots x_n.(x_1 \text{ `}elem\text{` } t_1 \wedge \ldots \wedge x_n \text{ `}elem\text{` } t_n \wedge *)\} \cup (AdaptedEnumerate(\{x_1, \ldots, x_n\})$, where $t_1, \ldots t_n \in S$.
      $P_2 = \{length([(x_1 \ldots x_n) \mid x_1 \text{ `}elem\text{` } t_1 \wedge \ldots \wedge x_n \text{ `}elem\text{` } t_n \wedge *])\#*\} \cup AdaptedEnumerate(\{x_1 \ldots x_n\})$, where $t_1, \ldots t_n \in S$, $\#$ is one of the comparison operators $=$, $<$, $>$, $\leq$, $\geq$.
      $P = P \cup P_1 \cup P_2$.

n.b. the $*$ indicates an empty branch in the subtree
```

Figure 3.7: The AdaptedEnumerate algorithm

from a single set or list at a time), resulting in all calls to the algorithm being made with a singleton S. This is a limitation and should therefore be addressed as further work. The Enumerate algorithm has been recently extended to include trees, graphs, and multisets (Bowers *et al.* 1999). These data types are not strictly necessary for the problems considered in this thesis, but the AdaptedEnumerate algorithm can be similarly extended.

3.4.1.1 An Illustration of the AdaptedEnumerate Algorithm: The Bongard Problem

In order to illustrate the functioning of the AdaptedEnumerate algorithm, the representation for the Bongard problem presented in Figure 3.5 will be used as a working example. The aim of the problem is to classify diagrams into one of two classes. Each example is a diagram and diagrams in the individuals-as-terms representation are sets of pairs.

1. $AdaptedEnumerate(\{(d1 :: \{(Shape, Int)\})\})$

   $d1$ is of type set, therefore we can

   (a) pull an item out of a set:

   ```
   exists \x1 -> x1 'in' d1
   ```

   putting the resulting subtree into the alphabet for learning and then apply AdaptedEnumerate to the set item extracted:

   $AdaptedEnumerate(\{(x1 :: (Shape, Int))\})$

2. $AdaptedEnumerate(\{(x1 :: (Shape, Int))\})$

   $x1$ is a tuple, therefore we can

   (a) project onto each of the positions of the tuple and apply the AdaptedEnumerate algorithm to each of the projections

   $AdaptedEnumerate(\{(proj1(x1) :: Shape), (proj2(x1) :: Int)\})$

3. $AdaptedEnumerate(\{(proj1(x1) :: Shape)\})$

   $proj1(x1)$ is of type $Shape$ which is a base data type. However there are two cases:

(a) case 1: the data type *Shape* is simply a *Triangle* or a *Circle*. In which case we obtain the following

```
proj1(x1) ==
```

The corresponding subtree is placed in the alphabet and no more recursive calls are made to the AdaptedEnumerate algorithm.

(b) case 2: the data type *Shape* consists of a *Shape* inside another *Shape*. This is achieved by using the constructor function *Inside*. The shapes combined by the constructor function are explicitly quantified so that conditions can be made on them:

```
exists \x2,x3 -> (proj1(x) == Inside(x2,x3)
```

The corresponding subtree is placed in the alphabet and the quantified variables are presented to the AdaptedEnumerate algorithm:

$AdaptedEnumerate(\{(x2 :: Shape), (x3 :: Shape)\})$

note this will result in similar processing to step 3 above.

4. $AdaptedEnumerate(\{(proj2(x1) :: Int)\})$

$proj2(x1)$ is of type *Int* which is a base data type.

(a) We can compare the value of $proj2(x1)$ to other integer values by:

```
proj2(x1) #
```

where # is one of the comparison operators (e.g. $<$, $\leq$, $>$, $\geq$, $==$)

The corresponding subtree is placed in the alphabet and no more recursive calls are made to the AdaptedEnumerate algorithm.

3.4.1.2 An Illustration of the AdaptedEnumerate Algorithm: The Trains Problem

To illustrate further the workings of the AdaptedEnumerate algorithm, the trains problem presented in section 4.4.2 is considered. The Escher description of a train can be seen in

```
data Shape = Rectangular | DoubleRectangular | UShaped |
             BucketShaped | Hexagonal | Ellipsoidal;
data Length = Long | Short;
data Roof = Flat | Jagged | Peaked | Curved | Open;
data Object = Circle | Hexagon | Rectangle | LongRectangle |
              Triangle | InvertedTriangle | Square | Diamond | Null;
data Direction = East | West;

type Wheels = Int;
type Load = (Object,Int);
type Car = (Shape, Length, Wheels, Roof, Load);
type Train = [Car];
```

Figure 3.8: The Escher representation of a train

Figure 3.8. Here each example is a train, where a train consists of a list of cars and where a car is a tuple of features including its shape, its length, the number of wheels it has, the type of roof it has, and its load. The load of a train is itself a pair consisting of the type of load and the quantity.

1. $AdaptedEnumerate(\{(x1 :: [Car])\})$

   $x1$ is of type list, therefore we can

   (a) pull an item out of a list:

   ```
   exists \x2 -> (x2 ‘elem‘ x1)
   ```

   putting the resulting subtree into the alphabet for learning and then apply AdaptedEnumerate to the list item extracted:

   $AdaptedEnumerate(\{(x2 :: Car)\})$

   (b) or we can obtain the length of a list:

   ```
   length(filter (\x3 -> (x3 ‘elem‘ x1)))
   ```

   putting the resulting subtree into the alphabet for learning and then apply AdaptedEnumerate to the list item:

   $AdaptedEnumerate(\{(x3 :: Car)\})$

note this results in a identical processing by the AdaptedEnumerate algorithm to the case when the list item is represented by $x2$ below

2. $AdaptedEnumerate(\{(x2 :: Car)\})$

   the Car data type is a tuple with five positions, therefore we can create selector functions to project onto each of these five positions and apply the AdaptedEnumerate algorithm to each of the projections

$$\begin{aligned} AdaptedEnumerate \quad & (\{(proj1(x2) :: Shape), (proj2(x2) :: Length) \\ & (proj3(x2) :: Wheels), (proj4(x2) :: Roof), \\ & (proj5(x2) :: Load)\}) \end{aligned}$$

3. $AdaptedEnumerate(\{(proj1(x2) :: Shape), (proj2(x2) :: Length),$
   $(proj3(x2) :: Wheels), (proj4(x2) :: Roof),$
   $(proj5(x2) :: Load)\})$

   (a) The result of projecting onto each of the first four positions of $x2$ obtains a base data type. Therefore appropriate comparisons can be made with each of them using appropriate comparison operators:

   ```
   proj2(x1) #
   ```

   where # is one of the comparison operators (e.g. $<$, $\leq$, $>$, $\geq$, $==$)

   The corresponding subtrees are placed in the alphabet and no more recursive calls are made to the AdaptedEnumerate algorithm.

   (b) however the data type of the fifth position of the tuple is itself a tuple of two base data types therefore the AdaptedEnumerate algorithm is applied to the projections onto each of the position in the pair.

   $AdaptedEnumerate(\{(proj1(proj5(x2)) :: Object), (proj2(proj5(x))) :: Int)\})$

   where subtrees allowing appropriate comparisons are placed in the alphabet for learning.

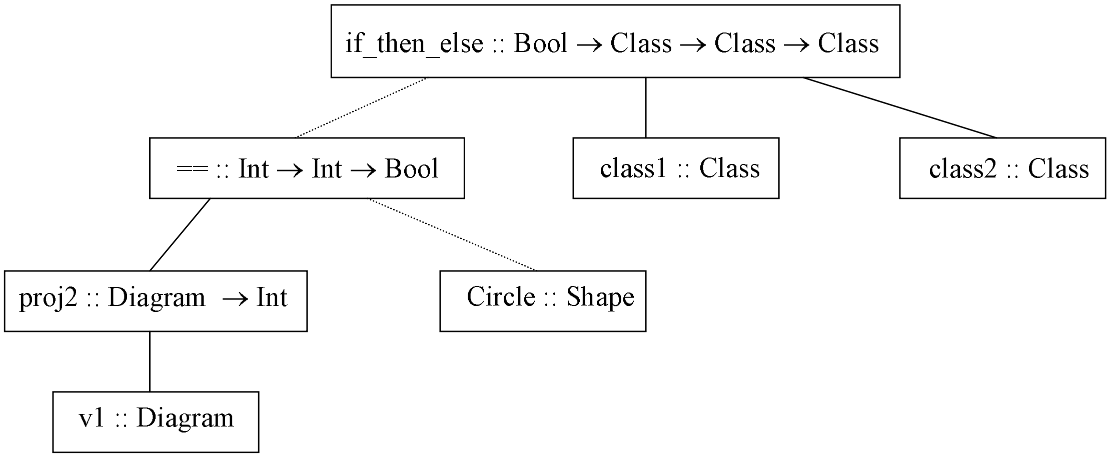


Figure 3.9: An example of type consistency violation

3.4.2 Initialisation

Trees in the initial population are formed by randomly selecting and combining subtrees from the problem alphabet. The total alphabet for a problem consists of the appropriate selector function subtrees, any additional functions provided by the user, and the domain-derived constants. These constants are all values extracted from the training examples provided to the learner. The function set provided by the user typically includes the connective functions `&&` and `||` (the boolean functions conjunction and disjunction) so that a number of comparisons can be made on the components of the data types.

However, subtrees selected to fill in a blank slot in a partially created program tree must satisfy certain constraints so that only valid Escher programs are produced. These constraints are type and variable consistency. In order to maintain type consistency, each node in a subtree in the alphabet is annotated with a type signature indicating its argument and return types. A subtree selected to fill in a blank slot must be of the appropriate return type. The program tree in figure 3.9 is an example of type consistency violation.

Here, type signatures are in curried form and dotted-lines indicate where a subtree has been added. The addition of the

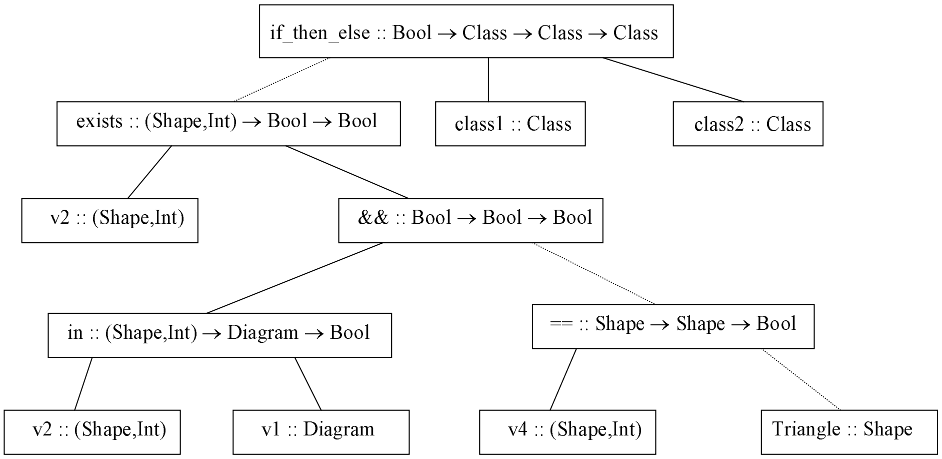

if_then_else :: Bool → Class → Class → Class
exists :: (Shape,Int) → Bool → Bool
class1 :: Class
class2 :: Class
v2 :: (Shape,Int)
&& :: Bool → Bool → Bool
in :: (Shape,Int) → Diagram → Bool
== :: Shape → Shape → Bool
v2 :: (Shape,Int)
v1 :: Diagram
v4 :: (Shape,Int)
Triangle :: Shape

Figure 3.10: An example of variable consistency violation

```
Circle :: Shape
```

subtree violates type consistency, as it is of type Shape and the function

```
== :: Int -> Int -> Bool
```

requires a subtree returning type Int as its second argument.

In order to maintain variable consistency, the local variables in a subtree selected to fill in a blank slot in the partially created program tree must be within the scope of a quantifier. In addition, all quantified variables in a program tree must be used in the conditions of their descendant subtrees to avoid redundancy. The program tree in figure 3.10 is an example of variable consistency violation.

The addition of the subtree rooted at

```
== :: Shape -> Shape -> Bool
```

violates variable consistency as the variable

```
v4 :: Shape
```

is not within the scope of a quantifier. In addition variable consistency is violated by not using the quantified variable

```
v2 :: (Shape, Int).
```

3.4.3 Genetic Operators

3.4.3.1 Modified Crossover

The requirement for type and variable consistent program trees needs to be maintained during the evolution of the programs so that only syntactically correct programs are evolved. In addition to this, it is necessary to preserve the structure of the selector function subtrees. This results in a situation where crossover can only be applied to certain nodes within a program tree. These crossover points correspond to the roots of the subtrees in the function set. Once a crossover point has been randomly selected from the first parent, a crossover point that will maintain type and variable consistency can be randomly selected in the second parent. If no such crossover point is available then an alternative crossover point is sought. Due to the strong contraints that need to be maintained during the crossover of subtrees, subtrees are exchanged in one direction only, i.e., the subtree rooted by the crossover point selected in the first parent is exchanged for the subtree rooted at the crossover point selected in the second parent, and not vice versa.

Note empirical results show that due to the strong constraints necessary in order for crossover to take place, a structure altering crossover operation (i.e. an exchange of subtrees rooted by nodes other than the roots of the parent trees) can take place as little as 2% of the time for some structured problems (e.g. the trains problem described in section 4.4.2). An extension of the implementation of STEPS based on the unification of variables would increase the applicability of the crossover operator for these cases, however it has not been necessary for the problems considered here and is therefore left as a consideration for further work.

3.4.3.2 Mutation

During successive iterations of the evolutionary process, the amount of genetic variation in a population decreases. In an extreme case, this can lead to the loss of genetic material that is essential to the search for an optimal solution and a method for reintroducing such lost material is required. This effect is extreme in STEPS as the crossing over of subtrees during recombination only occurs in one direction due to the necessary constraints that have to be preserved. STEPS ensures the preservation of genetic diversity through the extensive application of mutation operators, unlike traditional Genetic Programming. These mutation operators are applied to the terminal and functional nodes of a tree. The functional mutation can only be applied at the crossover points in a program tree and must preserve type and variable consistency. The terminal mutation operator is applied to alter the constant values in a program tree. Some additional mutation operators, tailored to the problem of concept learning have also been designed (see chapter 4).

3.4.4 Learning Strategy

STEPS creates an initial population of a specified size ensuring that each tree preserves the necessary constraints and is unique. In order to perform population updates, *Generational Replacement* was found to be more suitable than the *Steady State Replacement* (Syswerda 1989) technique usually favoured by Strongly Typed Genetic Programming based approaches (Montana 1995, Clack and Yu 1997). This is due to the increased rate at which trees with low fitness rates, containing essential genetic material necessary to the search for the solution, are discarded. Parent program trees are selected by the tournament selection technique and are recombined using both crossover and mutation. Again the choice of tournament selection over fitness proportionate selection was made to prevent the loss of essential genetic material in the early generations. The single direction of the crossover operator used by STEPS accentuates this problem which can lead to the premature convergence of the population onto a suboptimal solution if measures such as those mentioned in this section are not taken. An extensive use of the mutation operator will also help to alleviate this problem by reintroducing diversity into the population. A new complete generation is created in an elitist manner by directly copying the best performing 10% of the previous generation. The remaining 90% of the population are created using the genetic operators. However the choice of which operator is to be applied is determined by the individual selected from the current population as will be seen in Section 4.5. Genetic operators that are specialised to a particular problem type may be designed and these operators may be applied in a strategic manner with particular criteria in mind. Fitness is evaluated as the predictive accuracy of a program tree over the set of examples. The evolutionary process of STEPS is depicted in Figure 3.11.

3.5 Summary

In this chapter the need for a more powerful, general search technique in order to allow the vast space of highly expressive Escher programs to be searched has been discussed. An evolutionary approach to learning has been considered, with the particular evolutionary algorithm of Genetic Programming being identified as a suitable candidate to search the vast space of expressive Escher programs which arises from its complex representation and flexible nature. However the Genetic Programming approach to learning has some drawbacks, namely the essential constraint of the closure of the alphabet. Two similar solutions to this problem, Constrained Syntactic Structures and Strongly Typed Genetic Programming, were presented, Strongly Typed Genetic Programming being the more popular approach to be adapted by the Genetic Programming community. Finally it was shown how the basic ideas of Strongly Typed Genetic Programming could be extended so that STEPS, whose aim is to generate expressive Escher programs from the complex individuals-as-terms representation (presented in the previous chapter), could be designed.

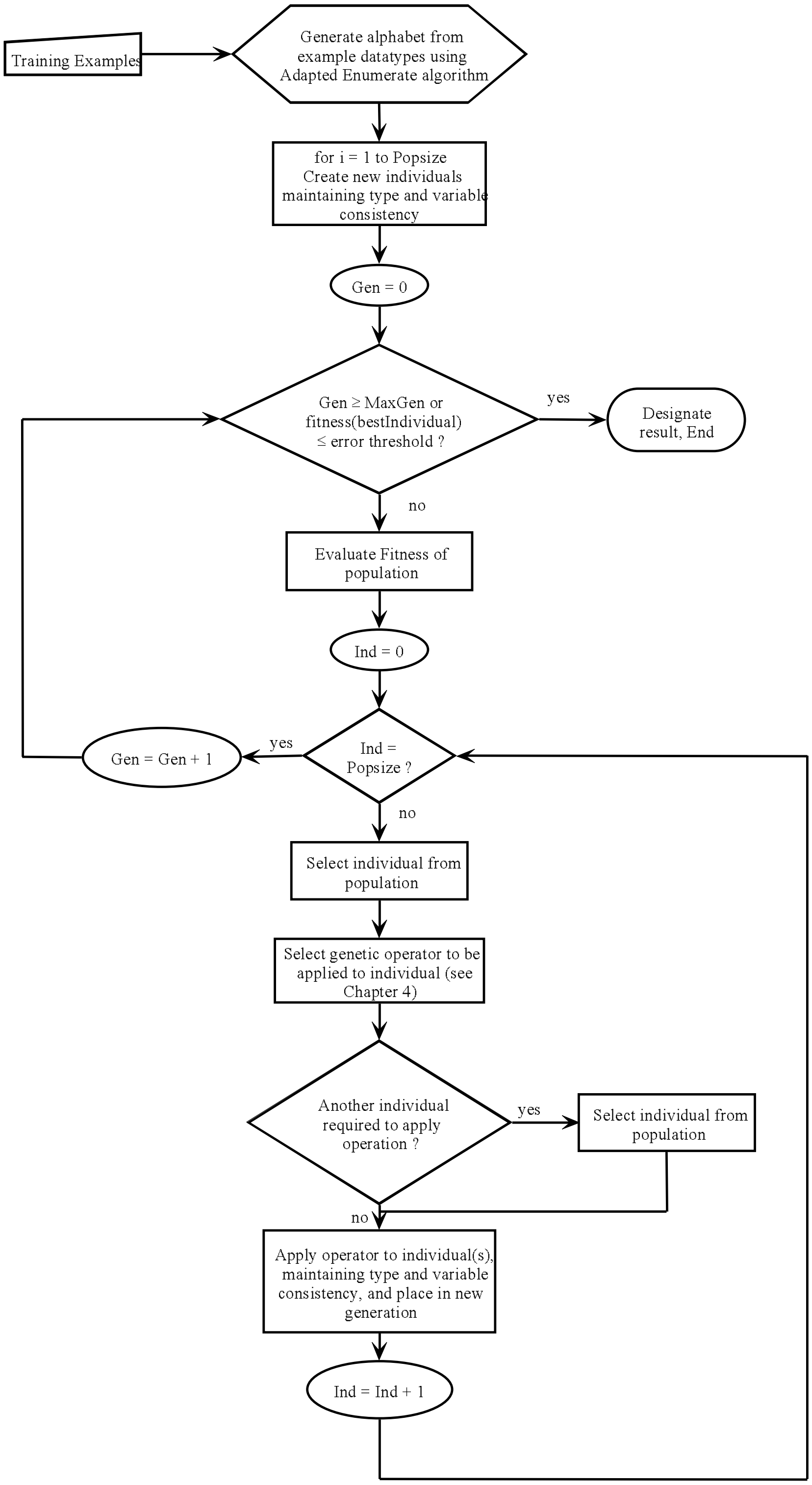

Training Examples
Generate alphabet from example datatypes using Adapted Enumerate algorithm
for i = 1 to Popsize Create new individuals maintaining type and variable consistency
Gen = 0
Gen ≥ MaxGen or fitness(bestIndividual) ≤ error threshold ?
yes
Designate result, End
no
Evaluate Fitness of population
Ind = 0
Ind = Popsize ?
yes
Gen = Gen + 1
no
Select individual from population
Select genetic operator to be applied to individual (see Chapter 4)
Another individual required to apply operation ?
yes
Select individual from population
no
Apply operator to individual(s), maintaining type and variable consistency, and place in new generation
Ind = Ind + 1

Figure 3.11: The evolutionary process of STEPS

Chapter 4

Concept Learning

4.1 Introduction

In this chapter the application of STEPS to the problem of concept learning is discussed. First of the problem of concept learning is reviewed, then the closed term representation used by STEPS to represent the training examples is described. Then some specialised genetic operators that can be applied by STEPS during concept learning are presented. The use of STEPS on some simple concept learning problems is illustrated and some strategies that can be applied when learning concept descriptions are discussed and evaluated (using some simple concept learning problems). Some of the work in this chapter has appeared as (Kennedy 1998, Kennedy and Giraud-Carrier 1999*a*, Kennedy and Giraud-Carrier 1999*b*).

4.2 Concept Learning

Evolutionary techniques have been successfully applied to concept learning in both an attribute value and a relational setting. Approaches in both of these areas can be further divided into those approaches using a mapping between the logical concept descriptions and a bit string representation and those approaches that manipulate the logical expressions directly. The learning systems GIL (Genetic Inductive Learner) (Janikow 1993) and GABIL (Genetic Algorithm based Batch Incremental Learner) (DeJong *et al.* 1993) both use a binary string representation to express concept descriptions in a modified Disjunctive Normal Form(DNF) with internal disjunction. Rules are mapped to a fixed length binary string with a bit for each value that each feature can take. Variable length rule sets to allow for disjunctive rule sets are manipulated. Both approaches use a basic Genetic Algorithm that has been biased towards the task of Concept Learning by defining specialised genetic operators. In GIL these operators are defined at the feature, rule and rule set level.

In (Koza 1991) Genetic Programming is used to learn the functional equivalent to a decision tree classification for an attribute value problem. A direct mapping between the two approaches involves converting the attributes into functions and placing them in the function set and placing the various class names in the terminal set. SAMUEL (Strategy Acquisition Method Using Empirical Learning) is another essentially propositional approach in which the actual representation of the induced rules is manipulated directly (Schultz and Grefenstette 1990). The approach is used to evolve rules that allow an aeroplane to evade a missile.

The learning system DOGMA (Domain Orientated Genetic MAchine) (Hekanaho 1998) and the extension Genetic Logic Programming (Osborn *et al.* 1995) are both approaches that use a binary string to represent a relational program. The Genetic Logic Programming approach is used to evolve Prolog interpreters to be applied to the domain of Natural Language Understanding (NLU). Existing NLU models are placed in a gene pool and each chromosome in the population is effectively a mask which dictates which facts or rules from the existing models will be included in that particular individual. DOGMA uses a *language template* to define the structure and complexity of the bit string formulae. The template is the maximal conjunctive formula representable with the given language; that is, all possible formulae that can be expressed with the given language can be obtained by omitting parts of the template. This is achieved by setting the corresponding bits to 1 in order for a particular formula to express that a predicate takes on a particular value. During evolution an individual corresponds to a *Family* of these formulae. As with the systems GIL and GABIL, DOGMA's evolutionary algorithm is biased towards the task of concept learning through a set of specialised genetic operators. These genetic operators are defined at the formula and family levels. REGAL, like DOGMA uses a language template to map first-order concept descriptions to a fixed length bit string so that they can be manipulated by a Genetic Algorithm (Giordana and Saitta 1993). The language template of REGAL allows the expression of internal negation as well as internal disjunction (i.e., disjuncts and negation are permitted within predicates). REGAL can learn disjunctive concept descriptions either by evolving a single disjunct at a time or by learning many disjuncts at a time by forming sub-populations. REGAL also uses specialised genetic operators during the learning process.

Inductive Genetic Programming with Decision Trees (GPDT) is an alternative method for evolving concept descriptions (Nikolaev and Slavov 1997). In GPDT, a population of decision trees is evolved. The closure property is maintained by designing genetic operations suitable for the decision tree genotype, e.g., crossover points are chosen in order to only produce offspring with non-repeating attribute tests in each branch.

EVIL_1 and Genetic Logic Programming are both learners that induce relational theories by directly manipulating the first-order logic representation (note this Genetic Logic Programming system is a different system to the one described earlier in this section). The Genetic Logic Programming System (GLPS) (Wong and Leung 1995) represents a first-order logic program as a forest of AND and OR trees. Each rule is made up of a skeleton of ORs which connect the clauses of the rule. Each clause is a set of literals contained in an AND subtree or is a leaf node. Crossover may be applied at the rule, the clause and the literal level. During initialisation an AND-OR skeleton is randomly grown for each sub-concept and the target concept to be learnt. Literals to be added to the skeleton are generated from the predicate symbols and terms. EVIL_1 (Reiser and Riddle 1999) consists of a population of *agents* that induce a logical theory using the Inductive Logic Programming system Progol, given some user defined background information and a subset of the training data. Rules are exchanged between agents' theories every certain number of generations. Fitness is evaluated as predictive accuracy over the validation set (which is in fact the entire training set).

The idea of, and basic assumptions for the application of, evolutionary higher-order concept learning were presented in (Kennedy 1998). This chapter details the application of

STEPS to the problem of concept learning within Escher. In this context, examples are closed terms and concept descriptions take the form of program trees. STEPS starts from a randomly generated, initial population of program trees and iteratively manipulates it by genetic operators until an optimal solution is found.

4.3 Specialised Genetic Operators

The aim of concept learning is to induce a definition of a particular entity from a set of examples. The definitions produced are made up of logical expressions combined together by connectives (i.e., conjunctions and disjunctions). Using the idea that definitions are combinations of terms combined with connective functions, a set of genetic operators specialised to the problem of concept learning can be defined. The evolutionary concept learning systems GABIL (DeJong *et al.* 1993), GIL (Janikow 1993), and DOGMA (Hekanaho 1998) all bias a basic genetic algorithm to the task of concept learning by defining specialised genetic operators that operate on the bit string representation. GABIL has two specialised genetic operators in addition to the usual crossover and mutations. These specialised operators are special cases of the mutation operators and have the same effect as adding an alternative value to a feature's values and dropping all the values for a feature which in effect drops the condition from the rule. The addition of these operators significantly improves its performance on concept learning tasks. GIL has a number of specialised operators that can act on the various levels of a rule set, i.e., on the rule set itself, on a particular rule in a rule set and on a particular feature in a rule. The various operators are classified as independent, specialising and generalising depending on the operator's effect on the individuals coverage of the examples. Among DOGMA's genetic operators are two crossover operators that are especially designed with the task of concept learning in mind. The operators are generalising and specialising crossover. The generalising crossover works by performing a bitwise OR on selected bits in the parents to produce offspring. The specialising crossover works in a similar manner except a bitwise AND is performed. REGAL also provides generalising and specialising crossover operators that work in a similar manner to those of DOGMA (Giordana and Saitta 1993).

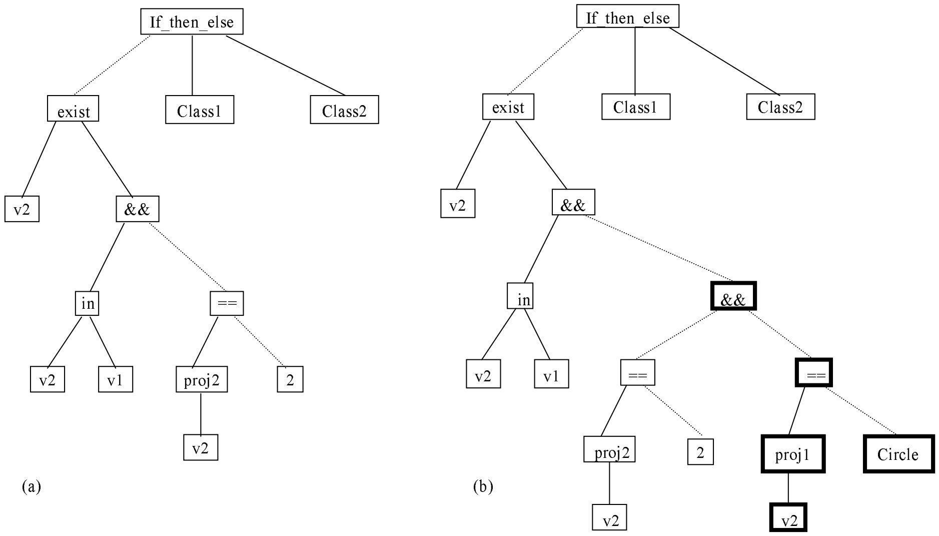


Figure 4.1: Sample AddConjunction Mutation

These ideas can be extended so that a set of specialised operators can be defined that act on an Escher program tree representation. The specialised functional mutations include AddConjunction, DropConjunction, AddDisjunction and DropDisjunction. AddConjunction and AddDisjunction insert an `&&` or `||` node respectively at the node to be mutated. The first argument of the node is the subtree originally rooted at that node and its second argument is randomly grown. For example, if we apply the AddConjunction operator to the `==` node in the tree of Figure 4.1(a), then we could obtain the tree of Figure 4.1(b).

The DropConjunction and DropDisjunction operators randomly select an `&&` or `||` crosspoint respectively, replacing it with the subtree that makes up its first argument.

In addition to the specialised mutations, some specialised crossover operators are available during evolution. The first operator AndCrossover involves randomly selecting a crossover node with a return type `Boolean` in the first parent and a crossover node that preserves type and variable consistency in the second parent. The subtrees rooted at the crossover points in both parents are combined as the arguments to an `&&` node and this new subtree is used to replace the subtree selected from the first parent. The second operator OrCrossover works in the same way, except the subtrees selected from both parents are combined as the arguments to an `||` node.

4.4 Some Simple Concept Learning Problems

All experiments reported in this thesis were carried out on a Sun Enterprise 4000/5000 (168 MHz) or a Sun Ultra 1 (143 MHz). Both of these machines are shared machines therefore all statistics regarding running time are estimates that depend both on the machine used and the load of the machine.

4.4.1 Playing Tennis

Problem Description

For the playTennis problem, the objective is to induce a description that characterises the weather in which it is suitable to play tennis (Mitchell 1997). The weather on a particular day is described by four attributes: Outlook, which can take on the values Sunny, Overcast or Rain; Temperature, which can take on the values Hot, Mild, Cool; Humidity which can take on the values High or Normal; and Wind, which can take on the values Strong or Weak. This is an example of a simple propositional problem and was used as a running example in chapter 2.

Representation

The weather on a particular day in the Escher closed term representation is a tuple of values for the outlook, temperature, humidity, and wind observed on that day. The objective is to induce the function `playTennis` that takes a tuple of values representing the weather for a particular day and returns `True` if the conditions are suitable for playing tennis and `False` if they are not.

```
data Outlook = Sunny | Overcast | Rain;
data Temperature = Hot | Mild | Cool;
data Humidity = High | Normal;
data Wind = Strong | Weak;
data Class = Yes | No;
```

```
type Weather = (Outlook, Temperature, Humidity, Wind);

playTennis :: Weather -> Class;
```

The examples for the problem are as follows:

```
playTennis(Overcast,Hot,High,Weak) = True;
playTennis(Overcast,Hot,Normal,Weak) = True;
playTennis(Rain,Mild,High,Weak) = True;
playTennis(Rain,Cool,Normal,Weak) = True;
playTennis(Overcast,Cool,Normal,Strong) = True;
playTennis(Sunny,Cool,Normal,Weak) = True;
playTennis(Rain,Mild,Normal,Weak) = True;
playTennis(Sunny,Mild,Normal,Strong) = True;
playTennis(Overcast,Mild,High,Strong) = True;
playTennis(Sunny,Hot,High,Weak) = False;
playTennis(Rain,Mild,High,Strong) = False;
playTennis(Sunny,Hot,High,Strong) = False;
playTennis(Rain,Cool,Normal,Strong) = False;
playTennis(Sunny,Mild,High,Weak) = False;
```

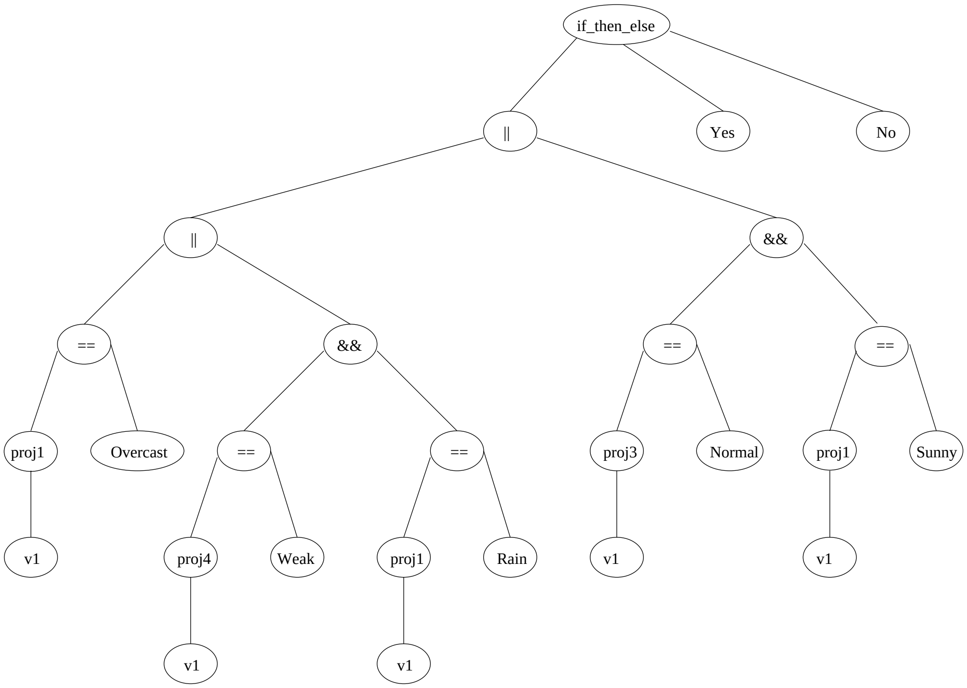
Learning Parameters

The learning parameters used in the experiments are as follows:

Parameter	Setting
Replacement	Generational
Selection	Tournament (size 2)
Population size	300
Maximum Depth	8
Minimum Depth	4
Elitism	10%
Connectives	`&&` , `||`
Comparatives	`==`
Genetic Operators	All

Results

The experiment was carried out 30 times with an optimal solution (i.e. a 100% correct description) found in every run in an average of 13.9 generations or 4170 fitness evaluations. In one run the solution was found within 4 generations. The average size of the solutions found was 40.5 nodes. Figfigtennis is an optimal solution found in one of the runs in tree form and then as an Escher program. It translates into English as *if the outlook is overcast, or the wind is weak and the outlook is rain, or the wind is normal and the outlook is sunny then the weather is suitable to play tennis otherwise it is not.* Each run took between 1.5 and 12 minutes to complete.



```
playTennis(v1) = if proj1(v1) == Overcast ||
                   (proj4(v1) == Weak && proj1(v1) == Rain) ||
                   (proj3(v1) == Normal && proj1(v1) == Sunny)
                 then Yes
                 else No;
```

Figure 4.2: An optimal solution to the Tennis problem in tree and Escher code form

4.4.2 Trains

Problem Description

The objective of the trains problem is to generate a concept description that distinguishes trains that are travelling East from trains travelling West - the so called East West challenge (Muggleton and Page 1994, Michalski and Larson 1977). A train is made up of cars with each car characterised by its shape, the style of its roof (if any), its length, its number of wheels, and the load that it is carrying (if any).

Representation

A train in the Escher closed term representation is a list of cars with each car represented by a tuple of characteristics including shape, length, wheel, roof and load. The load itself is a pair indicating the number of a particular shape that is being carried by the car. The objective is to induce the function `direction` that takes a list of cars that make up a train and returns `East` if the train is travelling East and `West` if the train is travelling West. This is a good example of the compactness of the Escher closed term representation. The same information represented as Prolog flattened horn clauses uses 1427 program symbols in (Muggleton and Page 1994) as compared to the symbols 254 presented here.

```
data Shape = Rectangular | DoubleRectangular | UShaped |
             BucketShaped | Hexagonal | Ellipsoidal;
data Length = Long | Short;
data Roof = Flat | Jagged | Peaked | Curved | Open;
data Object = Circle | Hexagon | Rectangle | LongRectangle |
              Triangle | InvertedTriangle | Square | Diamond | Null;
data Direction = East | West;

type Wheels = Int;
type Load = (Object,Int);
type Car = (Shape, Length, Wheels, Roof, Load);
type Train = [Car];

direction :: Train -> direction;
```

The examples for the problem are as follows:

```
direction([(Rectangular, Long, 2, Open, (Square, 3)),
           (Rectangular, Short, 2, Peaked, (Triangle, 1)),
           (Rectangular, Long, 3, Open, (Hexagon, 1)),
           (Rectangular, Short, 2, Open, (Circle, 1))])
         = East;

direction([(UShaped, Short, 2, Open, (Triangle, 1)),
           (BucketShaped, Short, 2, Open, (Rectangle, 1)),
           (Rectangular, Short, 2, Flat, (Circle, 2))])
         = East;

direction([(Rectangular, Short, 2, Open, (Circle, 1)),
           (Hexagonal, Short, 2, Flat, (Triangle, 1)),
           (Rectangular, Long, 3, Flat, (InvertedTriangle, 1))])
         = East;

direction([(BucketShaped, Short, 2, Open, (Triangle, 1)),
           (DoubleRectangular, Short, 2, Open, (Triangle, 1)),
           (Ellipsoidal, Short, 2, Curved, (Diamond, 1)),
           (Rectangular, Short, 2, Open, (Rectangle, 1))])
         = East;

direction([(DoubleRectangular, Short, 2, Open, (Triangle, 1)),
           (Rectangular, Long, 3, Flat, (LongRectangle, 1)),
           (Rectangular, Short, 2, Flat, (Circle, 1))])
         = East;

direction([(Rectangular, Long, 2, Flat, (Circle, 3)),
           (Rectangular, Short, 2, Open, (Triangle, 1))])
         = West;

direction([(DoubleRectangular, Short, 2, Open, (Circle, 1)),
           (UShaped, Short, 2, Open, (Triangle, 1)),
           (Rectangular, Long, 2, Jagged, (Null, 0))])
         = West;

direction([(Rectangular, Long, 3, Flat, (LongRectangle, 1)),
           (UShaped, Short, 2, Open, (Circle, 1))])
         = West;

direction([(BucketShaped, Short, 2, Open, (Circle, 1)),
           (Rectangular, Long, 3, Jagged, (LongRectangle, 1)),
           (Rectangular, Short, 2, Open, (Rectangle, 1)),
           (BucketShaped, Short, 2, Open, (Circle, 1))])
         = West;

direction([(UShaped, Short, 2, Open, (Rectangle, 1)),
    (Rectangular, Long, 2, Open, (Rectangle, 2))])
  = West;
```

Learning Parameters

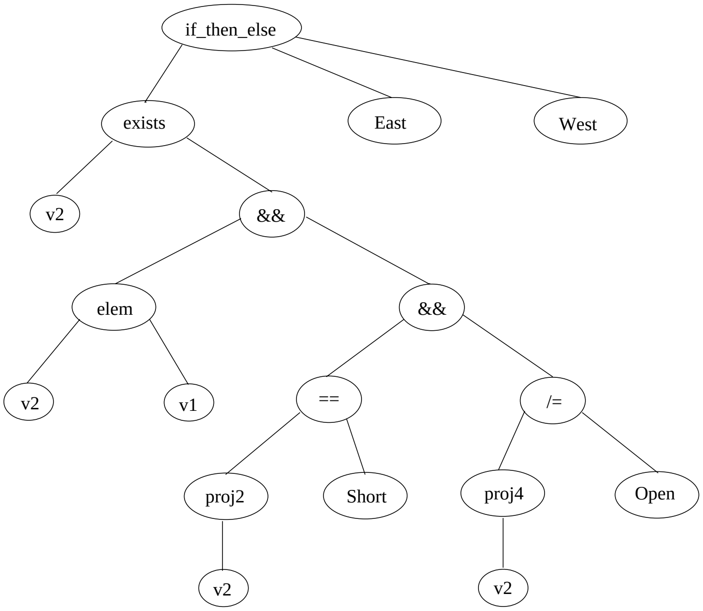
The learning parameters used in the experiments are as follows:

Parameter	Setting
Replacement	Generational
Selection	Tournament (size 2)
Population size	300
Maximum Depth	8
Minimum Depth	4
Elitism	10%
Connectives	`&&`
Comparatives	`==` ,`/=`, `<`, `>=`
Genetic Operators	All but AddDisjunction, DropDisjunction

Note, that the connective `||` is left out of the learning alphabet, and hence the AddDisjunction and DropDisjunction operators not used, as it was found not to be necessary to the solution.

Results

The experiment was carried out for 30 runs with an optimal (i.e. 100% correct) solution found in each run. The optimal solution was found in an average of 9.43 generations or 2829 fitness evaluations with the solution being found in the first generation of one run. The average size of the solutions found was 32.4 nodes. Figure 4.3 is the optimal solution found in one of the runs in tree and Escher code form. In plain English this solution reads as “ *A train is travelling East if it contains a short closed car.*” Each run took between 20 seconds and 15 minutes to complete.



```
direction(v1) = if exists \v2 ->
                     v2 'elem' v1 &&
                     proj2(v2) == Short &&
                     proj4(v2) /= Open
                then East
                else West;
```

Figure 4.3: An optimal solution to the Michalski's train problem in tree and Escher code form

4.4.3 Animals

Problem Description

The objective here is to generate a concept description that distinguishes between animal classes i.e. a description that can classify animals into their correct class (Mammal, Fish, Bird or Reptile). An animal is characterised by its covering, its habitat, the number of legs it has, whether it produces eggs, whether it has gills, whether it produces milk and whether it is homeothermic. This is an example of a classification problem with more than two classes, i.e., this problem would be hard for a standard genetic programming system to solve.

Representation

An animal in the Escher closed term representation is a tuple of characteristics. The objective is to induce the function `animalClass` that takes a tuple of characteristics that make up an animal and returns the class to which the animal belongs to.

```
data Covering = Hair | None | Scales | Feathers;
data Habitat = Land | Air | Water;
data Class = Mammal | Fish| Reptile | Bird;

type Legs = Int;
type Haseggs = Bool;
type Hasgills = Bool;
type Hasmilk = Bool;
type Homeothermic = Bool;

type Animal = (Legs,Haseggs,Hasgills,Hasmilk,Homeothermic,Covering,Habitat)

animalClass :: Animal -> Class;
```

The examples for the problem are as follows:

```
animalClass(4,False,False,True,True,Hair,Land) = Mammal;      --dog
animalClass(0,False,False,True,True,None,Water) = Mammal;     --dolphin
animalClass(2,True,False,True,True,Hair,Water) = Mammal;      --platypus
animalClass(2,False,False,True,True,Hair,Air) = Mammal;       --bat
animalClass(0,True,True,False,False,Scales,Water) = Fish;     --trout
animalClass(0,True,True,False,False,Scales,Water) = Fish;     --herring
animalClass(0,True,True,False,False,Scales,Water) = Fish;     --shark
animalClass(0,True,True,False,False,None,Water) = Fish;       --eel
animalClass(4,True,False,False,False,Scales,Land) = Reptile;  --lizard
animalClass(4,True,False,False,False,Scales,Water) = Reptile; --crocodile
animalClass(4,True,False,False,False,Scales,Land) = Reptile;  --t_rex
animalClass(0,True,False,False,False,Scales,Land) = Reptile;  --snake
animalClass(4,True,False,False,False,Scales,Water)= Reptile;  --turtle
animalClass(2,True,False,False,True,Feathers,Air) = Bird;     --eagle
animalClass(2,True,False,False,True,Feathers,Land) = Bird;    --ostrich
animalClass(2,True,False,False,True,Feathers,Water) = Bird;   --penguin
```

Note, the animal names at the end of each example are not processed by the learning algorithm.

Learning Parameters

Parameter	Setting
Replacement	Generational
Selection	Tournament (size 2)
Population size	300
Maximum Depth	4
Minimum Depth	7
Elitism	10%
Connectives	`&&`
Comparatives	`==`
Genetic Operators	All but AddDisjunction, DropDisjunction

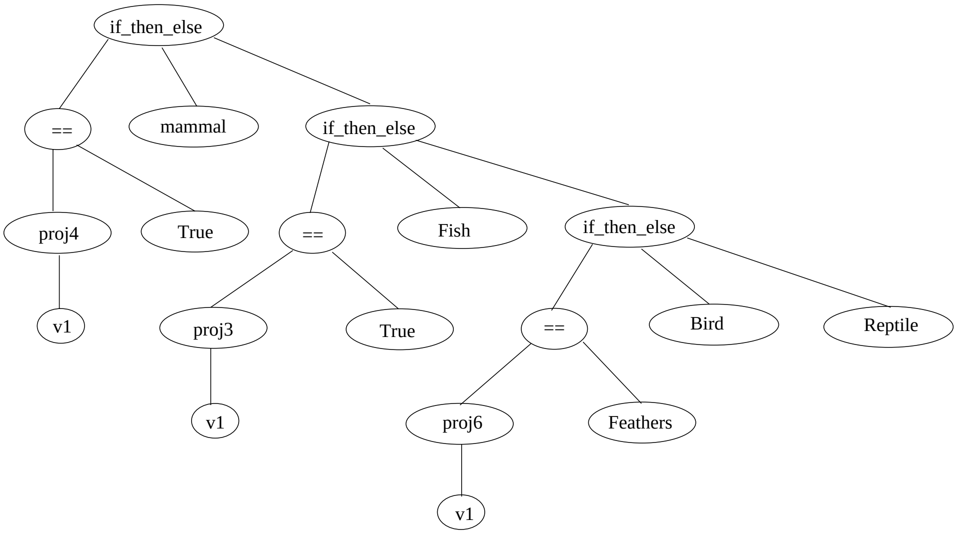
Note, again the connective `||` is left out of the learning alphabet, and hence the AddDisjunction and DropDisjunction operators not used, as it was found not to be necessary to the solution.

Results

The experiment was carried out for 30 runs with an optimal solution found in each run. The optimal solution was found in an average of 8.6 generations or 2580 fitness evaluations. An optimal solution was found in the first generation in one particular run. The average size of the solutions found was 25.3 nodes. Figure 4.4 is the optimal solution found in one of the runs in tree and Escher code form. In plain English this solution reads as "*If an animal has milk then it is a mammal, otherwise if it has gills then it is a fish, otherwise if it has a covering of feathers then it is a bird, otherwise it is a reptile.*" Each run took between 1 minute and 8 minutes to complete.

4.5 Learning Strategies

Among the many parameters of an evolutionary algorithm that have to be initialised before the start of a run are the various probabilities at which the genetic operators will



```
class(v1) = if proj4(v1) == True
               then Mammal
               else if proj3(v1) == True
                       then  Fish
                       else if proj6(v1) == Feathers
                               then Bird
                               else Reptile;
```

Figure 4.4: An optimal solution to the Animal Class problem in tree and Escher code form

be applied. These parameters are often experimented with over a number of preliminary runs before a desirable balance of probabilities is obtained. Once the probabilities have been set they remain static throughout the run. However it has been discovered that in order to carry out the necessary exploration of the search space at the beginning of a run with the exploitation of the search space later on in the run, the probabilities with which the genetic operators are to be applied must change (Rosca and Ballard 1995).

There have been several approaches where the probabilities with which to apply the genetic operators have been adapted dynamically throughout the run in the Genetic Algorithms literature. In the main they focus on identifying genetic operators that are producing good offspring and increasing the probability that these operators will be applied. For Davis' *adaptive operator fitness* each new offspring's parents and the operator that created it are recorded (Davis 1989). When a new offspring is evaluated to be fitter than the fittest chromosome of the current population, credit is given to the operators that created it and the operators that created its parents etc. Once a certain number of offspring have been generated, the probabilities with which to apply the genetic operators are updated as a weighted sum of their current value and the credits they have accumulated since the last update. In (Srinvas and Patnaik 1994) each chromosome has its own probability that crossover and mutation will be applied to it. These probabilities are larger when the mean fitness is near the maximum and smaller for chromosomes with larger fitnesses, i.e. the operators are aggressively applied when the population is approaching convergence but at the same time avoiding replacing the fitter chromosomes. The *Adaptive Operator Probabilities* (ADOPP) approach taken in (Julstrom 1995) is a more sophisticated version of Davis' approach. An operator tree is maintained for each new offspring going back a set number of generations recording the operators that have contributed to the creation of the chromosome. The operator tree is used to compute the credit for the operators when an improved chromosome is generated. A queue of the most recent chromosomes and the credits associated with each of the genetic operators at the time of their creation is maintained. The probabilities of the application of the genetic operators are updated after each new offspring is created from the credit values and the number of each type of operator in the queue.

The approach taken here is to choose an appropriate genetic operator according to a particular learning strategy. During evolution, as an alternative to picking a genetic operator according to a particular distribution, STEPS allows the strategic choice of the genetic operator to be based on the genetic material of the individual that is randomly selected from the population. Two distinct learning strategies made available to STEPS through its specialised genetic operators will be described and evaluated. These two strategies are combined to produce a Hybrid strategy. The Hybrid learning strategy is then be compared to the two learning strategies from which it is combined.

4.5.1 Coverage Strategy

The first adaptive learning strategy available to STEPS is the Coverage strategy. The Coverage strategy is inspired by one of the traditional approaches to concept learning. Concept learning is often viewed as a heuristic search through a state space of possible concept descriptions (Nilsson 1980). The operators that allow the state space to be searched are generalisation and specialisation rules (Michalski 1983).

In general an induced theory is considered to require generalising if it doesn't *cover* a number of the positive instances of the concept to be learnt. This means that the concept description is not *complete* with respect to the training instances. In contrast to this, an induced theory is considered to require specialising if it *covers* a number of negative instances of the concept. In this case the concept description is not *consistent* with respect to the learning examples. For a concept description to be acceptable it must satisfy both the completeness and consistency requirements. However in the real world, data often contains noise and to completely satisfy these requirements is impractical. In this case both requirements are satisfied as closely as possible i.e. the aim is to cover as many positive instances as possible while covering as few negative instances as possible.

In (Michalski 1983) a number of generalisation operators are given. The most suitable operators for the concept learning setting are the *dropping condition rule* where a concept description can be generalised by removing a conjunctively linked expression, and the *adding alternative rule* where a concept description can be generalised by adding an alternative expression through the use of logical disjunction. In a program tree setting these rules are analogous to the DropConjunction AddDisjunction mutation operators available to STEPS for concept learning. In addition the effect of the second generalisation rule can be obtained through the OrCrossover operator. The appropriate rules for specialising a concept description are the reverse of these rules, namely removing a disjunctively linked expression and adding an alternative expression through the use of logical conjunction. These rules are analogous to the DropDisjunction and AddConjunction mutation operators used by STEPS and the effect of the second rule can be achieved through the AndCrossover operator.

The learning systems GABIL (DeJong *et al.* 1993) and GIL (Janikow 1993) both use their specialised operators in an adaptive manner. GABIL achieves this by adding two control bits to each chromosome, one for each of the specialised operators. If an operators' bit is switched on (i.e., it has the value 1) then the operator is applied to the chromosome. The control bits are evolved along with the rest of the chromosome. GABIL's approach to adapting the probabilities of the application of the operators is more complicated. Each individual operator has a probability of application associated with it. During learning these probabilities are adapted according to the coverage of the components of the current individual and the rate of reproduction for the current generation. If the current rate of reproduction is considered to be too high, then the probabilities of all genetic operators are increased by a fraction, conversely if the rate is considered to be too low then the probabilities are decreased by a fraction. In addition to this the probabilities of generalising operators are increased for application to incomplete structures (i.e., rule sets, rules and features), while the probabilities for specialising operators are increased for those structures that are inconsistent.

The STEPS Coverage strategy works by allowing a randomly selected program tree to select its own genetic operator according to its coverage of the training examples. For each individual in the population, the relative number of instances misclassified as the default value class (in the template) and the relative number of instances misclassified as a non-default class is recorded. If for an individual the number of instances misclassified as the default class is greater than the number of instances misclassified as the non-default class (i.e. the theory is relatively more incomplete than it is inconsistent) then the theory is considered to need generalising so one of the generalising operators (DropConjunction, AddDisjunction, or OrCrossover) is selected at random to be applied to the individual. If on the other hand the number of instances misclassified as non-default is greater than the number of instances misclassified as default (i.e. the theory is relatively more inconsistent than it is incomplete) then the concept description is considered to require specialising. In this case one of the specialising genetic operators (DropDisjunction, AddConjunction, or AndCrossover) is selected at random and applied to the individual. However if the number of instances misclassified as the default class is the same as the number of instances misclassified as the non-default class, then a genetic operator is selected from any of the genetic operators available to STEPS for that run.

4.5.1.1 Evaluation

In order to evaluate the STEPS coverage strategy (Cover) its performance on the simple problems from section 4.4 is compared to that of GP (i.e., crossover as the sole genetic operator) and the STEPS basic learning strategy (*Basic* - where any of the specialised genetic operators can be randomly selected by each individual). The Basic and Cover approaches both apply mutation operators to newly created offspring to ensure that all program trees in a population are unique.

During the experiments the parameters used are the same for each of those used for the problems in section 4.4. The experiments were carried out 30 times for each of the three learning approaches. For each run, Tournament selection was used to select individuals from a population of size 300. Predictive accuracy (the number of correctly classified examples divided by the total number of examples) was used as the fitness evaluation and for each problem the optimal solution was defined as a program tree with a predictive accuracy of 100% on the training data.

Tennis

Table 4.1 gives the results for the comparison runs for the Cover strategy for the playTennis problem. The number in brackets indicates the percentage of runs in which an optimal solution was found within the maximum number of generations (set to 60). The average number of generations, average size of solutions, and average number of fitness evaluations are calculated over successful runs only (i.e. runs in which an optimal solution was found). The results show that the GP approach was only able to find an optimal solution in a third of its runs, while an optimal solution was found in every run for the Basic and Cover approaches. The cover approach found a solution in, on average, approximately 25% to 30% fewer generations than the other two approaches. Each run took between 2 minutes and 25 minutes to complete.

Table 4.1: Average No. of Generations for Tennis

STRATEGY	AV. NO. GENS.	AV. SIZE	AV. NO. FIT. EVALS.
GP	12.6 (33%)	39	3780
Basic	13.9 (100%)	40.5	4170
Cover	9.53 (100%)	39.7	2859

Trains

The results for the comparison runs for the Cover strategy for the Trains problem are provided in Table 4.2. The results show that again the GP approach only finds an optimal solution in just under a third of its runs, however when it does find a solution it finds it fairly early on in the run. Again the Basic and Cover approaches found an optimal solution in every generation with the Cover approach finding the optimal solution in, on average, fewer generations than the Basic approach. Each run took between 20 seconds and 20 minutes to complete.

Table 4.2: Average No. of Generations for Trains

STRATEGY	AV. NO. GENS.	AV. SIZE	AV. NO. FIT. EVALS.
GP	4.78 (30%)	21.3	1434
Basic	9.43 (100%)	32.4	2829
Cover	7.13 (100%)	35.4	2139

Animals

Table 4.3 gives the results for the comparison runs for the Cover strategy for the Animals problem. For this problem all three approaches find an optimal solution in all 30 runs. On this occasion The GP approach finds the solution in, on average, fewer generations than with the Cover approach. This is due to the fact that the optimal solution doesn't contain any conjunctions even though this function is placed in the alphabet. The Cover approach relies heavily on there being conjunctions in the search space as the main operators it employs for this problem are the AddConjunction and DropConjunction mutation operators. Each run took between 1 minute and 12 minutes to complete.

Table 4.3: Average No. of Generations for Animals

STRATEGY	AV. NO. GENS.	AV. SIZE	AV. NO. FIT. EVALS.
GP	5.93 (100%)	21.8	1779
Basic	8.6 (100%)	25.3	2580
Cover	11.76 (100%)	25.8	3528

Discussion of Results

Results show that for two of the problems carried out, the Cover learning strategy gives an advantage over the other two approaches in terms of a fewer average number of runs being required to find the optimal solution. The Cover approach helps to focus the search exploiting the nature of the problem being solved. However, the search is biased towards solutions where the number of *False Positives* (i.e. the number of instances misclassified as the non-default class) is equal to the number of *False Negatives* (i.e. the number of instances misclassified as the default class). In addition, the strategy relies on there being connectives in the search and solution space - but this is not an unreasonable assumption as most problems are complicated enough that their solution is made up of a combination of attributes and characteristics. The poor results for the GP approach are due to the irrecoverable loss of genetic material as a result of the single direction of the crossover operator compounded by the lack of mutation operators to reintroduce the diversity.

4.5.2 Depth Strategy

Most evolutionary algorithms rely on fixed-length representations. Clearly, such representations simplify implementations. However, they often require the user to have some knowledge of the appearance and structure of the final solution. More recently, variable-length representations have been used to alleviate these limitations. Although more flexible and less demanding of prior knowledge, the variable-length representation can lead to a general increase in the depth and size of the individuals in the population over successive generations.

This increase in size is called *bloat* and is due to *introns*, extra pieces of information/code that do not contribute anything to the fitness of the individual (Angeline 1994, Blickle and Thiele 1994). The emergence of introns has been attributed to a number of factors. The primary cause is thought to be due to the destructive effects of structure altering genetic operators. Although not useful in terms of fitness, introns seem to protect fit pieces of code (Angeline 1996). Unfortunately, introns tend to spread throughout the population by *Hitch-Hiking* along with the good pieces of code they are protecting during evolution (Tackett 1994). Other explanations for bloat include the use of fitness based selection (Langdon and Poli 1997, Langdon and Poli 1998), and more recently *removal bias*, the notion that offspring are at least as fit as their parents are on average bigger than their parents, causing a steady growth of individuals (Langdon *et al.* 1999). Bloat due to removal bias can be avoided by designing *size fair* genetic operators that modify individuals by inserting a new subtree who's size is, on average, the same size as the subtree it is to replace.

If left unchecked, introns may allow the individuals in the population to bloat uncontrollably, thus putting a considerable strain on the computational resources. Consequently, it is generally agreed that some form of restriction on the depth and/or size of the trees generated during evolution has to be incorporated into the system.

In general, size restrictions are implemented statically in a somewhat *ad hoc* way, for example, a size cut-off or maximum depth value is used in Genetic Programming. These parameters are set *a priori* by the user. Within an evolutionary paradigm, there is something rather unnatural about such hard-coded bounds. It would be more elegant (and more in keeping with the philosophy of evolutionary computing) to let solutions grow or shrink according to the demands of the environment, as measured by the fitness function.

STEPS provides such flexibility through the use of its specialised genetic operators. Large/deep trees are allowed in the population, thus preserving potentially good genetic material that would otherwise be lost if a hard bound on tree size/depth were used. However, large trees have a higher probability of being pruned than smaller trees, whilst small trees have a higher probability of being grown.

There are three main types of restriction for preventing the unbounded growth of individuals in the evolving population. These include aborting offspring if they are greater than a specified depth or size, editing the extra non-contributing code in the individuals, and penalising large individuals through the fitness function.

Specifying a maximum depth cut-off is the most common approach to restricting the growth of trees during evolution in GP. It works by putting a limit on the depth (the longest path from the root of a tree to any of its leaf nodes) or size (the number of nodes) of an individual. Once an offspring has been obtained, its depth (or size) is measured. If its depth (or size) exceeds the maximum allowed depth (or maximum allowed size) then it is considered to be illegal and is not placed into the next generation. Instead a copy of its parent is placed into the population or the genetic operator is re-applied until a legal offspring is produced.

Another method for restricting the size of the variable length individuals is to remove or edit from the individual the extra pieces of code that are not contributing to its fitness. These extra pieces of code can be calculated *a priori* from properties of the function set (Soule *et al.* 1996). Deleting Crossover is a variation of this method (Blickle 1996). It involves marking the parts of the code traversed during evaluation and removing the unmarked parts of the code as they are redundant and do not contribute to the individual's fitness.

The idea behind *Parsimony Pressure* is to penalise large programs. A penalty proportional to the size of an individual is incorporated into the fitness function. Therefore larger trees will have a lower fitness value providing a bias towards smaller solutions. For example in (Zhang and Mühlenbein 1996) the Minimum Description Length principle is used to adaptively balance accurate and parsimonious trees according to the accuracy and complexity of the current best individual.

When the cut-off depth control method is used in conjunction with crossover as the main genetic operator, it can lead to a loss of diversity of genetic material which can cause the population to converge on a suboptimal solution (Gathercole and Ross 1996). In addition to this, it is difficult to identify which value to set the cut-off limit at. If it is too big then memory and CPU time is wasted - too small and the solution will never be found. It is an unnatural and harsh way to control depth and is not in keeping with the theme of natural evolution.

Editing explicitly removes the introns that are protecting the good pieces of code from the destructive effects of crossover and other genetic operators. The removal of the introns is computationally expensive and removes the protection exposing the good code and thus making it vulnerable to destruction. In addition to this, these ineffective blocks of code can be modified during the evolutionary process into useful pieces of code.

Parsimony pressure favours smaller solutions but selecting the correct pressure bias to apply is difficult to determine. In addition to this it is not always possible to use parsimony pressure without an explicit depth restriction such as maximum depth cut off (Gathercole 1998).

The depth controlling strategy works by allowing a randomly selected program tree to select its own genetic operator according to its depth. If the depth of the program tree is greater than the specified maximum depth, then the tree is considered to be too big so a mutation operator that is likely to reduce the size of the tree (i.e., by dropping a disjunction or a conjunction) is chosen to modify the tree. If the depth of the selected tree is less than the minimum specified depth, then the tree is considered to be too small so a mutation operator that is likely to increase the size of the tree (i.e., by adding a disjunction or a conjunction) is chosen to modify the tree. If the depth of the program tree lies within the specified depth constraints then any genetic operator can be randomly selected to modify it.

This strategy is a depth controlling strategy used to keep the size of the program trees under control rather than allowing the trees to grow in an unconstrained manner. This provides a more flexible method for controlling the depth of the tree. If a tree is considered too big it is not thrown away, but its size is reduced giving any good genetic material that it may contain a chance to survive.

4.5.2.1 Evaluation

In order to evaluate the STEPS depth controlling strategy (*Depth*) its performance on the simple problems from section 4.4 is compared to that of GP and the STEPS basic learning strategy (Basic). Both the GP and Basic approaches use max depth cut off in order to prevent the uncontrollable growth of the program trees. The Basic and Depth approaches both apply mutation operators to newly created offspring to ensure that all program trees in a population are unique.

During the experiments the use of solution structure knowledge such as minimal required alphabet for optimum solution was avoided. This leads to a sub-optimal performance of the algorithm, but gives more realistic conditions as for real world problems such information is not available. Therefore for each problem the alphabet consists of the selector function subtrees, all available connectives (conjunction, disjunction and negation) regardless of whether they were necessary for the solution, and the problem dependent constants.

For each problem, each learning approach was carried out with a large maximum depth parameter and then with a small maximum depth parameter. The large maximum depth parameter was set to 15 - sufficiently big enough to find the known optimal solution for each problem. The small maximum depth parameter's value varied with each problem. It was set to be small enough so that it was smaller than the depth of the known optimal solution but large enough so that complete program trees could be generated. The experiments were carried out 30 times for each of the three learning approaches. For each run, Tournament selection was used to select individuals from a population of size 300. Predictive accuracy was used as the fitness evaluation and for each problem the optimal solution was defined as a program tree with an error of 0.0.

Tennis

The experiments were first carried out with the maximum depth parameter set to 15 and then repeated with the maximum depth parameter set to 5. The results are expressed as the average number of generations taken to find an optimal solution (for successful cases), the average size of the solutions found, and the average number of fitnessevaluations carried out, in Table 4.4 for the maximum depth of 15 and Table 4.5 for the maximum depth set to 5. The number in brackets indicates the percentage of runs in which an optimal solution was found within the maximum number of generations (set to 60). A '-' indicates that an optimal solution was not found in any of the runs carried out. Each run took between 1.5 minutes and 25 minutes to complete.

Table 4.4: Average No. of Generations for Tennis, Max-Depth = 15

STRATEGY	AV. NO. GENS.	AV. SIZE	AV. NO. FIT. EVALS.
GP	16.69 (87%)	75.4	5007
Basic	11.17 (100%)	54.9	3351
Depth	12.97 (100%)	67.6	3891

Table 4.5: Average No. of Generations for Tennis, Max-Depth = 5

STRATEGY	AV. NO. GENS.	AV. SIZE	AV. NO. FIT. EVALS.
GP	- (0%)	-	-
Basic	- (0%)	-	-
Depth	39.41 (57%)	84.33	11823

Trains

The experiments were carried out first with the maximum depth parameter set to 15 and then repeated with the maximum depth parameter set to 6. The results are expressed in Table 4.6 for the maximum depth set to 15 and in Table 4.7 for the maximum depth set to 6. Each run took between 20 seconds and 20 minutes to complete.

Table 4.6: Average No. of Generations for Trains, Max-Depth = 15

STRATEGY	AV. NO. GENS.	AV. SIZE	AV. NO. FIT. EVALS.
GP	8.71 (93%)	47.1	2613
Basic	6.03 (100%)	60.6	1809
Depth	6.5 (100%)	62.8	1950

Animals

The experiments were first carried out with the maximum depth parameter set to 15 and then with the maximum depth parameter set to 5. The results are expressed in Table 4.8 for the maximum depth set to 15 , and in Table 4.9 for the maximum depth set to 5. Each run took between 1.5 minutes and 20 minutes to complete.

Discussion of Results

The results show that for a large maximum depth there is no real difference between the

Table 4.7: Average No. of Generations for Trains, Max-Depth = 6

STRATEGY	AV. NO. GENS.	AV. SIZE	AV. NO. FIT. EVALS.
GP	- (0%)	-	-
Basic	37 (7%)	44	11100
Depth	12.31 (97%)	37.1	3693

Table 4.8: Average No. of Generations for Animals, Max-Depth = 15

STRATEGY	AV. NO. GENS.	AV. SIZE	AV. NO. FIT. EVALS.
GP	7.167 (100%)	32.5	2151
Basic	9.133 (100%)	46.5	2739
Depth	10.1 (100%)	52.4	3030

performance of the Depth and Basic approaches. Both perform slightly better than GP on the Tennis and the train problem. However for the Animals problem, it is the GP approach that performs slightly better than the other two approaches, finding a solution on average in fewer generations. When the number of generations evolved in each run (regardless of whether an optimal solution is found) for each of these approaches is statistically analysed using the Anova test (as have three strategies to compare) in conjunction with the Tukey post hoc comparison (in order to obtain the each of the pairwise comparisons of the three strategies) the difference performance between the Depth and the GP approaches and between the Basic and GP approaches is found to be statistically significant at the 99% level for both the Tennis and Train problem at this depth setting. The difference in performance of the Depth and Basic approaches is not statistically significant For all three problems.

On runs with a small maximum depth the Basic and GP approaches never find a solution for the Tennis problem, whereas the Depth control approach finds a solution in over half the runs for the Tennis problem and in every run for the Animals problem and all but one run for the Michalski's train problem. The Basic approach is able to find a solution in 2 out of 30 of its runs for the Michalski's train problem with the maximum depth threshold set to 6, and 4 out of 30 of its runs for the Animals problem with the maximum depth threshold set to 5. This is due to the application of the mutation operators to newly

Table 4.9: Average No. of Generations for Animals, Max-Depth = 5

STRATEGY	AV. NO. GENS.	AV. SIZE	AV. NO. FIT. EVALS.
GP	- (0%)	-	-
Basic	22.5 (13%)	23.0	6750
Depth	8.3 (100%)	34.3	2490

created offspring. The GP approach never finds the solution if the depth bound is set too small. Not surprisingly, when the number of runs carried out are analysed in the same manner as above for the small depth setting, the difference in performance between the Depth and the Basic strategies and the Depth and GP strategies is statistically significant at the 99% level.

The basic idea behind the STEPS depth controlling strategy is to grow or shrink trees to fit the problem. Consequently, no real advantage is gained by using this strategy when a large maximum depth is specified. On the other hand, such large bounds often lead to inefficiency. The STEPS depth controlling strategy allows experienced users to retain efficiency without jeopardising the chances of finding a (larger) optimal solution by specifying conservative depth restrictions. The system will compute efficiently within these restrictions and only incur additional computational costs if these are strictly necessary to produce a better solution. In addition, the system also becomes more robust since the uninformed guesses of inexperienced users do not hinder its capacity to find an optimal solution.

Figure 4.5 illustrates that while the average depth of the program trees are kept to (approximately) within the depth constraints, the depth of the fittest trees are allowed to grow beyond this threshold as necessary.

4.5.3 Hybrid Strategy

The final learning strategy available to STEPS is a hybrid combination of the previous two learning strategies Cover and Depth. The basic idea behind this strategy is to utilise the focus of the Cover strategy in combination with the dynamic flexibility of the Depth strategy thus obtaining the benefits from both approaches in one strategy.

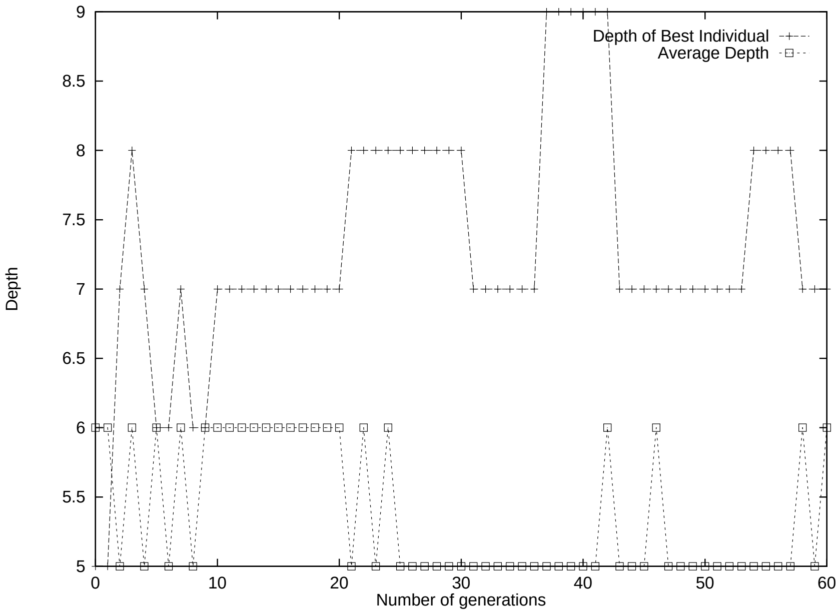


Figure 4.5: A graph of the depth statistics from a run using the STEPS' depth monitoring strategy with maximum depth set to 5

The algorithm for the hybrid strategy is presented in Figure 4.6. Once an individual has been selected as a parent from the current population, the first thing to consider is its depth in order to keep exponential growth under control by applying the Depth learning strategy. If the depth of the program tree lies within the specified depth constraints then the next thing we consider is the individual's coverage of the training examples by applying the Cover learning strategy. However if the number of instances misclassified as the default class is the same as the number of instances misclassified as the non-default class, then a genetic operator is selected from any of the genetic operators available to STEPS for that run is selected at random and applied to the individual.

4.5.3.1 Evaluation

In order to evaluate the STEPS hybrid strategy (Hybrid) its performance on the simple problems from section 4.4 is compared to that of the Cover and Depth learning strategies from which the Hybrid strategy is amalgamated. All three approaches apply mutation operators to newly created offspring to ensure that all program trees in a population are

```
**Input**: An program tree $T$
**Output**: A genetic operator $G$ to be applied to $T$
if $depth(T) > maxdepth$
  then $G = select\{DropConjunction, DropDisjunction\}$;
  else if $depth(T) < mindepth$
      then $G = select\{AddConjunction, AddDisjunction\}$;
      else if $ndefaultmisclass > defaultmisclass$
          then $G = select\{DropConjunction, AddDisjunction, OrCrossover\}$;
          else if $ndefaultmisclass < defaultmisclass$
              then $G = select\{AddConjunction, DropDisjunction, AndCrossover\}$;
              else $G = select\{$ $AddConjunction, DropDisjunction, AndCrossover,$
                              $DropConjunction, AddDisjunction, OrCrossover,$
                              $Crossover, FunctionalMutation, TerminalMutation\}$;
```

Figure 4.6: The algorithm for the Hybrid learning strategy

unique.

In order for a fair comparison to be carried out, the experimental settings that give both the Depth and the Cover strategies an advantage are utilised during the experiments. In this case three sets of experimental settings are used. The first setting is the same setting used for the Cover strategy comparisons in section 4.5.1.1. This means that during the experiments the parameters used are the same for each of those used for the problems in section 4.4. There is some domain knowledge used about the structure of the optimal solution in the form of an adequate maximum depth parameter and a minimal alphabet. The other two settings are the experimental settings used for the Depth strategy comparisons in section 4.5.2.1. Therefore the alphabet consists of the selector function subtrees, all available connectives (conjunction, disjunction and negation) regardless of whether they were necessary for the solution, and the problem dependent constants. Then with this maximal alphabet, each learning strategy is carried out with a large maximum depth parameter and then with a small maximum depth parameter. The large maximum depth parameter is set to 15 - sufficiently big enough to find the known optimal solution for each problem. The small maximum depth parameter's value varies with each problem.

The experiments were carried out 30 times for each of the three settings for each of the three learning approaches. For each run, Tournament selection was used to select individuals from a population of size 300. Predictive accuracy was used as the fitness evaluation and for each problem the optimal solution was defined as a program tree with a predictive accuracy of 100% on the training data.

Tennis

The results are expressed in the same format as the previous results, in Table 4.10 for the maximum depth of 15, Table 4.11 for the maximum depth of 8 and Table 4.12 for the maximum depth set to 5. Each run took between 1.5 minutes and 25 minutes to complete.

Table 4.10: Average No. of Generations for Tennis, Max-Depth = 15

STRATEGY	AV. NO. GENS.	AV. SIZE	AV. NO. FIT. EVALS.
Cover	8.57 (100%)	61.23	2571
Depth	12.96 (100%)	67.6	3888
Hybrid	8.97 (100%)	60.7	2691

Table 4.11: Average No. of Generations for Tennis, Max-Depth = 8

STRATEGY	AV. NO. GENS.	AV. SIZE	AV. NO. FIT. EVALS.
Cover	9.53 (100%)	39.7	2859
Depth	14.33 (100%)	47.33	4299
Hybrid	9.3 (100%)	46	2790

Table 4.12: Average No. of Generations for Tennis, Max-Depth = 5

STRATEGY	AV. NO. GENS.	AV. SIZE	AV. NO. FIT. EVALS.
Cover	- (0%)	-	-
Depth	39.41 (57%)	31.82	11823
Hybrid	37.35 (57%)	34.23	11205

The results show that when the maximum depth is set to 15, all three approaches find an optimal solution in all of their 30 runs. However both the Cover and the Hybrid approaches find the solution in, on average, 25% fewer generations than the Depth approach. This is also the case for the situation when the approximate alphabet and depth is known. However when the maximum depth is small, the Cover approach is unable to find a solution and the Depth and Hybrid approaches both perform equally well finding the optimal solution in 57% of their runs.

Trains

The results are expressed in Table 4.13, Table 4.14 and Table 4.7 respectively. Each run took between 20 seconds and 20 minutes to complete.

Table 4.13: Average No. of Generations for Trains, Max-Depth = 15

STRATEGY	AV. NO. GENS.	AV. SIZE	AV. NO. FIT. EVALS.
Cover	4.4 (100%)	63.5	1320
Depth	6.5 (100%)	62.8	1950
Hybrid	4.56 (100%)	70.8	1368

Table 4.14: Average No. of Generations for Trains, Max-Depth = 8

STRATEGY	AV. NO. GENS.	AV. SIZE	AV. NO. FIT. EVALS.
Cover	7.13 (100%)	35.4	2139
Depth	6.97 (100%)	35.9	2091
Hybrid	4.6 (100%)	39.7	1380

Table 4.15: Average No. of Generations for Trains, Max-Depth = 6

STRATEGY	AV. NO. GENS.	AV. SIZE	AV. NO. FIT. EVALS.
Cover	41 (13%)	37.8	12300
Depth	12.31 (100%)	37.6	11286
Hybrid	11.63 (100%)	36	10800

The results tell a similar story to the results for the Tennis problem. When the maximum depth is set to 15, all three approaches find an optimal solution in all of their 30 runs, both the Cover and the Hybrid approach find the solution in, on average, fewer generations than the Depth approach. When the maximum depth is reduced to 8 and the minimal alphabet for the problem is know then all three approaches, again, find an optimal solution in all of its runs, with the Hybrid approach finding the solutions in, on average, fewer runs. When the alphabet is increased again and the maximum depth is further reduced to 6 the Cover approach is only able to find an optimal solution in 13% of its runs. Both the Depth and the Hybrid approaches are able to find an optimal solution in all of their runs, obtaining the solution in about the same number of generations.

Animals

The results are expressed in Table 4.16, Table 4.17, and Table 4.18 respectively. Each run took between 1.5 minutes and 20 minutes to complete.

Table 4.16: Average No. of Generations for Animals, Max-Depth = 15

STRATEGY	AV. NO. GENS.	AV. SIZE	AV. NO. FIT. EVALS.
Cover	8.37 (87%)	59.7	2511
Depth	10.1 (100%)	52.4	3030
Hybrid	8.17 (100%)	54.1	2451

Table 4.17: Average No. of Generations for Animals, Max-Depth = 8

STRATEGY	AV. NO. GENS.	AV. SIZE	AV. NO. FIT. EVALS.
Cover	11.77 (100%)	25.8	3531
Depth	11.07 (100%)	27.7	3231
Hybrid	11.7 (100%)	29.7	3510

Table 4.18: Average No. of Generations for Animals, Max-Depth = 5

STRATEGY	AV. NO. GENS.	AV. SIZE	AV. NO. FIT. EVALS.
Cover	19 (3%)	19	5700
Depth	10.17 (100%)	34.3	3051
Hybrid	11.07 (100%)	35.8	3321

For the Animals problem, when the maximum depth is set to 15, the Cover approach is able to find a solution in only 87% of its runs. This is due to the lack of connectives (conjunctions and disjunctions) in the solution space. The Hybrid and Depth approaches are both able to find an optimal solution in all of their runs with the Hybrid approach obtaining its solution in slightly fewer generations. However when the maximum depth is set to 7 and the alphabet is reduced then all three approaches perform at about the same level. When the maximum depth is set to 5, the Cover approach only finds an optimal solution in one of its runs. The Depth and the Hybrid approach both find an optimal solution in all of their runs in about the same number of generations.

Discussion of Results

The results comparing the Hybrid learning strategy to the two strategies from which the strategy is composed, Depth and Cover, shows a general trend throughout all three experiments. It appears that the Hybrid approach performs at least as well as the Cover strategy approach when an adequate maximum depth for tree growth is specified. The difference in performance (in terms of the number of generations carried out) of the Hybrid and Cover approaches are not statistically significant at the large or normal depth setting when analysed in the same manner as the previous section, for all three problems, except for the trains problem with a normal depth setting. In addition to this, it performs at least as well as the Depth strategy when an insufficient maximum depth value is specified. Again there is no statistical significance between the performance of the two strategies at this depth setting. The Hybrid approach seems to have the ability to exploit the situations where the two strategies from which it is composed have an advantage. This is to be expected as when the depth of a program tree lies between the specified depth constraints, the hybrid strategy behaves as the cover strategy and when the depth of a program tree lies outside these depth restrictions, the hypbrid strategy behaves as the depth strategy. Therefore during runs with a small maximum depth specified, the hybrid strategy will behave like the depth strategy the majority of the time, and during runs with a large maximum depth specified, the hybrid strategy will behave like the cover strategy the majority of the time.

4.5.4 Discussion of Strategies

The specialising and generalising approach of the Cover strategy appears to give an advantage in situations where both the search and the solution space contain connectives and an adequate maximum depth has been specified. The Cover strategy seems to allow the search to focus, taking advantage of the nature of the problem space, providing a more efficient convergence (in terms of number individuals generated and evaluated) to the optimal solution. However if the maximum depth parameter is set at insufficient level then this approach will rarely find the solution.

The basic idea behind the STEPS depth controlling strategy is to grow or shrink trees to fit the problem. Consequently, no real advantage is gained by using this strategy when a large maximum depth is specified.

The preliminary experiments demonstrate that the Hybrid approach performs as well as the other two approaches in their respective speciality settings making it an ideal *all rounder* learning strategy. Further experimentation is necessary to determine the full implications of the approach. In particular experiments will be carried out on some more substantial, real world problems in the next chapter.

Chapter 5

Predicate Invention

5.1 Introduction

There are many connections and parallels between the motivations for Predicate Invention used in Inductive Logic Programming, and the use of Automatically Defined Functions and λ-abstractions, used in conjunction with evolutionary learning systems. However, the author is not aware of any previously established connections between the the techniques from the two different learning settings.

During learning, language biases are applied through the hypothesis language (see chapter 2) to restrict the number of expressible hypotheses in the generally huge search space. If the biased hypothesis language is unable to express a hypothesis that is consistent with the training examples then the hypothesis language must be altered to compensate. In the propositional learning setting, *Constructive Induction* is applied to the hypothesis language in order to create additional features (Matheus 1991, Wnek and Michalski 1994). The additional features are to compensate for the missing information about the relationship between the existing features that cannot be directly inferred using the current hypothesis language.

Inductive Logic Programming (ILP) is concerned with learning in a first-order Horn clause setting (Muggleton 1992*b*). Constructive induction in this first-order setting has become known as *Predicate Invention* (Muggleton 1994). Reformulation approaches to predicate invention introduce new predicates to an existing hypothesis in order to compress it, providing the acquired information in a more concise manner.

Automatically Defined Functions (ADFs) is an extension to Genetic Programming (GP) that involves the evolution of extra functions or “modules” that support code reuse, promoting the compression of the main parse tree.

Both automatically defined functions in GP and reformulation approaches to predicate invention in ILP are motivated by a need to compress the program code that results from their corresponding learning processes. Both approaches achieve compression of the program code by learning additional functions or predicates that are to be used in the body of the main program. In this chapter the connection between reformulation predicate invention and the automatically defined functions extension to genetic programming is established. A special case of automatically defined functions, λ-abstractions in the context of strongly typed genetic programming, is discussed in order that the particular approach to predicate invention adopted by STEPS may be introduced.

5.2 Predicate Invention and Inductive Logic Programming

Constructive induction carried out in a first-order setting, and in particular the Inductive Logic Programming (ILP) framework, is known as Predicate Invention (Muggleton 1994). As we saw in chapter 2, in order to represent the knowledge that is gained through learning, a hypothesis language is necessary. In ILP the hypothesis language consists of the predicates, variables, and constants that are used to induce a logical theory that is consistent with the training examples. The resulting hypothesis is sometimes called a *finite axiomisation* of the examples (Ling 1991, Stahl 1993).

Language biases are enforced to restrict the number of expressible hypotheses, thus constraining the search space. In some cases, the biased hypothesis language is insufficient to obtain a finite axiomisation consistent with the examples. In such cases, the hypothesis language needs to be augmented with additional predicates not present in the original hypothesis in order to *shift the bias* imposed on it so that the target predicate may be learnt (Stahl 1995). However, recognising the fact that the hypothesis language is unable to produce a consistent finite axiomisation of the examples is undecidable (Stahl 1993), unless the language and algorithm biases allow the full enumeration of the search space.

Ling distinguishes between two types of invented predicates, namely *necessary* predicates and *useful* predicates (Ling 1991). A newly invented predicate[1] is necessary if

> ". . . without inventing such new terms(s), the theory is not learnable according to the criterion of successful learning."

A newly invented predicate is useful if

> ". . . its invention does not affect the learnability of the theory."

These two types of invented predicates correspond to the two approaches to predicate invention, the *Reformulation* approach and the *Demand-driven* approach, noted by Stahl in her overview paper (Stahl 1993).

5.2.1 Reformulation Approaches

The Reformulation approach to predicate invention introduces new predicates that are a *reformulation* of an existing theory in order that it is compressed. Stahl notes that there are two types of system that carry out the Reformulation approach to predicate invention, Inverse Resolution systems and schema (or scheme) driven systems (Stahl 1993).

5.2.1.1 Inverse Resolution Systems

Systems that are based on inverse resolution and that carry out predicate invention make use of the W or inter-construction and intra-construction operators. The W operators are so called as they combine two V, or inverse resolution, operators back to back. The process of resolution involves getting rid of a literal that occurs in two clauses (as a positive literal in one of the clauses and as a negative literal in the other clause) and combining the remainder of the two clauses to form a new clause. Often substitutions (denoted θ) are required to be applied to each of the original clauses to find the same

[1] In this case Ling refers to newly invented terms which is a generalisation of a predicate.

literal that occurs in both clauses. The basic form of the W operators is illustrated in figure 5.1 taken from (Muggleton 1992*a*). The W operators takes two clauses as its input (B_1, B_2) and constructs the three clauses (C_1, C_2, and A) that would resolve to obtain the original clauses using appropriate substitutions (θ_{C_1}, θ_{A_1}, θ_{A_2}, θ_{C_2}) . The literal that would be resolved on in the constructed clauses (C_1, C_2, and A) doesn't appear in the clauses given to the W operator and so nothing is known about its symbol, so it is in effect invented. If this literal is assumed to be negative then the particular W operator is called intra-construction, otherwise if the literal is assumed to be positive then the W operator is called inter-construction. However there is then the open problem in deciding on the arguments for the newly invented predicate.

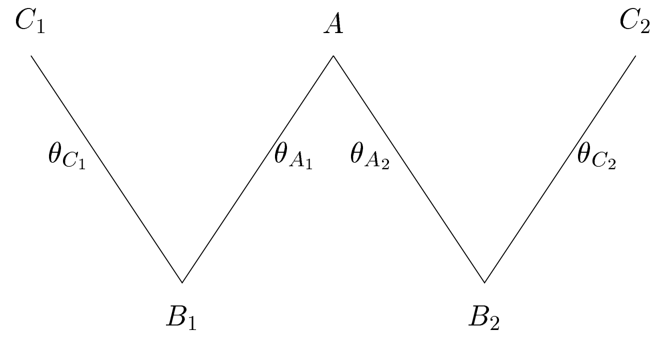


Figure 5.1: The basic form of a W operator: Two resolution steps with a common clause A

Some examples of systems that carry out predicate invention using the W operator include CIGOL (Muggleton and Buntine 1988), LFP2 (Wirth 1989), ITOU (Rouveirol 1992), RINCON (Wogulis and Langley 1989), and Banerji's system (Banerji 1992).

5.2.1.2 Schema-driven Approaches

Learning systems that use the schema-driven approach to predicate invention compress the information provided in an induced theory by combining useful predicates using a schema.

Two such systems that use this schema driven approach to predicate invention are CIA (used in conjunction with the learning system CLINT) (DeRaedt and Bruynooghe 1992) and FOCL (Silverstein and Pazzani 1993). CIA matches clauses against higher-order schemas by instantiating their predicate variables with predicate symbols. Clauses that match a schema are presented to the user, who on finding them useful abstractions, names them. FOCL, an extension of the well known learning system FOIL (Quinlan 1990), considers combinations of literals restricted to relational cliches. These relational cliches are schemas that restrict the number and constituent elements of the literals combined.

5.2.2 Demand Driven Approaches

The demand driven approach to predicate invention attempts to detect possible situations where the hypothesis language is insufficient to produce a hypothesis that is consistent with the training examples. However, as mentioned above, this problem is undecidable unless the search space can be fully enumerated. The demand-driven approaches therefore have to use some kind of heuristic information in making the decision to use predicate invention.

The learning system INPP and its predecessor MENDEL use the size of an induced theory to decide when to apply predicate invention (Ling 1995, Ling 1991). If the size of the theory grows beyond a certain bound then it is assumed that the hypothesis language is insufficient and the intra-construction operator is applied to create a new predicate. SIERES invents a new predicate if none of the existing predicates in the hypothesis language can successfully complete a clause leaving no unused input terms or no unbound output terms (Wirth and O'Rorke 1992). DBC, MOBAL and CWS decide to invent a new predicate in order to specialise an over-general clause. Once the inconsistent clause has been identified, if the existing hypothesis language is unable to specialise it with respect to the training examples then a literal containing a newly invented predicate is used (Wrobel 1994, Srinivasan *et al.* 1992, Kijsirikul *et al.* 1992).

5.2.3 More Recent Approaches to Predicate Invention

The more recent ILP systems have moved away from the inverse resolution technique and are favouring saturation techniques in an attempt to avoid the difficulties associated with inverse resolution when learning recursive clauses (Rouveirol 1994). This move away from inverse resolution also leads to a move away from the convenience of the W operators for inventing new predicates. This appears to be correlated with a decrease in ILP learning systems claiming the ability to carry out predicate invention. In fact, there has been relatively little work in the area of predicate invention in recent years. The work that has been carried out appears to take a different approach to the techniques discussed above.

In (Sutton 1994) the author suggests that constructive induction would be a useful tool when solving a sequence of learning tasks requiring the same representation but different solutions. Solving a sequence of problems of this nature is called continuing learning (Sutton 1992). This idea that constructive induction should be based on continuing learning is adapted and supported by work on predicate invention in (Khan *et al.* 1998). In this work the learning system PROGOL (Muggleton 1995) is used to learn a series of rules for the chess end game domain, and then again with the predicate invention option switched on. It is reported that predicate invention does not seem to improve accuracy, but drastically increases training time. The use of predicate invention in a process analogous to continuing learning, called repeat learning, is then investigated. Intra repeat learning involves using previously invented new predicates for a particular chess piece to learn rules for that piece. Inter repeat learning, on the other-hand, involves using previously invented new predicates for one chess piece to learn the rules for another chess piece. Using the newly invented predicates when carrying out repeat learning is seen to increase accuracy, and reduce training time and rule complexity, supporting Sutton's claim.

PROGOL's mechanism for inventing new predicates involves using constraint-solving. However some of the flexibility seen with other systems with predicate invention capabilities is lost as the arity and types of the arguments must be known before the new predicate can be produced.

Another alternative use for predicate invention is to restructure a relational database

(Flach 1993). The learning system INDEX is used to carry out *inductive data engineering*, which is the *"... interactive process of restructuring a knowledge base by means of induction"*. New predicates are introduced by constructing integrity constraints during the reformulation of an extensional relational database.

5.2.4 Some Issues of Predicate Invention

The depleted interest in predicate invention over the last few years may also be due to its difficulty. The act of inventing new predicates with which to augment the hypothesis language is relatively easy. However, knowing when this act is useful to the learning system is a difficult task to solve. Most learners, particularly those that carry out demand driven predicate invention, tend to use an over general clause that may not be specialised with the existing hypothesis language as an indication that predicate invention is desirable. However this over generality may be due to noise in the training examples in which case inventing new predicates may just lead to overfitting of the data.

Once the decision to use predicate invention has been made there is still the issue of which of the many possible newly invented predicates should be selected to augment the hypothesis language. This is the problem of predicate *utility* (Muggleton 1994). Very little work has been done on this problem and it remains largely an open issue.

5.3 Automatically Defined Functions in Genetic Programming

Automatically Defined Functions (ADFs) are an enhancement to Genetic Programming that allows the Genetic Programming algorithm to exploit the hierarchical nature of a difficult problem (Koza 1992). The divide and conquer technique is often used in problem solving to decompose a difficult problem into more manageable sub-problems. These sub-problems themselves may be difficult to tackle and so are further subdivided. Once all the sub-problems have been solved, their solutions are combined in a hierarchical manner to produce a single solution to the original difficult problem. The divide and conquer

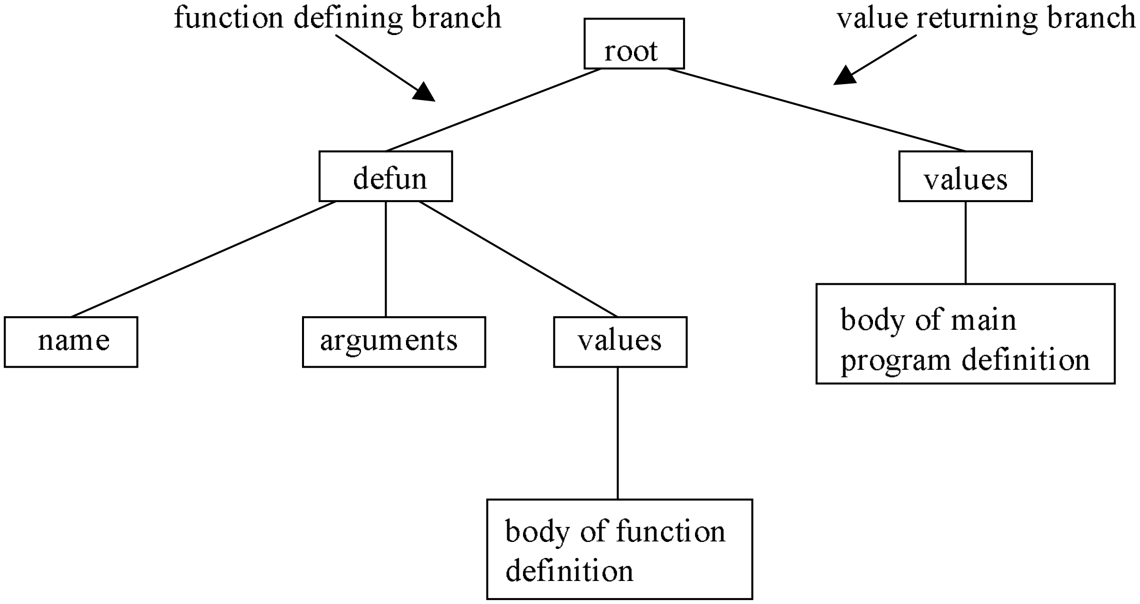


Figure 5.2: The structure of the program tree when using Automatically Defined Functions with Genetic Programming

approach to problem solving is reflected in the use of modules when writing computer programs. Modules or subroutines encapsulate a group of useful functions as a unit so that their combination may be used repeatedly in a main function. This code reuse leads to simpler, more general and shorter programs so that more complex programs may become more manageable.

ADFs allow the use of subroutines or modules in GP. The basic program tree of genetic programming is enhanced to become a structure combining a main program and the subroutines or automatically defined functions that are to be used by the main program. The ADFs and the main program are evolved simultaneously during learning. Figure 5.2 illustrates the structure of the special program tree.

The root of the special program tree has two main types of branches, a value returning branch, of which there is always one, and a function defining branch, of which there may be several. The value producing branch contains the subtree that constitutes the body of the main program. The value that is returned by this subtree is the value that is returned by the program tree. The main program is able to make calls to an ADF defined elsewhere in the program tree by using its name and providing it arguments as with other functions from the function set. Each function defining branch contains the definition for an ADF or subroutine. A complete definition for an ADF consists of its name, an argument list and a subtree that defines the behaviour of the ADF. The variables that appear in the body of the definition for an ADF are local variables and so do not appear in the main program. Consequently the main program definition and each ADF have their own alphabet defined. ADFs are able to make calls to each other enforcing a hierarchical arrangement of function calls, but care must be taken so that the program does not enter an infinite recursive loop during evaluation.

When using ADFs with genetic programming the main structure of the special program tree remains fixed during the evolutionary process. Only the program bodies are manipulated and altered. Therefore the number of automatically defined functions that will be used (i.e., how many function defining branches each tree in the population will have) and the number of arguments that each ADF will take must be known before learning takes place. The crossover operator has to be suitably adapted so that the main structure is preserved. In addition to this, in order to evolve syntactically correct trees, crossover may only take place between subtrees from equivalent branches in different trees. For example, if the node randomly selected in the first parent program tree is in the ADF1 definition subtree, then the node selected from the second parent must also be from its ADF1 definition subtree.

Not only does the use of ADFs produce smaller more general solutions to problems of a hierarchical nature, their use also enables the solution of problems that could not be solved by ordinary genetic programming on its own. For example, the Even N Parity problem with N = 11 (Koza 1994). ADFs have become a hot topic of research in recent years, with a number of variations of it being developed.

Adaptive Representation Learning

Adaptive Representation Learning (ARL) is an alternative to ADFs that abstracts useful subtrees from program trees so that they may be used as a subroutine in future populations (Rosca and Ballard 1996). A useful block of code in an individual is identified by looking at an individual's *differential fitness* compared to its least fit parent and *block activation* (i.e., the number of times a particular block of code is activated during evaluation). Blocks of code with high activity in individuals with high positive differential fitness are generalised by replacing a random subset of terminals with variables and placed as a new subroutine in the subroutine set. All the subroutines in the subroutine set are assigned a *utility* value according to the outcome of its use. Subroutines with low utility are deleted from the subroutine set in order to keep the size of the set below a certain number. The subroutine discovery algorithm is activated when the long term decreases in population diversity occur (as measured using entropy).

Evolutionary Module Acquisition

Evolutionary Module Acquisition provides a general purpose technique for creating modules in any evolutionary based system (Angeline and Pollack 1993). Components of an individual being evolved are chosen at random to be collated into a module by a *compress operation*. Once a module has been formed, it is isolated from further manipulation by the genetic operators. An *expansion operation* can be used to release components from a module in order that they may be manipulated by the genetic operators again. If a module is useful to the problem being solved then it will be passed onto future offspring through fitness based selection. Conversely if the module is not useful, then it may be eliminated from the population. The *atomisation* compression operator allows frozen modules to be compressed into other modules so that a hierarchical organisation of the modules may be created.

Automatically Defined Macros

Spector enhances genetic programming by allowing the algorithm to simultaneously evolve a main program and automatically defined macros that may be used by the main program

(Spector 1996). A Macro is an instruction that is replaced by a sequence of instructions prior to the compilation of the program. They are similar to subroutines in that they allow programs to be modularised, but they have the advantage that can allow new control structures to be implemented through their influence over the evaluation of their arguments. Automatically defined macros have been found to be useful in situations where the function set is not purely functional, i.e., the function set contains functions with side effects. The author suggests that a genetic programming system should be augmented with both automatically defined functions and automatically defined macros as they are both available to the programmer in *the real world* and compliment each other in different programming situations.

”Demand Driven” Automatically Defined Functions

In (Koza *et al.* 1999) the authors note that it is not always possible to determine the architectural choices necessary before automatically defined functions may be evolved (i.e. the number of automatically defined functions required and the number of arguments that each of them should take). In order to overcome this difficulty, the authors present some operators to be applied during learning that have the effect of selecting the architecture of the program structure in an evolutionary manner. The six architecture altering operations include functions that can create, duplicate or delete an automatically defined function (subroutine) or an ADF's parameter. These operators are applied to a small percentage of the current population in conjunction with the standard genetic operators. The idea is for the architecture altering operations to provide the individuals in the population with the ability to develop their own architecture instead of dictating the structure before learning takes place. Thus subroutines and their arguments can be added or deleted as necessary for the particular problem being solved. In this setting, the subroutines are added in a *demand-driven* manner.

In addition to the operators altering the structure of the subroutines, a set of functions allowing iterations, loops, recursion, and storage to be automatically defined and evolved in the same manner as the subroutines is defined. These additional constructs allow genetic programming to be a very flexible, powerful problem solving technique that may be applied to very difficult intractable problems (Koza *et al.* 1999).

5.4 λ-Abstractions and Strongly Typed Genetic Programming

PolyGP is a strongly typed genetic programming system that evolves Haskell programs (Yu and Clack 1998*a*). In (Yu and Clack 1998*b*) the PolyGP system is augmented with the ability to incorporate modules into the evolving programs. The modules in this case take the form of λ-abstractions. A λ-abstraction can be used to denote an "anonymous" function where the name or type is not specified:

$$(\lambda x(*\ x\ x))$$

The above λ-abstraction expresses the function square. These λ-abstraction modules can be reused by passing them as arguments to other functions. Functions that are capable of taking other functions as their arguments are called *higher-order* functions. The higher-order functions used in this system are those of *map* and *fold.* These functions employ *implicit* recursion to apply functions to lists of elements. The use of implicitly recursive functions in evolutionary systems avoids the problems of the evolution of explicit recursion such as requiring a base case and non-terminating loops due to the convergence away from the base case of the function. These problems are automatically solved by incorporating them into the definition of the implicitly recursive function. The *map* function repeatedly applies the same function to every element in a list:

$$map(\lambda x(*\ x\ x))[1,2,3] = [1,4,9]$$

The *fold* function places a function requiring two arguments between each element of a list collapsing it into a single element. *Fold* has two varieties, *foldl* and *foldr* depending on whether the resulting expression is associated to the left or to the right:

$$foldl(\lambda xy(+xy))0[1,2,3] = (((0+1)+2)+3) = 6$$

The second argument is a value to be used when the list is empty.

The λ-abstractions are evolved as arguments to the higher-order functions in the main body of the program structure. They therefore evolve dynamically and as with demand-driven automatically defined functions, they are not a prefixed condition of evolution, but are evolved if helpful to the solution. The number and type of arguments of a λ-abstraction are implicitly decided by the type signature of the higher-order function of which they are a functional argument.

The combination of higher-order functions with λ-abstractions provides a sort of structure abstraction as the combination is preserved by the type system, and the structure preserving crossover operator employed (see section 5.3). Crossover between two λ-abstractions is only permitted if they have the same number and type of arguments. The actual structure of the λ-abstraction (its number of arguments and their type) does not change, but its content, i.e. the function that it expresses, is evolved along with the rest of the program according to the specified fitness function.

This setting was applied to the Even-N-Parity problem and a general solution (i.e., for any N) was learnt (Yu and Clack 1998*b*). The system was able to *invent* the boolean function *xor*, *exor*, and *xand* in the various evolved solutions. At the time of this paper GP augmented with ADFs had only produced a specific solution for the Even-N-Parity problem for N = 11.

5.5 λ-Abstractions and Strongly Typed Evolutionary Programming

STEPS, like PolyGP, has the ability to invent new definitions for functions that are not contained in the original alphabet for the learner, by making use of λ-abstractions. In a similar manner to PolyGP, STEPS also provides the λ-abstractions as arguments to higher-order functions to achieve this.

The particular higher-order functions used by STEPS are *filter* and *setfilter*. The *filter* function takes as its arguments a boolean function and a list. It then return the list of elements for which the boolean function returns true. The boolean function corresponds to a property that is required of the elements of the resulting list:

$$filter(\lambda\ x\ (=\ \ (mod\ x\ 2)\ 0))\ [1,2,3,4,5,6] = [2,4,6]$$

For example in the above application of the *filter* function, all the even elements in the original list are filtered out and returned. The *setfilter* function is applied in a similar manner to obtain the set (rather than the list) containing only those elements that have a specified property.

These higher-order functions correspond to the list and set data types. Their use is therefore inappropriate unless their corresponding data types are used to construct the training examples. In the situation where the data types are used in the construction of the examples, then a subtree encapsulating the appropriate filter function will be generated by the *AdaptedEnumerate* algorithm (see Chapter 3). This subtree will have an empty branch waiting for a λ-abstraction, defining the property with with to constrain the list or set, to be generated during the population creation. The number of arguments of the newly generated λ-abstraction is dictated by the nature of the filter function[2]. The type of the arguments, however, are determined by the type of the data structure (i.e. the type of the list or the set), as this will determine the types of the filter function that is to be applied to it.

The λ-abstractions used by STEPS, like those used by PolyGP, and the subroutines of demand-driven automatically defined functions, are evolved dynamically if they are found to be useful in the solution of a problem. Their use is not explicitly requested by the user, but implicitly requested by the structure of the examples provided. During evolution, if the filter function is found to be helpful, then the λ-abstraction evolves to become a useful

[2] by filter function, it is meant either the *filter* function applied to lists or the *setfilter* function applied to sets

property with which to constrain a list or a set. If the filter function and its corresponding λ-abstraction is not found to be useful, then the combination will not appear in the fitter trees in the evolving population.

An example of the use of the filter function and λ-abstraction combination by STEPS can be found in chapter 6. The evolved function definitions encapsulated by the λ-abstraction were able to select useful properties of a chemical molecule. These properties were useful in the solution of a complex real-world problem.

It should be noted that STEPS has the potential to make use of any higher-order function and is not just restricted to the higher-order functions *map, fold* and *filter*.

5.6 Predicate Invention and Automatically Defined Functions - A Comparison

The Reformulation (i.e. the invention of *useful* predicates) and Demand-driven (i.e. the invention of *necessary predicates* types of predicate invention have their associated issues. For reformulation predicate invention the main issue is *what* to extract from the hypothesis to form the new predicate. For demand-driven predicate invention, the main, very difficult to decide, issue is *when* to apply predicate invention.

Automatically defined functions are used in conjunction with genetic programming in order to reformulate the main evolving program by allowing useful subroutines to be coevolved and used in the main program. These subroutines compress the amount of code required in the main program and can often make learning more efficient, and allow the solution to previously unsolvable problems to be evolved. However the problem of *when* they should be incorporated into the GP system is a difficult problem to decide. If the problem is of a hierarchical nature then the technique will often prove to be very fruitful, however, more often or not their use is discovered by trial and error.

Adaptive Representation Learning is an alternative to Automatically defined function that abstracts *useful* subtrees from the evolving program trees so that they may be reused as subroutines. The decision as to *when* to apply the technique is made by examining the entropy of the population.

The so called demand-driven automatically defined functions approach to learning is an extension of the original automatically defined functions. The problem of *when* they should be applied is overcome by allowing the number and their structure to be determined through the evolutionary process according their usefulness as evaluated by the fitness function.

The use of a combination of λ-abstractions and higher-order functions allows *useful* functions (or properties in the case of STEPS) to be *invented* during the evolutionary process. As with demand-driven automatically defined functions, the decision as to *when* they should be used in order to solve a problem is made through their survival (i.e., their usefulness as evaluated by the fitness function) during the learning process.

The more recent approaches to subroutine or function invention (demand-driven automatically defined functions and λ-abstractions) provide the most flexible methods, leaving the answer to the questions of *what* should be invented, and *when* it should be invented to be themselves *learnt*.

5.7 Summary

In this chapter the setting, motivations and various techniques for Predicate Invention in Inductive Logic Programming have been presented. Predicate Invention can be used in two situations providing two different settings, Reformulation, where an existing hypothesis is reformulated to achieve compression, and Demand-driven, where the invention of a new predicate is necessary in order for the target concept to be learnt. The use of automatically defined functions in Genetic Programming is motivated by subroutine reuse, i.e. the compression of the main program by allowing the coevolution of subroutines that may be repeatedly used in the main program. There are several variations on the automatically defined functions theme, some of which have been discussed in this chapter. In particular the use of demand-driven automatically defined functions, and λ-abstractions as highly flexible methods for inventing new useful functions has been discussed. STEPS makes use of λ-abstractions in order to invent useful new properties that allow good solutions to difficult real world problems to be obtained (e.g., see chapter 6). Finally a comparison of the various techniques for inventing new necessary or useful predicates or functions and the setting in which they are used was made. This comparison established a previously unmade (to the authors knowledge) connection between the issues of predicate invention, and automatically defined functions.

Chapter 6

Real World Concept Learning Problems

6.1 Introduction

The simple problems introduced in section 4.4 are often called *Toy* problems due to their synthetic nature. These problems are deliberately designed to be simple, small (both in number of features and examples), and with perfect data (i.e. no noise) so that the actual answer to the problem is easily identifiable. The purpose of these simple toy problems is to make diagnostics of the features of a learning system easier and to act as a proof of concept. In chapter 4 some of these toy problems were used to identify in which cases particular learning strategies would be useful.

However the ultimate aim of learning systems is to solve non-synthetic *Real World* problems. These problems are based on real measurements for a real problem which involves learning a description for a particular entity. Often these measurements are taken over many cases with many features making the problem too complex to be solved by a human expert, so a result that is produced by a learning system is of practical importance. Because of the nature of the data collection, the examples to be provided to the learning system often contain noise, so that an exact solution for the dataset does not exist. The large noisy datasets provided for these hard problems have certain implications for the learning process. A large number of features leads to an explosion in the search space which compounded with the large number of examples leads to a huge increase in the time necessary to find a solution. In addition it is difficult to identify when an optimal solution has been found and so terminate the learning as the data is often imperfect and messy. Therefore increasingly complex techniques are required.

There are many real world datasets readily available on the Internet resulting in many such problems becoming benchmark problems in machine learning. The UCI machine learning repository (UCI 1999) contains a substantial number of these datasets, however in the main these are attribute value problems. The Oxford University Comlab database (Oxf 1999) on the other hand contains a number of more structured and relational datasets to which many ILP systems have been applied. It is the success of ILP systems on a wide variety of real world concept learning problems that has contributed to the interest and immense growth in this field. The areas to which ILP has been applied include molecular biology (Muggleton *et al.* 1992, King *et al.* 1992, King and Srinivasan 1995, King *et al.* 1996, Srinivasan *et al.* 1994, Srinivasan *et al.* 1995*a*, Srinivasan *et al.* 1995*b*), finite element mesh design (Dolšak *et al.* 1994, Dolšak and Muggleton 1991), diagnosis and control (Feng 1992, Cameron-Jones and Quinlan 1993, Klingspor 1994, Klingspor *et al.* 1996, Lavrač and Džeroski 1994, Michie and Camacho 1994), chess (Džeroski 1993, Lavrač and Džeroski 1994, Bain and Srinivasan 1995, Bain and Muggleton 1994, Muggleton *et al.* 1989), and natural language processing (Zelle and Mooney 1993, Zelle and Mooney 1994, Zelle and Mooney 1996, Ling 1994, Mooney 1996, Mooney and Califf 1995, Malerba 1993, Semeraro *et al.* 1994, Esposito *et al.* 1993*a*, Esposito *et al.* 1993*b*, Esposito *et al.* 1994).

In this chapter STEPS is applied to two real world concept learning problems from the molecular biology domain. The inherent structure of the graphical representation of molecules is naturally captured by the individuals-as-terms representation used in STEPS. In section 6.2 STEPS is applied to the well known benchmark problem of predicting mutagenic compounds. Section 6.3 reports on the application of STEPS to the *IJCAI* Predictive Toxicology Evaluation (PTE2) challenge.

Some of the work presented in this chapter has appeared as (Kennedy *et al.* 1999)

6.2 Predicting Mutagenic Compounds

6.2.1 Problem Description

The first real world problem to be examined is a Structure Activity Relationship (SAR) problem involving the prediction of mutagenic activity (Srinivasan *et al.* 1996, Srinivasan *et al.* 1994). SAR problems involve obtaining descriptions that accurately characterise levels of biological activity of chemical compounds in terms of their molecular structure. The particular class of chemical compounds involved in this study are nitro-aromatic compounds that occur in car exhaust fumes and are produced during the synthesis of thousands of industrial compounds.

Chemical compounds found to have a high level of mutagenic activity are often found to be carcinogenic and so are likely to damage DNA. Prediction of mutagenic activity is necessary so that less hazardous compounds can be produced. The levels of mutagenic activity are obtained using an Ames test. However standard testing procedures are not always possible due to the toxicity to the test organisms, therefore a method for the automatic prediction of mutagenic activity is of practical importance.

In (Debnath *et al.* 1991) the Ames test results of 230 nitro-aromatic compounds are provided. The paper also identifies that these 230 chemical compounds can be divided into 188 compounds that are amenable to the technique of linear regression, i.e. their results are readily predicted by linear regression (this set is called regression friendly), and the remaining 42 compounds that linear regression is unable to predict (this set is referred to as regression unfriendly).

6.2.2 Representation

In order to tackle the Mutagenicity problem using STEPS the original Prolog representation (Oxf 1999), consisting of the 230 training cases, was translated into the Escher closed term representation.

The Escher representation for each chemical molecule is a highly structured term consisting of some properties of the molecule identified as being useful by domain experts (Debnath *et al.* 1991) and the atoms and bonds that are connected to give the structure of the molecule:

```
type molecule = (Ind1, IndA, Lumo, {Atom}, {Bond});.
```

The non-structural and pre-compiled structural properties of the molecule consist of `Ind1`, a Boolean indicator that identifies compounds containing three or more fused rings:

```
data Ind1 = Bool;,
```

and `Inda`, a Boolean indicator that identifies acenthrylenes. There are five of these in the dataset. These compounds have been identified as being less active than expected, the reason for which is unknown (Debnath *et al.* 1991):

```
type IndA = Bool;,
```

and `LUMO`, the energy of the compound's lowest unoccupied molecular orbital obtained from a quantum mechanical molecular model:

```
type Lumo = Float;.
```

The atom and bond structure that make up the molecule is represented as a graph, i.e., a set of atoms and a set of bonds connecting pairs of atoms. An atom consists of a label (which is used to reference that atom in a bond description), an element, one of 233 types represented by integers, and a partial charge which is a real value:

```
type Atom = (Label,Element,AtomType,PartialCharge);
data Element = Br | C | Cl | F | H | I |
               N  | O | S;
```

A bond is a tuple consisting of a pair of labels for the atoms that are connected by the bond and the type of the bond, which is indicated by a number:

```
type Bond = ((Label,Label),BondType);
```

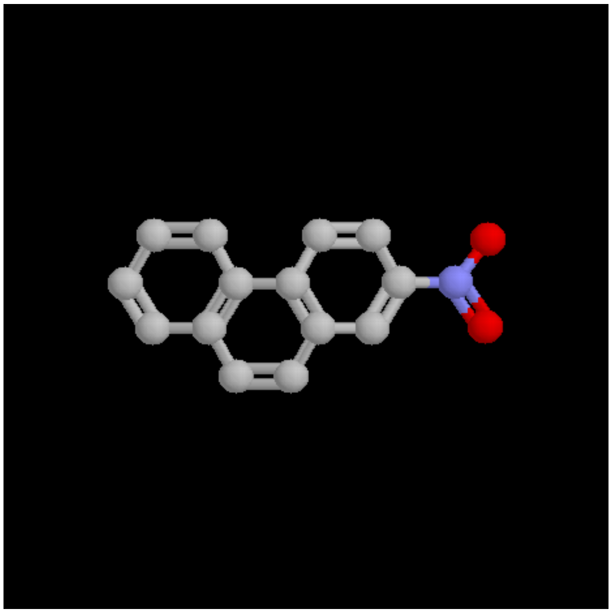


Figure 6.1: The chemical compound *Phenanthrene, 2-nitro-* in ball and stick format

It should be noted that the original Prolog representation included an additional attribute `logP`, the log of a compound's octanol/water partition coefficient (i.e. hydrophobicity). Initial experiments with the regression friendly dataset found this attribute to be superfluous and it was therefore not included in the experiments presented here.

Figure 6.2 gives the Escher closed term representation of the sample molecule from the Mutagenicity dataset in Figure 6.1. Figure 6.2 clearly illustrates the use of the integer atom labels that are used to reference each atom in the bond descriptions. Recall that each bond consists of a pair of atom labels and a number indicating the type of bond between the two atoms specified. This representation of the bonds is not completely flawless as the pair implies that there is an ordering of the atoms between which a bond occurs. This predicament could be overcome by specifying another bond description for each bond containing the swapped pair of atom references, causing some redundancy. This is a general problem for representing undirected graphs in this manner, however as mentioned in section 3.4.1, the enumerate algorithm has recently been enhanced to include a graph datatype which alleviates this problem.

```
(True,
 False,
 (-1.246),
 -- Atom set
 {(1,C,22,(-0.117)),(10,C,3,0.142),(11,C,27,(-0.087)),
  (12,C,27,0.013),(13,C,22,(-0.117)),(14,C,22,(-0.117)),
  (15,C,3,0.143),(16,C,3,0.143),(17,C,22,(-0.117)),
  (18,C,22,(-0.117)),(19,C,22,(-0.117)),(2,C,22,(-0.117)),
  (20,C,22,(-0.117)),(21,C,3,0.142),(22,C,3,0.143),
  (23,C,3,0.142),(24,N,38,0.812),(25,O,40,(-0.388)),
  (26,O,40,(-0.388)),(3,C,22,(-0.117)),(4,C,195,(-0.087)),
  (5,C,195,0.013),(6,C,22,(-0.117)),(7,C,3,0.142),
  (8,C,3,0.143),(9,H,3,0.142)},
 -- Bond set
 {((1,2),7),((1,7),1),((11,12),7),((11,17),7),
  ((12,13),7),((13,14),7),((13,15),1),((14,16),1),
  ((14,5),7),((17,18),7),((17,21),1),((18,19),7),
  ((18,22),1),((19,20),7),((2,3),7),((2,8),1),
  ((20,12),7),((20,23),1),((24,19),1),((24,25),2),
  ((24,26),2),((3,4),7),((3,9),1),((4,11),7),
  ((4,5),7),((5,6),7),((6,1),7),((6,10),1)});
```

Figure 6.2: Escher representation of a sample molecule from the Mutagenicity dataset

6.2.3 Experiments

6.2.3.1 Method

The data provided includes both structural and non-structural information. The non-structural information consists of the precompiled structural attributes (i.e., `Ind1` and `Inda`) and a number representing a molecules' `LUMO` value. The structural information is simply the graphical representation of the molecules in terms of atoms and bond connectives. The inherent structure of the graphical representation of molecules is naturally captured by the closed-term representation used in STEPS.

The aim of the Mutagenicity problem is to generate a concept description that can distinguish between chemical compounds with log mutagenicity > 0, or *Active* compounds, and chemical compounds with log mutagenicity < 0 or *InActive* compounds. As previously mentioned, the concepts induced here are restricted to a simple template. So with two classes, the template for the concept description is given by:

```
IF Cond THEN C1 ELSE C2;
```

Since either *Active* or *Inactive* can be used for `C1` (leading to potentially different induced theories) and the non-structural information can be included in the information provided to the learner or not (see experiments with PTE2 data), there are four potential settings to compare.

As there is no explicitly defined test set associated with the Mutagenicity data, the results reported in the literature for this problem are often expressed as the result of a k-fold cross validation. This technique is often used in machine learning for measuring an algorithm's performance on a particular data set. It involves randomly splitting the data up into k parts and then training the algorithm on $k-1$ parts of the data and using the remaining holdout set as a test set. This process is repeated until all k parts have been used as the holdout test set. The results are expressed as an average of the results over the k experiments.

As evolutionary algorithms are by nature stochastic, it is not immediately clear how to apply the k-fold cross validation technique as each run may produce different results. This may hinder an evolutionary algorithms as their advantage lies in the adjustment of the parameters over a number of runs to obtain an optimal solution. For the purposes of this experiment, a single run will be carried out for each holdout set. For each of these runs there will be an identical initial population (i.e. the same seed for the random number generator will be used) so that as little as possible is altered. However - this population may not be the population that will give the best result for the particular dataset being used.

As mentioned in Section 6.2.1, the Mutagenicity dataset can be considered to have two parts: the regression friendly dataset and the regression unfriendly dataset. These two sub-datasets are considered as two separate problems in the literature. This is the case here as well. Therefore a 10-fold cross validation is carried out on the 188 chemical compounds in the regression friendly dataset for each of the four setting configurations. Then a 10-fold cross validation is carried out for the 42 regression unfriendly compounds for each of the four setting configurations.

The best individual in the population is selected to be the individual with the highest accuracy with respect to the training dataset. If more than one such individual exists then the most recently evolved individual is selected. This is an arbitrary decision. This individual is then tested on the holdout test set to obtain an accuracy to contribute to the average over all the k holdout sets.

The parameters for STEPS, for all experiments are as follows:

Parameter	**Setting**
population size	300
maximum no. generations	60
maximum no. fitness evaluations	18000
minimum depth	3
maximum depth	20
selection	tournament

Results are provided for both the Depth learning Strategy and the Hybrid learning strategy. The cover strategy was not considered to be useful in this situation as the depth of an optimal solution is not known.

6.2.3.2 Results - Regression Friendly Dataset

Table 6.1 gives the results for a 10-fold cross validation for each of the four configurations using the Depth learning strategy on the regression friendly mutagenicity data set, while Table 6.2 provides the results using the Hybrid learning Strategy. The average accuracy is the average of the accuracies of the best performing theory for each run for a particular configuration. The average size is the average of the sizes (i.e. number of nodes in the program tree) of the best performing theory for each run for a particular configuration. Each run took on average 10 hours to complete.

Little difference can be observered in the results obtained for both of the learning strategies examined. For both the Depth and Hybrid learning strategies, the best results for this data set were obtained using all of the available data. This is because good results can be

Table 6.1: Results obtained by STEPS for the four configurations for the regression friendly dataset using the Depth learning Strategy

Configuration (C1-Info)	Av. Accuracy	Std. Deviation	Av. Solution Size
Active-All	87.3%	6.1%	12.6
Inactive-All	86.8%	6.6%	23.9
Active-Struc	76.6%	11.1%	91.6
Inactive-Struc	79.3%	11.7%	81.0

Table 6.2: Results obtained by STEPS for the four configurations for the regression friendly dataset using the Hybrid learning Strategy

Configuration (C1-Info)	Av. Accuracy	Std. Deviation	Av. Solution Size
Active-All	87.3%	6.1%	13.7
Inactive-All	86.8%	6.6%	18.4
Active-Struc	78.1%	11.4%	80.9
Inactive-Struc	81.4%	11.1%	89.7

obtained though the following program statement

```
proj1(v1) == True || proj3(v1) < -2.368
```

for descriptions of *Active* molecules and the reverse for descriptions of *Inactive* molecules. This statement identifies molecules with the value True for their `Ind1` boolean indicator, or molecules with a `LUMO` value less than -2.368, i.e. it identifies molecules with three or more fused rings or molecules whose energy of the lowest unoccupied molecular orbital is less than -2.368. This statement alone covers 89% of the examples in the dataset and is therefore a good indicator of mutagenic molecules. Once an individual description contains this statement, its fitness soars and it therefore dominates the population. This statement is often found in individuals in the initial population. These descriptions only improve their fitness by adding additional statements that will cover one or two individual examples. Therefore once this statement has been discovered the problem is practically solved and so is no longer an interesting problem.

However this problem was also attempted by examining a molecule's atom and bond

Table 6.3: Results obtained by STEPS for the four configurations for the regression unfriendly dataset using the Depth learning Strategy

Configuration (C1-Info)	Av. Accuracy	Std. Deviation	Av. Solution Size
Active-All	71%	24.4%	61.0
Inactive-All	76%	26.6%	41.6
Active-Struc	68.5%	22.3%	72.0
Inactive-Struc	73%	25.2%	53.3

Table 6.4: Results obtained by STEPS for the four configurations for the regression unfriendly dataset using the Hybrid learning Strategy

Configuration (C1-Info)	Av. Accuracy	Std. Deviation	Av. Solution Size
Active-All	66%	22.8%	77.8
Inactive-All	73%	25.2%	43.7
Active-Struc	68%	24.2%	78.4
Inactive-Struc	73%	25.2%	53.3

information only (i.e. the structural information related to a molecule). Although the accuracies resulting form the 10-fold cross validations are not as effective as those obtained using all the information, this problem is more interesting as the solution is expressed in terms of characteristics extracted from the structure making up the molecule.

The results obtained on both datasets (i.e., with and without the non-structural information) are comparable to the results of other learning systems on the same datasets (e.g. see (Srinivasan *et al.* 1996))

6.2.3.3 Results - Regression Unfriendly Dataset

Table 6.3 gives the results for a 10-fold cross validation for each of the four configurations using the Depth learning strategy on the regression Unfriendly mutagenicity data set, while Table 6.4 provides the results using the Hybrid learning Strategy. Each run took on average 12 hours to complete.

For the regression unfriendly dataset, the most accurate result obtained from a ten fold cross validation was for the Depth strategy inducing a description for an *Inactive* molecule from all the available data. Both strategies obtain almost identical results when learning from the structural information only. However the Depth strategy appears to outperform the Hybrid strategy in terms of accuracy when learning from all of the available data. A reason for this could be that the Hybrid strategy has a tendency to overfit the training data set. Here it obtains a lower average error on the training data, with a higher error rate on the test data, compared to those obtained for the Depth learning strategy.

The results obtained by STEPS for the regression unfriendly dataset are worse than those obtained by Progol in (Muggleton *et al.* 1998). Progol was able to obtain an accuracy of 83% both including and not including the indicator variables `Inda` and `Ind1`. These results were obtained using a leave-one-out cross validation strategy and in addition to the original Prolog representation of the atom and bond information and the `LUMO` attribute, the learner had access to the `logP` information (which in retrospect should have been included in the data provided to STEPS) and various elementary chemical concepts (e.g. definitions of methyl groups, nitro groups, aromatic rings etc.).

6.3 Predicting Carcinogenic Compounds

Whilst the application of STEPS to the Mutagenesis problem is an interesting exercise, the motivation for this study was to *pave the way* for the application of STEPS to the seemingly more challenging problem of predicting the carcinogenic activity of chemical compounds.

6.3.1 Problem Description

The National Institute of Environmental Health Sciences (NIEHS) in the USA provides access to a large database on the carcinogenicity (or non-carcinogenicity) of chemical compounds through the National Toxicology Program (NTP). The information has been obtained by carrying out long term bioassays that classify over 300 substances to date. The Predictive Toxicology Evaluation (PTE) challenge was organised by the NTP to gain insight into the features that govern chemical carcinogenicity (Bristol *et al.* 1996).

The first *International Joint Conference on Artificial Intelligence (IJCAI)* PTE challenge involved the prediction of 39 chemical compounds that were, at the time, undergoing testing by the NTP. The training set consisted of the remaining compounds in the NTP database. The participants consisted of both experts in the area of chemical toxicology and machine learning systems. Symbolic machine learning, and in particular Inductive Logic Programming, has been applied with great success to bio-molecular problems in the past ((Srinivasan *et al.* 1994), (King *et al.* 1996)). Symbolic machine learning techniques are particularly suitable for problems of this type since it is not only the prediction that is interesting, but also the induced theory which provides an explanation for the predictions. The learning system Progol, for example, was entered into the PTE challenge and obtained results that were competitive with those obtained by the expert chemists (Srinivasan and King 1997).

Following on the success of the first challenge, a second round of the PTE challenge (PTE2) (NIE 1999) was presented to the AI community at *IJCAI* in 1997 (Srinivasan *et al.* 1997). The PTE2 challenge involves the prediction of 30 new bioassays for carcinogenesis being conducted by the NTP. The training set consists of the remaining 337 bioassays in the NTP database. At the time of writing the results for 7 of the chemical compounds in the test set are still unknown. Ten machine learning entries have been made so far in reaction to the *IJCAI'97* challenge, and their performance has been calculated on the 23 chemical compounds whose results are known (OUC 1997). In addition to predictive accuracy, entries have been evaluated according to whether or not they exhibit explanatory power, where the explanatory power of a theory exists "... if some or all of it can be represented diagrammatically as chemical structures." (Srinivasan *et al.* 1997).

The PTE challenges provide the machine learning/data mining communities with an independent forum in which intelligent data analysis programs and expert chemists can work together on a difficult scientific knowledge discovery problem. In this section, we report on the application of STEPS to PTE2.

6.3.2 Representation

In order to tackle the PTE2 problem using STEPS the original Prolog representation (OUC 1997), consisting of the 337 training cases, was translated into the Escher closed term representation.

The Escher representation for each chemical molecule for the PTE2 dataset is similar to the representation of each chemical molecule for the mutagenicity dataset, consisting of a highly structured term consisting of some properties of the molecule and the atoms and bonds that connect up to give the structure of the molecule:

```
type molecule =
    (Ames,{Genotox},{Genotox},{Ashby},{Atom},{Bond});.
```

The properties of the molecule resulting from laboratory analyses consist of Ames test results (i.e., whether the compound is mutagenic or not - mutagenicity is an indication of carcinogenicity):

```
type Ames = Bool;,
```

two sets of genetic toxicology test results, one for positive and one for negative results:

```
data Genotox = Chromaberr | Chromex | CytogenCa | CytogenSce
        | DrosophilaRT | DrosophilaSlrl | MicronucF | MicronucM
        | MouseLymph | Salmonella | SalmonellaN | SalmonellaReduc;,
```

and a set of Ashby alerts and their counts (properties of the molecule that are likely to indicate carcinogenicity, discovered by a toxicology expert):

```
type Ashby = (Indicator, Count);
type Count = Int;
data Indicator = Amino | Di10 | Di227 | Di23 | Di232 | Di48
        | Di64 | Di66 | Di67a | Di8 | Ethoxy | Halide10
        | Methanol | Methoxy | Nitro | Ringsize4 | Cyanate
        | Di260 | Di281 | Di51;.
```

The atom and bond structure that make up the molecule is represented as a graph, i.e., a set of atoms and a set of bonds connecting pairs of atoms. An atom consists of a label (which is used to reference that atom in a bond description), an element, one of 233 types represented by integers, and a partial charge:

```
type Atom = (Label,Element,AtomType,PartialCharge);
data Element = As | Ba | Br | C | Ca | Cl | Cu | F | H | Hg
             | I | K | Mn | N | Na | Pb | O | P | S | Se
             | Sn | Te | Ti | Zn;
type AtomType = Int;
type PartialCharge = Float;.
```

A bond is a tuple consisting of a pair of labels for the atoms that are connected by the bond and the type of the bond:

```
type Bond = ((Label,Label),BondType);
type BondType = Int;.
```

Figure 6.4 gives the Escher closed term representation of the sample compound from the PTE2 dataset in Figure 6.3.

6.3.3 Experiments

6.3.3.1 Method

The data provided includes both structural and non-structural information as in the mutagenesis problem. The non-structural information consists of the outcomes of a number of laboratory analyses (e.g., Ashby alerts, Ames test results). The structural information is simply the graphical representation of the molecules in terms of atoms and bond connectives. Most contestants so far have relied heavily on results of short term toxicity (STT) assays. It appears that, for some learning tasks and systems, the addition of this type of information improves the predictive performance of the induced theories (Srinivasan *et al.* 1996). On the other hand, for other tasks and systems, the opposite seems to be

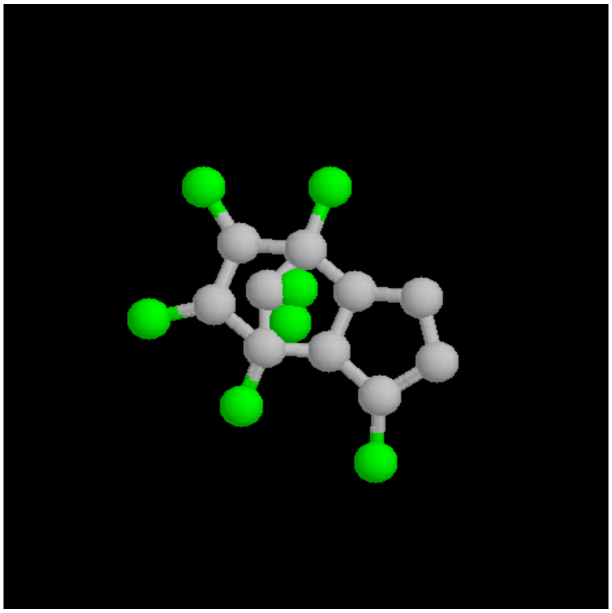


Figure 6.3: The chemical compound *Heptachlor* in ball and stick format

true, i.e., propositional information has a negative effect on generalisation (Flach and Lachiche 1999).

In this thesis, it is strongly argued that, for the PTE2 task, if toxicological information only makes explicit, properties implicit in the molecular structure of chemicals, the non-structural information is actually superfluous. In addition, obtaining such properties often requires time cost and in some cases substantial utilisation of laboratory animals (Bahler and Bristol 1993, Lee *et al.* 1996). Provided a sufficiently expressive representation language, good solutions may be obtained from the structural information only. Hence, prediction can potentially be made faster and more economically. The experiments reported here with STEPS support this claim. The inherent structure of the graphical representation of molecules is naturally captured by the closed-term representation used in STEPS. The rules obtained with structural information only are better in terms of accuracy to those obtained using both structural and non-structural information. In addition, our approach is more likely to provide insight into the mechanistic paths and features that govern chemical toxicity, since the solutions produced are readily interpretable as chemical structures. STEPS ranks joint 2nd of 10 in the current league table of the PTE2 chal-

```
( False,                              --The Ames test result
 {CytogenSce, MouseLymph},            --Positive set of Genotoxity tests
 {CytogenCa, Salmonella}              --Negative set of Genotoxity tests
 {(Halide10, 5)},                     --Ashby alerts
   --Atom set
 {(1,C,10,0.177), (10,H,3,0.127), (11,H,3,0.046), (12,H,3,0.177),
  (13,H,3,0.177), (14,C,10,0.477), (15,Cl,93,(-0.124)),
  (16,Cl,93,(-0.124)), (17,Cl,93,(-0.124)), (18,H,3,0.127),
  (19,Cl,93(-0.124)), (2,C,10,(-0.024)), (20,Cl,93(-0.124)),
  (21,Cl,93,(-0.124)), (22,Cl,93,(-0.123)), (3,C,16,(-0.124)),
  (4,C,16,(-0.124)), (5,C,10,(-0.024)), (6,C,10,(-0.074)),
  (7,C,10,0.177), (8,C,16,(-0.124)), (9,C,16,(-0.124)}
   --Bond set
 {((1,10),1), ((1,2),1), ((14,15),1), ((14,16),1), ((14,2),1),
  ((2,22),1), ((2,3),1), ((3,21),1), ((3,4),2), ((4,20),1),
  ((4,5),1), ((5,14),1),((5,19),1), ((5,6),1), ((6,1),1),
  ((6,11),1), ((6,7),1), ((7,17),1), ((7,18),1), ((7,8),1),
  ((8,12),1), ((8,9),2), ((9,1),1), ((9,13),1)});
```

Figure 6.4: Escher representation of a sample molecule from the PTE2 dataset

lenge when this criterion is considered, and joint 1st when interpretability is sacrificed for accuracy.

The aim of PTE2 is to generate a concept description that can distinguish between *Active*, or carcinogenic compounds and *InActive*, or non-carcinogenic compounds. As previously mentioned, the concepts induced here are restricted a simple template, therefore, as with the mutagenesis problem, there are four settings to compare.

As STEPS is a stochastic algorithm the experiments are repeated ten times for each particular setting. The best performing theories as measured on the training data are output at the end of a run. The theory with the highest accuracy on the test set is then chosen as the best theory for that particular run. The best theory from the set of ten experiments is selected as the theory for a particular setting of data and description format.

Genetic Boosting

The method of fitness evaluation used here is the Stepwise Adaption of Weights (SAW) method (van Hemert and Eiben 1997). The SAW fitness function essentially implements a weighted predictive accuracy measure, which is based on the perceived difficulty of the examples to be classified. During evolution, only training examples are used. The SAW fitness function rewards an individual for the correct classification of a *difficult* example by associating a weight with each example. An example is considered difficult if the current best theory of the generation can not classify it correctly, in which case its associated weight is incremented by an amount `delta weight`. The weights are adjusted every `weight gen` generations. The fitness for a particular individual therefore becomes a weighted sum of the number of training examples that it can correctly classify. The SAW technique can be viewed as *Genetic Boosting.* Boosting is a technique that is typically used with decision trees in order to produce multiple decision trees that can be combined to boost the accuracy of the classification for the problem (Freund and Schapire 1996). A weight is maintained for each training example in order to reflect the importance of its correct classification. By altering the weights, a number of different decision trees may be generated and a weighted combination of these trees is combined in order to obtain the classification.

The parameters for STEPS and SAW, for all experiments are as follows.

Parameter	**Setting**
delta weight	0.1
weight gen	5
population size	100
maximum no. generations	150
maximum no. fitness evaluations	15000
minimum depth	3
maximum depth	20
selection	tournament

6.3.3.2 Results

Table 6.5 gives the results for each of the four configurations using the Depth learning strategy, whilst Table 6.6 provides the equivalent information for the Hybrid learning strategy. The Best Accuracy is the accuracy of the best performing theory out of all ten

Table 6.5: Results obtained by STEPS for the four configurations using the Depth learning Strategy

Configuration (C1-Info)	**Best Accuracy**	**Av. Accuracy**
Active-All	65%	52.2%
Inactive-All	74%	50.4%
Active-Struc	70%	54.7%
Inactive-Struc	78%	68.2%

Table 6.6: Results obtained by STEPS for the four configurations using the Hybrid learning Strategy

Configuration (C1-Info)	**Best Accuracy**	**Av. Accuracy**
Active-All	70%	51.4%
Inactive-All	65%	53.6%
Active-Struc	65%	58.4%
Inactive-Struc	78%	60.6%

runs for a particular configuration. The average Accuracy is the average of the accuracies of the best performing theory for each run for a particular configuration. Each run took between 7 hours and 28 hours to complete.

The program selected as the best out of the forty runs with the various dataset and default class configurations for the Depth learning strategy achieves a predictive accuracy of 78% on the test data and was obtained with the *Inactive-Struc* Configuration. The program selected as the best using the Hybrid learning strategy also achieves a predictive accuracy of 78% on the test data and was also obtained with the *Inactive-Struc* Configuration. The theory obtained using the Depth learning strategy, was the first of the two different theories obtained with identical accuracy on the test data, to be obtained. This theory is currently undergoing more thorough analysis (Bristol 1999) and is joint second in the PTE2 league table (see (OUC 1997) and a summary of the results in Table 6.7). It is given here in Escher program format in figure 6.5 and in English in figure 6.6. Note that the notation `\x` in figure 6.5 is the Escher notation for λx (see section 5.5), and also note the use of the higher order function `setfilter` to extract useful properties of a chemical molecule.

```
carcinogenic(v1) =
       if
        ((((card (setfilter (\v3 -> ((proj2 v3) == O))
            (proj5 v1))) < 5) &&
          ((card (setfilter (\v5 -> ((proj2 v5) == 7))
            (proj6 v1))) > 19)) ||

        exists \v4 -> ((elem v4 (proj6 v1)) && ((proj2
           v4) == 3))) ||

        (exists \v2 -> ((elem v2 (proj5 v1)) &&
           ((((((proj3 v2) == 42) ||
           ((proj3 v2) == 8)) ||
           ((proj2 v2) == I)) ||
           ((proj2 v2) == F)) ||
           ((((proj4 v2) within (-0.812,-0.248)) &&
             ((proj4 v2) > -0.316)) ||
           (((proj3 v2) == 51) ||
           (((proj3 v2) == 93) &&
              ((proj4 v2) < -0.316))))))
        && ((card (setfilter (\v5 ->
                 ((proj2 v5) == 7))(proj6 v1))) < 15))
     then Inactive
     else Active;
```

Figure 6.5: The best theory produced by STEPS as an Escher program

```
A molecule is Inactive if it
            contains less than 5 oxygen atoms
               and has more than 19 aromatic bonds,
            or if it contains a triple bond
            or if it contains an atom that
                is of type 42 or 8 or 51
                or is an iodine or a fluorine atom
                or has a partial charge between -0.812 and -0.316
                or is of type 93 with a partial charge less than -0.316
             and contains less than 15 aromatic bonds
          Otherwise the molecule is active.
```

Figure 6.6: The best theory produced by STEPS in English

Table 6.7: Summary of the results for PTE2

Entry	Method	Accuracy	Explanatory
LRI-Distill	Stochastic voting technique	0.87	No
UB-STEPS	Strongly typed evolutionary technique	0.78	Yes
LRI-GloBo	Stochastic rule induction	0.78	Yes
OUCL-2	Decision tree and ILP	0.78	Yes
OFAI	Decision tree and Naive Bayes	0.74	Yes
Default		0.74	No
K.U. Leuven	ILP	0.70	Yes
K.U. Leuven	ILP	0.65	Yes
OUCL-1	Decision tree and ILP	0.57	Yes
AI Lab, NTHU	GA	0.52	Yes
K.U. Leuven	ILP	0.48	Yes

6.3.3.3 Discussion of initial results

The rules obtained by STEPS for both the Depth and Hybrid learning strategies using structural information only, outperform those obtained by STEPS using the additional non-structural information and are comparable in terms of accuracy to those obtained using both structural and non-structural information by all PTE2 participants (see Table 6.7 for a summary of the results). In addition, this approach may produce insights into the underlying chemistry of carcinogenicity, one of the principal aims of the PTE2 challenge. While undergoing further analysis by a toxicologist, the first part of the submitted theory, i.e, the rule *"contains less than 5 oxygen atoms and has more than 19 aromatic bonds"*, was found to have some similarity with an existing drug discovery rule.

> *" ... brings to mind a rule used in pharmacology and drug discovery, called the 'Rule of 5' mnemonic for bioavailability. In general, low bioavailability indicates low biological activity towards any endpoint."* (Bristol 1999).

Furthermore, as the theory produced by STEPS relies only on structural information, carcinogenic activity for a new chemical can be predicted without the need to obtain the non-structural information from laboratory bioassays. Hence, results may be expected in a more economical and timely fashion, while reducing reliance on use of laboratory animals.

The application of a stochastic learning algorithm such as STEPS to a learning problem such as the PTE2 challenge raises some interesting issues. When applying STEPS to the problem, a number of runs for each setting were carried out, each producing at least one best of run hypothesis. The best hypothesis for a particular setting was then chosen according to the best of run hypotheses performance on the test data provided. While this process for selecting a best hypothesis is the generally accepted method in evolutionary communities, the broader Machine Learning community appears to be uncomfortable with this process. In this case the general view is that a hypothesis should not be selected, and in the case of the PTE2 challenge submitted, on the basis of the test set.

One solution would be to submit every hypothesis obtained. However, in the case of applying STEPS to the PTE2 challenge this would mean swamping the contest with at least 80 hypotheses (two strategies *times* four configurations × 10 runs each). This would not be a desirable solution. In the next section two alternative solutions are examined.

6.4 Further Experiments

In this section further experiments are carried out using the PTE2 dataset. The first experiment involves adding extra flexibility to the rules induced by STEPS for the PTE2 problem by removing the restriction on the form of the description dictated by the initial template. The other two experiments attempt to resolve the problem of choosing among hypotheses of equal quality when a single solution is required / allowed.

6.4.1 Adding extra flexibility to the induced rules

The experiments carried out so far have made use of a template for the concept description of the form

```
IF Cond THEN C1 ELSE C2;
```

This template dictates whether a description for the particular class *Active* or *Inactive* compound is generated depending on the choice made for the C1 and C2 values. This learning of a description for a particular class is typical of many machine learning systems.

Table 6.8: Parameters for STEPS and SAW, for the experiments without the initial template

Parameter	Setting
delta weight	0.1
weight gen	5
population size	100
maximum no. generations	150
maximum no. fitness evaluations	15000
minimum depth	3
maximum depth	30
selection	tournament

However, techniques such as decision tree algorithms are capable of learning a description for more than one class in a single run. STEPS can easily be adapted so that it can remove the restriction of the template and induce a description that combines the characteristics of more than one class; in this case, both an *Active* and an *Inactive* compound.

In order for STEPS to learn a description combining the characteristics of both an *Active* and an *Inactive* compound the initial template from which STEPS grows its initial program trees is abandoned and an `if_then_else` function is placed into the alphabet for learning. The `if_then_else` function takes three arguments. The first argument is of boolean return type, and the second and third arguments expect an argument that returns a class label. The inclusion of the `if_then_else` function in the learning alphabet allows the induction of decision lists of arbitrary form and complexity. This in turn allows the arbitrary combination of characteristics of both *Active* and *Inactive* compounds.

The parameters for STEPS and SAW, for the experiments without the initial template and with the `if_then_else` function are presented in Table 6.8.

Note that these parameters differ from the parameters specified for the original PTE2 experiments in section 6.3.3.1 in the maximum depth field. Here the maximum depth parameter is extended due to the greater complexity of the problem, i.e., we are now searching for a description of both an *Active* and an *Inactive* compound instead of a description for one or the other of them.

Table 6.9: Results obtained by STEPS for PTE with function `if_then_else` in alphabet

Configuration (C1-Info)	Best Accuracy	Av. Accuracy
All - Depth	60.9%	47.2%
All - Hybrid	70%	58.6%
Struc - Depth	74%	56.6%
Struc - Hybrid	74%	62.3%

The experiments were carried out in a similar manner to the original PTE2 experiments, i.e. ten runs carried out for each setting. However the number of settings for each learning strategy is reduced to two here as we are no longer specifying a value C1 for the initial template.

Results

The results for the experiments carried out are presented in Table 6.9. Each run took an average of 15 hours to complete. As indicated by the results of the original experiments on the PTE2 dataset, these results suggest again that the accuracy of the best induced rules can be increased by examining only the structural information of the molecules. However due to the extra flexibility of the induced rules, the nice concise description of an active or inactive compound is lost. Hence if this is the most natural formulation for the descriptions for this problem to take, the introduction of the extra flexibility may lead to a loss in accuracy.

Note that, further to the results of chapter 4, the information gained here on the performance of the Depth and Hybrid learning strategies confirms that there does not appear to be a significant difference between them.

6.4.2 Genetic Bagging

Bagging is a technique similar to *Boosting* (see section 6.3.3.1) in that it is used to combine the results obtained from multiple decision trees to obtain an optimal classifier for a problem (Quinlan 1996). The multiple decision trees are obtained by sampling (with replacement) the training data. This results in a training data set in which some of the original examples do not appear whilst others may appear more than once. A data set is compiled in this manner in order to train each of the required number of decision trees. A classification of a new example is then obtained by taking a vote across the individual resultant decision trees.

The voting across multiple classifiers technique can be adapted to stochastic algorithms such as STEPS in order to overcome the problem of deciding which of classifications resulting from the multiple hypotheses produced to choose.

This voting technique was applied to all 16 best on training set on all 10 runs hypotheses produced by STEPS on the PTE2 problem using the Depth Learning strategy with the *Inactive-Struc* configuration. For each of the molecules in the test set for the PTE2 problem a classification was obtained by taking the class for which the majority of the individual hypotheses voted for. In the case of a tie, the majority class, *Active* was adopted. By *Bagging* the classifications of all the hypotheses generated by this particular setting, an accuracy of 87% was obtained. This result puts STEPS in joint first position for the PTE2 challenge. Interestingly the current first place in the PTE2 challenge is held by Distill, which obtained its classifications for the test set by employing a stochastic voting technique (Sebag 1998).

However, although this technique seems to provide a more robust classification for the test set, it does have some drawbacks, namely the loss of explanatory power. The fact that the classification for each of the test molecules is obtained through a vote across a number of different hypotheses, it is no longer clear which of the characteristics of a molecule are contributing to its classification. This loss of explanatory power is not a problem if we are only interested in a more robust and accurate classification. However it is not only accurate classifications that are of interested in here. The induced rules are of great interest in deciphering the problem of identifying potential carcinogenic compounds.

6.4.3 Seeding

Seeding involves providing an initial population, that is to be evolved by an evolutionary algorithm, with expert or heuristic knowledge in order to aid the evolutionary process (Michalewicz 1992). For example, a mesh known to have the ability to be altered into a near optimal solution was seeded into the initial population in order to assist a Genetic Algorithm solve a Finite Element Mesh Analysis problem (Sharif and Barrett 1998).

Species Adaption Genetic Algorithms is another way in which the ideas of seeding have been used to improve the performance of Genetic Algorithms (Harvey 1991). Under the framework of SAGA a complex problem can be split into a sequence of subtasks of increasing difficulty. Each subtask is learnt in order, with the final population of the previous subtask being used as the initial population for the current subtask.

The basic principles of SAGA were adapted in order to carry out further experiments with the PTE2 dataset. The aim was to combine the good genetic material contained in all the best of run hypotheses obtained for a particular setting, in order to produce a superior *thoroughbred* hypothesis containing the best genetic material from each of these individuals.

The experiments were carried out as follows. The initial population for each of ten runs consisted of all sixteen of the best of run hypotheses from the *Inactive-Struc* setting obtained with the depth learning strategy. The parameters for STEPS, were as follows:

Parameter	**Setting**
elitism	0.1
population size	16
maximum no. generations	60
maximum no. fitness evaluations	960
minimum depth	3
maximum depth	20
selection	tournament
genetic operators	crossover

Here crossover was selected as the sole genetic operator, as the desired outcome was to recombine the already good genetic material contained in these best of run hypotheses. However some form of diversity is introduced by mutating an individual whose *identical twin* is already present in the population. In order for the fitness of the current best individuals of a run never to deteriorate, elitism was applied by directly copying the top ten percent of each population into the next generation.

However despite the promise of the ideas behind the experiments, the results obtained were disappointing. The aim of the experiments was to evolve a supreme understandable hypothesis that consisted of the best characteristics of the pre-evolved best of run hypotheses. Instead a large number of best of run hypotheses were found for each run none of which were able to outperform the best of *setting* hypothesis found for this setting (i.e., the hypothesis currently in joint second place in the PTE2 challenge).

6.5 Summary

This chapter has demonstrated the application of STEPS to two real world concept learning problems. The first problem to be attempted was the Mutagenicity problem. The Mutagenicity problem involves the prediction of Mutagenic chemical compounds and is frequently found as a benchmark problem in the literature on Inductive Logic Programming. In this study both the *regression friendly* and *regression unfriendly* data sets were considered in their entirety (i.e. all the structural and non-structural information describing a molecule) and by investigating the impact of learning from the structural descriptions only (i.e., the atom and bond descriptions). The results produced by STEPS are comparable to those found in the literature.

The second problem presented to STEPS consisted of the PTE2 problem. This problem has been posed as an International challenge to the Machine Learning community. The PTE2 challenge involves the prediction of the Carcinogenic activity of a test set involving 30 chemical compounds. The PTE2 problem is a challenging real world problem. The test set is very small and as the identification of carcinogenic compounds is an ongoing process, not all of the results for the test set are known. The data is quite skewed as the compounds that the toxicologists think likely to be carcinogenic are examined first. That said, STEPS was again applied to two forms of the data provided, i.e., with and without the non-structural attributes of the molecule obtained through laboratory tests. STEPS was able to find two different hypotheses, from only the atom and bond descriptions of the molecules, that obtained an accuracy of 78% on the test data. One of these hypotheses was submitted to the PTE2 challenge and was subsequently placed in joint second place. This hypothesis has the advantage over the competing hypotheses submitted that it is formed in terms of characteristics of the atom and bond connectives of the chemical compounds and therefore does not rely on the outcome of the expensive (in monetary and lab animal usage terms) laboratory bioassays and other chemical tests.

Further experiments were carried out using the PTE2 dataset. The first additional experiment carried out was to demonstrate additional flexibility of the form of the rules obtained by STEPS. The remaining two experiments attempted to overcome the problem in deciding which of the best-of-run hypotheses obtained by STEPS for the PTE2 problem should be selected without the results on the test data. The first suggestion, Bagging, obtained a more robust and accurate classification of the compounds in the test set. The result obtained by voting across all of the best of run hypotheses for a particular setting was an accuracy of 87% putting the results of STEPS joint first in the challenge. The final suggestion was to take a seeding approach to evolution by evolving a thoroughbred hypothesis from an initial population consisting of all of the best of run hypotheses from a particular setting. Whilst this is an interesting idea that should be investigated further for other problems, the technique did not produce convincing results here.

Chapter 7

Conclusions and Further Work

This chapter summarises the contributions of the thesis, discusses some of the limitations of STEPS and outlines areas of future research.

7.1 Contributions of Thesis

Escher is a functional logic language whose higher-order constructs allow arbitrarily complex observations to be captured and highly expressive generalisations to be conveyed. The original aim of this thesis was to alleviate the challenging problem of identifying an underlying structure that allows the organisation of the resulting hypothesis space, in such a way that it may be searched efficiently. This was achieved by STEPS, an evolutionary based system that allows the vast space of highly expressive Escher programs to be explored. STEPS extends the Strongly Typed Genetic Programming (STGP) paradigm and provides a natural upgrade of the evolution of concept descriptions to the higher-order level (see Figure 2.6).

The following summarises the main contributions of the thesis.

7.1.1 Evolution of Escher Concept Descriptions

STEPS allows highly expressive concept descriptions to be evolved from structured examples. STEPS achieves this through a combination of the following features:

Tree based representation: The Escher programs learnt by STEPS are represented as parse trees.

Problem domain: The problems attempted by STEPS are highly structured concept learning problems, whereas STGP is typically applied to Program Synthesis problems, where a functional program is produced by examining examples of desired output.

Observation language: STEPS uses the individual-as-terms approach to knowledge representation where all the information provided by an example is localised as a single closed term so that examples of arbitrary complexity can be treated in a uniform manner.

Hypothesis language: The concept descriptions evolved by STEPS are expressed in the highly expressive Escher programming language. Escher integrates features of both functional and logical programming languages and is a superset of the programming language Haskell.

Generation of the alphabet: The alphabet used by STEPS is not hand crafted by the user. Instead, it is generated automatically based on the constituent data types of the examples by the AdaptedEnumerate algorithm.

Type consistency: The manner in which the elements of the learning alphabet can be combined by STEPS during initialisation and evolution is constrained by a type system as in STGP.

Variable consistency: In addition to maintaining type consistency, the program trees generated and evolved by STEPS also maintain variable consistency in order that valid Escher programs are produced.

Specialised genetic operators: The genetic operators used by STEPS are modified in order to preserve the type and variable consistency constraints. Some additional mutation operators, tailored to the problem of concept learning have also been designed.

Specialised learning strategies: The specialised genetic operators of STEPS provide the opportunity for the design of learning strategies which allow the choice of an appropriate genetic operator to be determined according to a particular criterion. The Cover strategy allows STEPS to choose a genetic operator according to the relative completeness and consistency of a randomly selected individual. The basic idea behind the Depth controlling strategy is to grow or shrink program trees to fit the problem. If a program tree is considered to be too big, then a genetic operator that is likely to reduce its size is chosen. This strategy is useful when solving real world problems when the depth of the optimal solution is unknown. The Hybrid strategy is a combination of the Cover and Depth strategies. Both the Hybrid and Depth learning strategies were used to learn highly competitive concept descriptions for the difficult real world problem of PTE2.

λ-abstractions: In a similar manner to the STGP system PolyGP (Yu and Clack 1998*b*), STEPS also provides λ-abstractions as arguments to higher-order functions to invent new definitions for functions that are not contained in the original alphabet.

7.1.2 Relationship between Predicate Invention and Automatically Defined Functions

There are many connections and parallels between the motivations for Predicate Invention used in Inductive Logic Programming, and the use of Automatically Defined Functions and λ-abstractions, used in conjunction with evolutionary learning systems. Both automatically defined functions and reformulation approaches to predicate invention are motivated by a need to compress the program code that results from their corresponding learning processes. Both approaches achieve compression of the program code by learning additional functions or predicates that are to be used in the body of the main program. In this thesis the connection between reformulation predicate invention and the automatically defined functions is established.

7.1.3 Good Solutions to the PTE2 Problem Achieved with STEPS

The main problem to which STEPS was applied in this thesis was the PTE2 problem. This problem has been posed as an International challenge to the Machine Learning community. The PTE2 challenge involves the prediction of the Carcinogenic activity of a test set involving 30 chemical compounds. The PTE2 problem is a challenging real world problem. STEPS was applied to two forms of the data provided, i.e., with and without the non-structural attributes of the molecule obtained through laboratory tests. One of the two

78% accurate hypotheses produced by STEPS was submitted to the PTE2 challenge and was subsequently placed in joint second place. This hypothesis has the advantage over the competing hypotheses submitted that it is formed in terms of characteristics of the atom and bond connectives of the chemical compounds and therefore does not rely on the outcome of the expensive laboratory bioassays and other chemical tests.

7.1.4 Problem of the Selection of Best Hypothesis

The best hypothesis for a particular learning setting is often chosen according to the best-of-run hypotheses' performance on the test data in evolutionary communities. The broader Machine Learning community appears to be uncomfortable with this process. In this thesis, two approaches that attempt to resolve the problem of choosing among hypotheses of equal quality when a single solution is required / allowed are presented in the context of the PTE2 challenge. By *Bagging* the classifications of all the hypotheses generated by a particular setting, an accuracy of 87% was obtained. This result puts STEPS in joint first position for the PTE2 challenge. The results obtained by evolving a population of superior best-of-run individuals obtained from a particular setting in a Lamarkian fashionthrough seeding were not convincing. However, this remains an interesting idea that should be investigated further for other problems and is therefore a consideration for further work.

7.2 Limitations

While STEPS has been successful in its application to a number of concept learning problems, there is still room for improvement, and with this in mind, this section details two of the most significant limitations of STEPS.

Generic and Polymorphic Functions: STEPS does not allow the evolution of Generic and Polymorphic functions used when carrying out program synthesis. Since this thesis focuses on the problem of concept description, such functions are not strictly necessary. Concept learning is a special case of program synthesis where the problem is essentially a classification task involving the mapping of examples to relatively few classes. The mapping takes the form of a description of the concept in terms of the characteristics of the

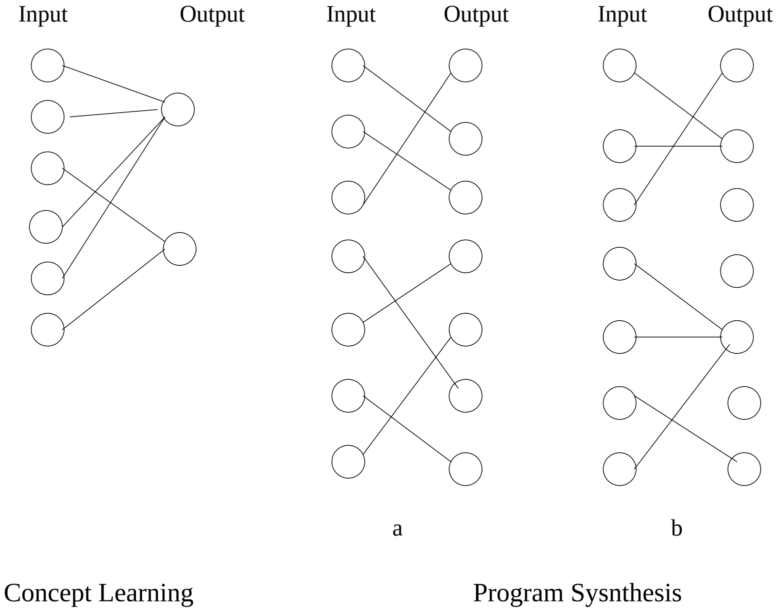


Figure 7.1: Concept learning versus program synthesis

training examples. In the more general case, program synthesis involves finding a mapping from a number of inputs to infinitely many outputs. This difference in the mappings can be viewed diagramatically in figure 7.1. For program synthesis the mappings can either be many-to-one (e.g. a sorting program can map a number of permutations of a list to a single sorted list, see figure 7.1 (b)), or a one-to-one mapping (e.g. a list reversing program will map a single list to a single reversed list, see figure 7.1 (a)). In both of these cases the mapping is formed by combining operations that may be performed on the example inputs to calculate a result. It is often desirable that only positive examples are presented to the learner and it is often assumed that these examples are 100% correct. In addition to this, it is often required that the output of the synthesized program be 100% correct. This is in contrast to the more specific case of concept learning where some tolerance in the accuracy of the learnt description is allowed to account for the possibility of noisy examples.

STEPS could clearly be extended to work on program synthesis, in which case, it would be desirable to design, within STEPS, methods for the evolution of Generic and Polymorphic functions. However, the learning strategies, genetic operators and method for obtaining the alphabet would have to be reconsidered as they are all biased towards the problem of concept learning in the present system.

Unification of Types: The current implementation of STEPS does not allow the unification of types, as found in the STGP system PolyGP (Yu and Clack 1998*b*). The effect of unification in STEPS would be mainly to increase the applicability of the crossover operator, since variable consistency would be relaxed by allowing consistency up to renaming. Of course, unification brings in additional computational complexity. In the problems considered here, and based on empirical evidence, unification was left out.

7.3 Further Work

The following outlines several potential topics of further research arising from the work presented in this thesis.

7.3.1 Variable Renaming

The current implementation of STEPS does not allow two variables with different names but the same type to be considered as equivalent. Along with type and variable consistency, this can further restrict the application of the crossover operator. An extension of the implementation of STEPS based on the unification of variables would increase the applicability of the crossover operator.

7.3.2 Enhancement of the data structures

The Enumerate algorithm has been recently extended to include trees, graphs, and multisets. These data types were not strictly necessary for the problems considered in this thesis, but the AdaptedEnumerate algorithm can be similarly extended. In particular the graph data type may prove useful in obtaining a solution to the PTE3 challenge (see later).

7.3.3 Seeding

Although the results of the seeding experiments with the PTE2 dataset were not convincing, the basic principles of the idea offer potential and should therefore be further investigated.

7.3.4 PTE3

A third round of the Predictive Toxicology Evaluation challenge series is due to be launched early next year. It will be interesting to employ all the knowledge gained by applying STEPS to the PTE2 data set to this problem.

Bibliography

Aha, D., D. Kibler and M. Albert (1991). Instance-based learning algorithms. *Machine Learning* **6**, 37–66.

Angeline, P. J. (1994). Genetic programming and emergent intelligence. In: *Advances in Genetic Programming* (K. E. Kinnear, Jr., Ed.). Chap. 4, pp. 75–98. MIT Press. Cambridge, MA, USA.

Angeline, P. J. (1996). Two self-adaptive crossover operators for genetic programming. In: *Advances in Genetic Programming 2* (P.J. Angeline and K.E. Kinnear, Jr., Eds.). Chap. 5, pp. 89–110. MIT Press. Cambridge, MA, USA.

Angeline, P. J. (1997). Comparing subtree crossover with macromutation. In: *Proceedings of the 6th International Conference on Evolutionary Programming* (J.R. McDonnell P.J. Angeline, R.G. Reynolds and R. Eberhart, Eds.). Vol. 1213 of *LNCS*. Springer. Berlin, Germany. pp. 101–111.

Angeline, P. J. and J. B. Pollack (1993). Evolutionary module acquisition. In: *Proceedings of the Second Annual Conference on Evolutionary Programming* (D. Fogel and W. Atmar, Eds.). La Jolla, CA, USA. pp. 154–163.

Angeline, P. J. and Kinnear, Jr., K.E., Eds.) (1996). *Advances in Genetic Programming 2*. The MIT Press. Cambridge, MA.

Bahler, D. R. and D. W. Bristol (1993). The induction of rules for predicting chemical carcinogenesis in rodents. In: *Proceedings of the 1st International Conference on Intelligent Systems for Molecular Biology* (L. Hunter, D. Searls and J. Shavlik, Eds.). AAAI Press. Menlo Park, CA, USA. pp. 29–37.

Bain, M. and A. Srinivasan (1995). Incremental inductive logic programming from large scale unstructured data. In: *Machine Intelligence* (D. Michie, S. Muggleton and K. Furukawa, Eds.). Vol. 14. pp. 233–267. Oxford University Press. Oxford.

Bain, M. and S. Muggleton (1994). Learning optimal chess strategies. In: *Machine Intelligence* (S. Muggleton, D. Michie and K. Furukawa, Eds.). Vol. 13. pp. 291–310. Oxford University Press. Oxford.

Banerji, R.B. (1992). Learning theoretical terms. In: *Inductive Logic Programming* (S. Muggleton, Ed.). pp. 93–112. Academic Press. London.

Banzhaf, W., P. Nordin, R. E. Keller and F. D. Francone (1998). *Genetic Programming - An Introduction.* Morgan Kaufmann Publishers. San Francisco, CA.

Blickle, T. (1996). Evolving compact solutions in genetic programming: A case study. In: *Parallel Problem Solving From Nature IV. Proceedings of the International Conference on Evolutionary Computation* (H.-M. Voigt, W. Ebeling, I. Rechenberg and H.-P. Schwefel, Eds.). Vol. 1141 of *LNCS*. Springer-Verlag. Berlin, Germany. pp. 564–573.

Blickle, T. and L. Thiele (1994). Genetic programming and redundancy. In: *Genetic Algorithms within the Framework of Evolutionary Computation (Workshop at KI-94, Saarbrücken)* (J. Hopf, Ed.). Max-Planck-Institut für Informatik (MPI-I-94-241). Im Stadtwald, Building 44, D-66123 Saarbrücken, Germany. pp. 33–38.

Bongard, M. (1970). *Pattern Recognition.* Spartan Books.

Bowers, A. F., C. Giraud-Carrier and J. W. Lloyd (1999). Higher-order logic for knowledge representation in inductive learning. In preparation.

Bowers, A. F., C. Giraud-Carrier, J. W. Lloyd C. J. Kennedy and R. Mackinney-Romero (1997). A framework for higher-order inductive machine learning. In: *Proceedings of CompulogNet Area Meeting on "Computational Logic and Machine Learning"* (P. Flach, Ed.). pp. 19–25.

Bristol, D. (1999). Personal Communication.

Bristol, D. W., J. T. Wachsman and A. Greenwell (1996). The NIEHS predictive toxicology evaluation project. *Environmental Health Perspectives* pp. 1001–1010. Supplement 3.

Cameron-Jones, R. M. and J. R. Quinlan (1993). Avoiding pitfalls when learning recursive theories. In: *Proceedings of the 13th International Joint Conference on Artificial Intelligence* (R. Bajcsy, Ed.). Morgan Kaufmann. pp. 1050–1057.

Chellapilla, K. (1997). Evolutionary programming with tree mutations: Evolving computer programs without crossover. In: *Genetic Programming 1997: Proceedings of the Second Annual Conference* (J.R. Koza, K. Deb, M. Dorigo, D.B. Fogel, M. Garzon, H. Iba and R.L. Riolo, Eds.). Morgan Kaufmann. Stanford University, CA, USA. pp. 431–438.

Chellapilla, K., H. Birru and R. Sathyanarayan (1998). Effectivenss of local search operators in evolutionary programming. In: *Genetic Programming 1998: Proceedings of the Third Annual Conference* (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, Eds.). Morgan Kaufmann. University of Wisconsin, Madison, Wisconsin, USA. pp. 753–761.

Clack, C. and T. Yu (1997). Performance enhanced genetic programming. In: *Proceedings of the 6th International Conference on Evolutionary Programming* (J. R. McDonnell P. J. Angeline, R. G. Reynolds and R. Eberhart, Eds.). Vol. 1213 of *LNCS*. Springer. Berlin, Germany. pp. 87–100.

Clark, P. and R. Niblett (1989). The CN2 induction algorithm. *Machine Learning* **3**, 261–284.

Davis, L. (1989). Adapting operator probabilities in genetic algorithms. In: *Proceedings of the 3rd International Conference on Genetic Algorithms* (J. David Schaffer, Ed.). Morgan Kaufmann. George Mason University. pp. 61–69.

Debnath, A., R. L. de Compadre, G. Debnath, A. Schusterman and C. Hansch (1991). Structure-activity relationship of mutagenic aromatic and heteroaromatic nitro compounds. correlation with molecular orbital energies and hydrophobicity. *Journal of Medicinal Chemistry* **34**, 796–797.

DeJong, K. A., W. M. Spears and D. F. Gordon (1993). A knowledge-intensive genetic algorithm for supervised learningusing genetic algorithms for concept learning. *Machine Learning* **13**, 161–188.

DeRaedt, L. and L. Dehaspe (1997). Clausal discovery. *Machine Learning* **26**, 99–146.

DeRaedt, L. and M. Bruynooghe (1992). Interactive concept learning and constructive induction by analogy. *Artificial Intelligence* **8**, 107–150.

Dolšak, B. and S. Muggleton (1991). The application of ILP to finite element mesh design. In: *Proceedings of the 1st International Workshop on Inductive Logic Programming* (S. Muggleton, Ed.). pp. 225–242.

Dolšak, B., I. Bratko and A. Jezernik (1994). Finite element mesh design: An engineering domain for ILP application. In: *Proceedings of the 4th International Workshop on Inductive Logic Programming* (S. Wrobel, Ed.). Vol. 237 of *GMD-Studien.* Gesellschaft für Mathematik und Datenverarbeitung MBH. pp. 305–320.

Džeroski, S. (1993). Handling imperfect data in inductive logic programming. In: *Proceedings of the 4th Scandinavian Conference on Artificial Intelligence.* IOS Press. pp. 111–125.

Esposito, F., D. Malerba and G. Semeraro (1993*a*). Automated acquisition of rules for document understanding. In: *Proceedings of the 2nd International Conference on Document Analysis and Recognition.* pp. 650–654.

Esposito, F., D. Malerba and G. Semeraro (1993*b*). Learning contextual rules in first-order logic. In: *Proceedings of the 4th Italian Workshop on Machine Learning.* pp. 111–127.

Esposito, F., D. Malerba and G. Semeraro (1994). Multistrategy learning for document recognition. *Applied Artificial Intelligence* **8**, 33–84.

Feng, C. (1992). Inducing temporal fault diagnostic rules from a qualitative model. In: *Inductive Logic Programming* (S. Muggleton, Ed.). pp. 473–494. Academic Press.

Flach, P. A. (1993). Predicate invention in inductive data engineering. In: *Proceedings of the 6th European Conference on Machine Learning* (P. Brazdil, Ed.). Vol. 667 of *LNAI*. Springer-Verlag. pp. 83–94.

Flach, P. A., C. Giraud-Carrier and J. W. Lloyd (1998). Strongly typed inductive concept learning. In: *Proceedings of the 8th International Conference on Inductive Logic Programming* (D. Page, Ed.). Vol. 1446 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag. pp. 185–194.

Flach, P. and N. Lachiche (1999). 1BC: A first-order Bayesian classifier. In: *Proceedings of the 9th International Workshop on Inductive Logic Programming* (S. Džeroski and P. Flach, Eds.). Vol. 1634 of *LNAI*. Springer. Berlin. pp. 92–103.

Fogel, L. J., A. J. Owens and M. J. Walsh (1966). *Artificial Intelligence through Simulated Evolution*. John Wiley & Sons. New York.
This work has been criticized as involving overly simple evolution of finite automata, but it was of pioneering significance at the time. For a review see.

Freund, Y. and R.E. Schapire (1996). Experiments with a new boosting algorithm. In: *Proc. 13th International Conference on Machine Learning*. Morgan Kaufmann. pp. 148–146.

Gathercole, C. (1998). An Investigation of Supervised Learning in Genetic Programming. PhD thesis. University of Edinburgh.

Gathercole, C. and P. Ross (1996). An adverse interaction between crossover and restricted tree depth in genetic programming. In: *Genetic Programming 1996: Proceedings of the First Annual Conference* (J.R. Koza, D.E. Goldberg, D.B. Fogel and R.L. Riolo, Eds.). MIT Press. Stanford University, CA, USA. pp. 291–296.

Giordana, A. and L. Saitta (1993). REGAL: an integrated system for learning relations using genetic algorithms. In: *Proceedings of the Second International Workshop on Multi Strategy Learning*. Morgan Kaufmann. pp. 234–249.

Giraud-Carrier, C. and T. Martinez (1995). An efficient metric for heterogeneous inductive learning applications in the attribute-value language. In: *Intelligent Systems (Proceedings of GWIC'94)*. Kluwer Academic Publishers. pp. 341–350.

Goldberg, D.E. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison Wesley.

Harvey, I. (1991). Species adaptation genetic algorithms: a basis for a continuing SAGA. In: *Proceedings of the First European Conference on Artificial Life. Toward a Practice of Autonomous Systems* (F. J. Varela and P. Bourgine, Eds.). MIT Press, Cambridge, MA. Paris, France. pp. 346–354.

Haynes, T. (1995). Clique detection via genetic programming. Technical Report UTULSA-MCS-95-02. The University of Tulsa.

Haynes, T. D., D. A. Schoenefeld and R. L. Wainwright (1996). Type inheritance in strongly typed genetic programming. In: *Advances in Genetic Programming 2* (P. J. Angeline and K. E. Kinnear, Jr., Eds.). Chap. 18, pp. 359–376. MIT Press. Cambridge, MA, USA.

Haynes, T., R. Wainwright, S. Sen and D. Schoenefeld (1995). Strongly typed genetic programming in evolving cooperation strategies. In: *Proceedings of the Sixth International Conference on Genetic Algorithms* (L. Eshelman, Ed.). Morgan Kaufmann Publishers, Inc.. San Francisco, CA. pp. 271–278.

Hekanaho, J. (1998). DOGMA : A GA-based relational learner. In: *Proceedings of the 8th International Conference on Inductive Logic Programming (ILP-98)* (David Page, Ed.). Vol. 1446 of *LNAI*. Springer. Berlin. pp. 205–214.

Janikow, C. Z. (1993). A knowledge-intensive genetic algorithm for supervised learning. *Machine Learning* **13**, 189–228.

Julstrom, B.R. (1995). What have you done for me lately? adapting operator probabilities in a steady-state genetic algorithm. In: *Proceeding of the Sixth International Conference on GeneticAlgorithms*. Morgan-Kauffman. pp. 81–87.

Kennedy, C. J. (1999). http://www.cs.bris.ac.uk/~kennedy/PTE2.html.

Kennedy, C.J. (1998). Evolutionary higher-order concept learning. In: *Late Breaking Papers at the Genetic Programming 1998 Conference* (John R. Koza, Ed.). Stanford University Bookstore. University of Wisconsin, Madison, Wisconsin, USA.

Kennedy, C.J. and C. Giraud-Carrier (1999*a*). An evolutionary approach to concept learning with structured data. In: *Proceedings of the Fourth International Conference on Artificial Neural Networks and Genetic Algorithms*. Springer Verlag. pp. 331–336.

Kennedy, C.J., C. Giraud-Carrier and D.W. Bristol (1999). Predicting carcinogenesis using structural information only. In: *Proceedings of the third European Conference on the Priciples of Data Mining and Knowledge Discovery*. Springer. pp. 360–365.

Kennedy, Claire J. and Christophe Giraud-Carrier (1999*b*). A depth controlling strategy for strongly typed evolutionary programming. In: *Proceedings of the Genetic and Evolutionary Computation Conference* (W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela and R. E. Smith, Eds.). Vol. 1. Morgan Kaufmann. Orlando, Florida, USA. pp. 879–885.

Khan, K., S. Muggleton and R. Parsons (1998). Repeat learning using predicate invention. In: *Proceedings of the Eigth International Workshop on Inductive Logic Programming* (D. Page, Ed.). Vol. 1446 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag.

Kijsirikul, B., M. Numao and M. Shimura (1992). Discrimination-based constructive induction of logic programs. In: *Proceedings of the 10th National Conference on Artificial Intelligence* (W. Swartout, Ed.). MIT Press. San Jose, CA. pp. 44–49.

King, R. and A. Srinivasan (1995). Relating chemical activity to structure: An examination of ILP successes. *New Generation Computing, Special Issue on Inductive Logic Programming* **13**, 411–434.

King, R., S. Muggleton, A. Srinivasan and M. Sternberg (1996). Structure-activity relationships derived by machine learning: The use of atoms and their bond connectivities to predict mutagenicity in inductive logic programming. *Proceedings of the National Academy of Sciences* **93**, 438–442.

King, R.D., S. Muggleton, R.A. Lewis and M.J.E. Sternberg (1992). Drug design by machine learning: the use of inductive logic programming to model the structure activity relationships of trimethoprim analogues binding to dihydrofolate reductase. *Proceedings of the National Academy of Sciences of the United States of America* **89**(23), 11322–11326.

Kinnear, Jr., K.E., Ed.) (1994). *Advances in Genetic Programming.* MIT Press. Cambridge, MA, USA.

Klingspor, V. (1994). GRDT: Enhancing model-based learning for its application in robot navigation. In: *Proceedings of the 4th International Workshop on Inductive Logic Programming* (S. Wrobel, Ed.). Vol. 237 of *GMD-Studien.* Gesellschaft für Mathematik und Datenverarbeitung MBH. pp. 107–122.

Klingspor, V., K. J. Morik and A. D. Rieger (1996). Learning concepts from sensor data of a mobile robot. *Machine Learning* **23**, 305–332.

Koza, J. R. (1994). *Genetic Programming II.* The MIT Press. Cambridge, Massachusetts.

Koza, J. R., D. Andre, F. H Bennett III and M. Keane (1999). *Genetic Programming 3: Darwinian Invention and Problem Solving.* Morgan Kaufman.

Koza, J.R. (1991). Concept formation and decision tree induction using the genetic programming paradigm. In: *Parallel Problem Solving from Nature* (H.-P. Schwefel and R. Männer, Eds.). Springer Verlag. pp. 124–128.

Koza, J.R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* The MIT Press. Cambridge, Massachusetts.

Langdon, W. B. and R. Poli (1997). Fitness causes bloat. In: *Soft Computing in Engineering Design and Manufacturing* (P. K. Chawdhry, R. Roy and R. K. Pant, Eds.). Springer-Verlag London. pp. 13–22.

Langdon, W. B. and R. Poli (1998). Fitness causes bloat: Mutation. In: *Proceedings of the First European Workshop on Genetic Programming* (W. Banzhaf, R. Poli,

M. Schoenauer and T. C. Fogarty, Eds.). Vol. 1391 of *LNCS*. Springer-Verlag. Paris. pp. 37–48.

Langdon, W.B. (1998). *Genetic Programming and Data Structures: Genetic Programming + Data Structures = Automatic Programming!*. Kluwer Academic Publishers. Amsterdam.

Langdon, W.B., T. Soule, R. Poli and J.A. Foster (1999). The evolution of size and shape. In: *Advances in Genetic Programming 3* (L. Spector, W.B. Langdon, U-M O'Rielly and Peter J. Angeline, Eds.). Chap. 8, pp. 163–190. MIT Press. Cambridge, MA, USA.

Langley, P. (1996). *Elements of Machine Learning*. Morgan Kaufmann.

Lavrač, N. and S. Džeroski (1994). *Inductive Logic Programming: Techniques and Applications.*. Artificial Intelligence. Ellis Horwood (Simon & Schuster).

Lee, Y., B.G. Buchanan and H.R. Rosenkranz (1996). Carcinogenicity predictions for a group of 30 chemicals undergoing rodent cancer bioassays based on rules derived from subchronic organ toxicities. *Environ Health Perspect* **104**(Suppl 5), 1059–1064.

Ling, C. X. (1991). Inventing necessary theoretical terms. Technical Report Nr. 302. Dept. of Computer Science, University of Western Ontario.

Ling, C. X. (1994). Learning the past tense of English verbs: The symbolic pattern associator vs. connectionist models. *Journal of Artificial Intelligence Research* **1**, 209–229.

Ling, C. X. (1995). Introducing new predicates to model scientific revolution. *International Studies in the Philosophy of Science.*

Lloyd, J. W. (1999). Programming in an integrated functional and logic language. *Journal of Functional and Logic Programming.*

Lloyd, J.W. (1995). Declarative programming in Escher. Technical Report CSTR-95-013. Department of Computer Science, University of Bristol.

Malerba, D. (1993). Document understanding: A machine learning approach. Technical report. Esprit Project 5203 INTERRID.

Matheus, C. (1991). The need for constructive induction. In: *Proceedings of the Eighth International Workshop on Machine Learning.* Morgan Kaufmann. pp. 173–177.

Michalewicz, Z. (1992). *Genetic Algorithms + Data Structures = Evolution Programs.* Springer-Verlag. Heidelberg, Germany.

Michalski, R.S. (1983). A theory and methodology of inductive learning. *Artificial Intelligence* **20**, 111–161.

Michalski, R.S. and J.B. Larson (1977). Inductive inference of VL decision rules. In: *Workshop on Pattern-directed Inference Systems.* pp. 34–44.

Michie, D. and R. Camacho (1994). Building symbolic representations of intuitive real-time skills from performance data. In: *Intelligence 13* (K. Furukawa, D. Michie and S. Muggleton, Eds.). pp. 385–418. Oxford University Press.

Mitchell, T.M. (1977). Version spaces: A candidate elimination approach to rule learning. In: *Proceedings of the Fifth International Joint Conference on Artificial Intelligence (IJCAI-77).* IJCAII. Cambridge, Massachusetts. pp. 305–310.

Mitchell, T.M. (1982). Generalization as search. *Artificial Intelligence* **18**, 203–206.

Mitchell, T.M. (1997). *Machine Learning.* McGraw-Hill.

Mitchell, T.M. (n.d.). Does machine learning really work. *Artificial Intelligence Magazine* **18**(3), 11–20.

Montana, D.J. (1995). Strongly typed genetic programming. *Evolutionary Computation* **3**(2), 199–230.

Mooney, R. J. (1996). Inductive logic programming for natural language processing. In: *Proceedings of the 6th International Workshop on Inductive Logic Programming* (S. Muggleton, Ed.). Stockholm University, Royal Institute of Technology. pp. 205–224.

Mooney, R. J. and M. E. Califf (1995). Induction of first–order decision lists: Results on learning the past tense of English verbs. *Journal of Artificial Intelligence Research* **3**, 1–24.

Muggleton, S. (1992*a*). Inductive logic programming. In: *Inductive Logic Programming* (S. Muggleton, Ed.). Chap. 1. Academic Press. London.

Muggleton, S. (1994). Predicate invention and utility. *Journal of Experimental and Theoretical Artificial Intelligence* **6**(1), 127–130.

Muggleton, S. (1995). Inverse entailment and progol. *New Generation Computing* **13**, 245–286.

Muggleton, S., A. Srinivasan, R. D. King and M. J. E. Sternberg (1998). Biochemical knowledge discovery using inductive logic programming. In: *Proceedings of the 1st International Conference on Discovery Science (DS-98)* (S. Arikawa and H. Motoda, Eds.). Vol. 1532 of *LNAI*. Springer. Berlin. pp. 326–341.

Muggleton, S. and C.D. Page (1994). Beyond first-order learning: Inductive learning with higher order logic. Technical Report PRG-TR-13-94. Oxford University Computing Laboratory.

Muggleton, S. and L. DeRaedt (1994). Inductive logic programming: Theory and methods. *Journal of Logic Programming* **19/20**, 629–679.

Muggleton, S. and W. Buntine (1988). Machine invention of first-order predicates by inverting resolution. In: *Proceedings Fifth International Workshop on Machine Learning*. Morgan Kaufmann. San Mateo, CA. pp. 339–352.

Muggleton, S., Ed.) (1992*b*). *Inductive Logic Programming*. Academic Press. London.

Muggleton, S. H., M. Bain, J. Hayes-Michie and D. Michie (1989). An experimental comparison of human and machine learning formalisms. In: *Proc. Sixth International Workshop on Machine Learning*. Morgan Kaufmann. San Mateo, CA. pp. 113–118.

Muggleton, S., R. D. King and M. J. E. Sternberg (1992). Protein secondary structure prediction using logic-based machine learning. *Protein Engineering* **5**(7), 647–657.

NIE (1999). http://dir.niehs.nih.gov/dirlecm/pte2.htm.

Nienhuys-Cheng, S.-H. and R. de Wolf (1997). *Foundations of Inductive Logic Programming*. LNAI Tutorial, Springer.

Nikolaev, N. I. and V. Slavov (1997). Inductive genetic programming with decision trees. In: *Proceedings of the 9th European Conference on Machine Learning* (M. van Someren and G. Widmer, Eds.). Vol. 1224 of *LNAI*. Springer. Berlin. pp. 183–190.

Nilsson, N.T. (1980). *Machine Learning Principles of Artificial Intelligence*. Tioga.

Nordin, P. (1997). *Evolutionary Program Induction of Binary Machine Code and its Application*. Krehl Verlag. Munster, Germany.

Osborn, T. R., A. Charif, R. Lamas and E. Dubossarsky (1995). Genetic logic programming. In: *1995 IEEE Conference on Evolutionary Computation*. Vol. 2. IEEE Press. Perth, Australia. p. 728.

OUC (1997). http://www.comlab.ox.ac.uk/oucl/groups/machlearn/PTE/.

Oxf (1999). http://www.comlab.ox.ac.uk/oucl/groups/machlearn/index.html.

Quinlan, J. R. (1996). Bagging, boosting, and C4. 5. In: *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference*. AAAI Press / MIT Press. Menlo Park. pp. 725–730.

Quinlan, J.R. (1986). Induction of decision trees. *Machine Learning* **1**, 81–106.

Quinlan, J.R. (1990). Learning logical definitions from relations. *Machine Learning* **5**, 239–266.

Quinlan, J.R. (1993). *C4.5: Programs for Machine Learning*. Morgan Kaufmann. San Mateo, CA.

Reiser, P.G.K. and P.J. Riddle (1999). Evolution of logic programs: Part of speech tagging. In: *Proceedings of the Congress on Evolutionary Computation* (P. J. Angeline, Z. Michalewicz, M. Schoenauer, X. Yao and A. Zalzala, Eds.). Vol. 2. IEEE Press. New Jersey, USA. pp. 1338–1346.

Rosca, J. P. and D. H. Ballard (1996). Discovery of subroutines in genetic programming. In: *Advances in Genetic Programming 2*. pp. 177–202. MIT Press.

Rosca, J.P. and D.H. Ballard (1995). Causality in genetic programming. In: *Proceeding of the Sixth International Conference on GeneticAlgorithms*. Morgan-Kauffman. pp. 256–263.

Rouveirol, C. (1992). Extensions of inverse resolution applied to theory completion. In: *Inductive Logic Programming* (S. Muggleton, Ed.). Chap. 3, pp. 63–92. Academic Press. London.

Rouveirol, C. (1994). Flattening and saturation: Two representaion changes for generalisation. *Machine Learning* **14**, 219–232.

Rumelhart, D. E. and J. L. McClelland (1986). *Parallel Distributed processing: exploration in the microstructure of cognition*. Vol. 1 & 2. MIT Press.

Schultz, A. C. and J. J. Grefenstette (1990). Improving tactical plans with genetic algorithms. In: *Proceedings of the 2nd International IEEE Conference on Tools for Artificial Intelligence*. number IEEE Cat. No. 90CH2915-7. IEEE Computer Society Press, Los Alamitos, CA. Herndon, VA. pp. 328–334.

Sebag, M. (1998). http://www.lri.fr/fabien/PTE/Distill/.

Semeraro, G., F. Esposito and D. Malerba (1994). Learning contextual rules for document understanding. In: *Proceedings of the 10th Conference on Artificial Intelligence for Applications* (Dan O'Leary and Peter Selfridge, Eds.). IEEE Computer Society Press. San Antonio, TX. pp. 108–115.

Sharif, A. M. and A. N. Barrett (1998). Seeding a genetic population for mesh optimisation and evaluation. In: *Late Breaking Papers at the Genetic Programming 1998 Conference* (J. R. Koza, Ed.). Stanford University Bookstore. University of Wisconsin, Madison, Wisconsin, USA. pp. 195–200.

Silverstein, G. and M.J. Pazzani (1993). Learning relational clichés. In: *Proceedings of the IJCAI-93 Workshop on Inductive Logic Programming* (F. Bergadano, L. De Raedt, S. Matwin and S. Muggleton, Eds.). Morgan Kaufmann. pp. 71–82.

SNN (1996). http://www-ra.informatik.uni-tuebingen.de/SNNS/.

Soule, T., J.A. Foster and J. Dickinson (1996). Code growth in genetic programming. In: *Genetic Programming 1996: Proceedings of the First Annual Conference* (J. R. Koza, D. E. Goldberg, D. B. Fogel and R. L. Riolo, Eds.). MIT Press. Stanford University, CA, USA. pp. 215–223.

Spector, L. (1996). Simultaneous evolution of programs and their control structures. In: *Advances in Genetic Programming 2* (P. Angeline and K.E. Kinnear, Eds.). The MIT Press. Cambridge, MA. pp. 137–154.

Spector, L., Langdon, W. B., O'Reilly, U.-M. and Angeline, P., Eds.) (1999). *Advances in Genetic Programming 3.* The MIT Press. Cambridge, MA.

Srinivasan, A. and R. King (1997). Carcinogenisis predictions using ILP. In: *Proceedings of the Seventh Inductive Logic Programming Workshop.* Vol. 1297 of *LNAI.* Springer. Berlin. pp. 273 – 287.

Srinivasan, A., R. King and S. Muggleton (1996). The role of background knowledge: Using a problem from chemistry to examine the performance of an ILP program. In: *Intelligent Data Analysis in Medicine and Pharmacology.* Kluwer Academic Press.

Srinivasan, A., R.D. King, S.H. Muggleton and M. Sternberg (1997). The predictive toxicology evaluation challenge. In: *Proceedings of the Fifteenth International Joint Conference Artificial Intelligence (IJCAI-97).* Morgan-Kaufmann.

Srinivasan, A., S. Muggleton and M. Bain (1992). Distinguishing exceptions from noise in non-monotonic learning. In: *Proceedings of the 2nd International Workshop on Inductive Logic Programming* (S. Muggleton, Ed.). Report ICOT TM-1182.

Srinivasan, A., S. Muggleton, R. D. King and M. J. E. Sternberg (1995*a*). The effect of background knowledge in inductive logic programming: A case study. Technical report. PRG-TR-9-95 Oxford University Computing Laboratory.

Srinivasan, A., S. Muggleton, R. D. King and M. J. E. Sternberg (1995*b*). Theories for mutagenecity: A study of first-order and feature based induciton. Technical report. PRG-TR-8-95 Oxford University Computing Laboratory.

Srinivasan, A., S. Muggleton, R. King and M. Sternberg (1994). Mutagenesis: ILP experiments in a non-determinate biological domain. In: *Proceedings of Fourth Inductive Logic Programming Workshop* (S. Wrobel, Ed.). Gesellschaft für Mathematik und Datenverarbeitung MBH. pp. 217 – 232.

Srinvas, M. and L.M. Patnaik (1994). Adaptive probabilities of crossover and mutation in genetic algorithms. *IEEE Transactions on Systems, Man, and Cybernetics* **24**, 656–667.

Stahl, I. (1993). Predicate invention in ILP - an overview. In: *Proceedings of the European Conference on Machine Learning* (P. B. Brazdil, Ed.). Vol. 667 of *LNAI*. Springer. Berlin. pp. 313–322.

Stahl, I. (1995). The appropriateness of predicate invention as bias shift operation in ILP. *Machine Learning* **20**(1/2), 95–118.

Sutton, R.S. (1992). Adapting bias by gradient descent: An incremental version of delta-bar-delta. In: *Proceedings of the Tenth International Conference on Artificial Intellignece* (W. Swartout, Ed.). MIT Press/AAAI Press. San Jose, CA. pp. 171–176.

Sutton, R.S. (1994). Constructive induction needs a methodology based on continuing learning. In: *Panel of the Workshop on Constructive Induction and Change of Representation at Eleventh International Conference on Machine Learning.*

Syswerda, G. (1989). Uniform crossover in genetic algorithms. In: *Proceedings of the 3rd International Conference on Genetic Algorithms* (J. D. Schaffer, Ed.). Morgan Kaufmann. George Mason University. pp. 2–9.

Tackett, W.A. (1994). Recombination, Selection, and the Genetic Construction of Computer Programs. PhD thesis. University of Southern California, Department of Electrical Engineering Systems.

Tucker, A. and X. Liu (1999). Extending evolutionary programming methods to the learning of dynamic bayesian networks. In: *Proceedings of the Genetic and Evolutionary Computation Conference* (W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon,

V. Honavar, M. Jakiela and R. E. Smith, Eds.). Vol. 1. Morgan Kaufmann. Orlando, Florida, USA. pp. 923–929.

UCI (1999). http://www.ics.uci.edu/~mlearn/MLRepository.html.

van Hemert, J.I. and A.E. Eiben (1997). Comparison of the SAW-ing evolutionary algorithm and the grouping genetic algorithm for graph coloring. Technical Report TR-97-14. Leiden University.

Wer (1992). Thyroid training and test data: ftp://ftp.ics.uci.edu/pub/machine-learning-databases/thyroid-disease/ann-Readme.

Wilson, D.R. and T.R. Martinez (1995). Improved heterogeneous distance functions. *Journal of Artificial Intelligence Research* **6**, 1–34.

Wirth, R. (1989). Completing logic programs by inverse resolution. In: *Proceedings of the Fourth European Working Session on Learning.* Pitman. London. pp. 239–250.

Wirth, R. and P. O'Rorke (1992). Constraints for predicate invention. In: *Inductive Logic Programming* (S. Muggleton, Ed.). pp. 299–318. Academic Press.

Wnek, J. and R.S. Michalski (1994). Hypothesis driven constructive induction in AQ17-HCI: A method and experiments. *Machine Learning* **14**, 139–168.

Wogulis, J. and P. Langley (1989). Improving efficiency by learning intermediate concepts. In: *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence.* Morgan Kauffman. Los altos, CA.. pp. 657–662.

Wolpert, D. H. and W. G. Macready (1995). No free lunch theorems for search. Technical Report SFI-TR-95-02-010. Santa Fe Institute. Santa Fe, NM.

Wong, M. L. and K. S. Leung (1995). Genetic logic programming and applications. *IEEE Expert.*

Wrobel, S. (1994). Concept formation during interactive theory revision. *Machine Learning* **14**(2), 169–191.

Yu, T. and C. Clack (1998*a*). PolyGP: A polymorphic genetic programming system in Haskell. In: *Genetic Programming 1998: Proceedings of the Third Annual Conference.* Morgan Kaufmann. pp. 416–421.

Yu, T. and C. Clack (1998*b*). Recursion, lambda abstractions and genetic programming. In: *Genetic Programming 1998: Proceedings of the Third Annual Conference.* Morgan Kaufmann.

Zelle, J. M. and R. J. Mooney (1993). Learning semantic grammars with constructive inductive logic programming. In: *Proceedings of the 11th National Conference on Artificial Intelligence.* AAAI Press. Menlo Park, CA, USA. pp. 817–823.

Zelle, J. M. and R. J. Mooney (1994). Inducing deterministic prolog parsers from treebanks: A machine learning approach. In: *Proceedings of the 12th National Conference on Artificial Intelligence. Volume 1.* AAAI Press. Menlo Park, CA, USA. pp. 748–753.

Zelle, J. M. and R. J. Mooney (1996). Learning to parse database queries using inductive logic programming. In: *Proceedings of the 14th National Conference on Artificial Intelligence.* AAAI Press. Menlo Park, CA, USA. pp. 1050–1055.

Zhang, B.-T. and H. Mühlenbein (1996). Adaptive fitness functions for dynamic growing / pruning of program trees. In: *Advances in Genetic Programming 2* (P. J. Angeline and K. E. Kinnear, Jr., Eds.). pp. 241–256. MIT Press. Cambridge, MA, USA.