# Evolutionary program induction directed by logic grammars

THESIS

Presented to the Department of Computer Science of The Chinese University of Hong Kong in partial fulfillment of the requirements for the Degree of Doctor of Philosophy

by

Wong Man Leung

June 1995

# ABSTRACT

Program induction generates a computer program with the desired behavior for a given set of situations. Genetic Programming (GP) and Inductive Logic Programming (ILP) are two of the approaches for program induction. GP is a method of automatically inducing S-expressions in Lisp to perform specified tasks while ILP involves the construction of logic programs from examples and background knowledge.

Since their formalisms are very different, these two approaches cannot be integrated easily although their properties and goals are similar. If they can be combined in a common framework, then their techniques and theories can be shared and their problem solving power can be enhanced.

This thesis describes a framework that integrates GP and ILP based on a formalism of logic grammars. A system called LOGENPRO (the LOgic grammar based GENetic PROgramming system) is developed. This system has been tested on many problems in program induction, knowledge discovery from databases, and meta-level learning. These experiments demonstrate that the proposed framework is powerful, flexible, and general.

Experiments are performed to illustrate that programs in different programming languages can be induced by LOGENPRO. The problem of inducing programs can be formulated as a search for a highly fit program in the space of all possible programs. This thesis shows that the search space can be specified declaratively by the user in the framework. Moreover, the formalism is powerful enough to represent context-sensitive information and domain-dependent knowledge. This knowledge can be used to accelerate the learning speed and/or improve the quality of the programs induced.

Knowledge discovery systems induce knowledge from datasets which are huge, noisy (incorrect), incomplete, inconsistent, imprecise (fuzzy), and uncertain. The problem is that existing systems use a limiting attribute-value language for representing the training examples and induced knowledge. Furthermore, some important patterns are ignored because they are statistically insignificant. LOGENPRO is employed to induce knowledge from noisy training examples. The knowledge is represented in first-order logic program. The performance of LOGENPRO is evaluated on the chess endgame domain. Detailed comparisons with other ILP systems are performed. It is found that LOGENPRO outperforms these ILP systems significantly at most noise levels. This experiment indicates that the Darwinian principle of natural selection is a plausible noise handling method which can avoid overfitting and identify important patterns at the same time.

An Adaptive Inductive Logic Programming (Adaptive ILP) system is implemented using LOGENPRO as the meta-level learner. The system performs better than FOIL in inducing logic programs from perfect and noisy training examples. The result is very encouraging as it suggests that LOGENPRO can successfully evolve a high performance ILP system.

# ACKNOWLEDGMENTS

I am very grateful to Dr. K. S. Leung who is my thesis advisor. He gave me his trust from the beginning to the end of my thesis. He gave excellent supervision and sound critiques of this project. I would also like to thank the members of the thesis committee: Professor Watada, Dr. Horace Ip, Dr. Jimmy Lee, and Dr. M. H. Wong. They provided valuable suggestions to improve the thesis. I acknowledge the members of the soft computing group: Dr. Y. Leung, Dr. T. C. Chan, Dr. K. S. Leung, and Clement Lee for their suggestions and interesting ideas. Danny Luk and H. K. Chan are very kind to proofread the drafts of the thesis.

I would like to acknowledge the Department of Computer Science and Engineering of the Chinese University of Hong Kong and the Croucher Foundation for the support of this research project.

# Table of Contents

# List of Figures

# List of Tables

# Chapter 1
# Introduction

Program induction is a method for automatic programming. The relation between them is discussed in the first section. The next section describes the motivation of this thesis and where the main theme of this thesis on evolutionary approach fits in the overall picture of automatic programming. The contributions of this research are summarized in section 1.3. The last section is an outline of the thesis.

## 1.1. Automatic programming and program induction

The software life cycle consists of the tasks of requirement acquisition, specification formation, software analysis, design, implementation, verification, validation and documentation. Researchers in the field of automatic programming investigate how to automate these tasks. The ultimate goal of automatic programming is to solve current software development problems by eliminating most of the software engineers currently required. This approach only requires the users to write specifications for what they want and a fully automatic system then systematically generate programs satisfying these specifications. These specifications can be complete or partial, formal or informal. Program generators have been successfully developed for a number of specific narrow application domains.

However, an end-user oriented, general purpose, and fully automatic system is still not a realistic goal. Rich and Waters (1988) identify several myths of automatic programming and indicate three achievable approaches to automatic programming. These approaches are:

- **Non-end-user oriented**: This approach tries to automate the tasks of program analysis, design, development, verification and documentation.

- **Narrow domain**: This approach focuses on a narrow enough domain so that it is possible to construct a fully automatic system that can communicate with the user directly.
- **Semi-automatic**: This approach attempts to provide various tools to assist in different aspects of programming. It also integrates these tools and improves the performance of individual tools.

An automatic programming system can be classified by either the types of specifications provided by the user or the mechanisms employed to generate executable programs from specifications (Rich and Waters 1988). Specifications can be described in different natural languages, special-purpose languages, very high-level languages, formal specification languages, and examples describing the inputs and the corresponding outputs of the desired programs.

Natural languages are efficient and convenient means for performing communication between the user and an automatic programming system. However, current researches in natural language processing still cannot provide effective and satisfactory natural language interface for this purpose.

Special-purpose languages provide domain-specific symbols, graphics and terminologies for the user and a system to communicate the requirement and the feedback effectively. However, special-purpose languages are essentially domain-specific and they are useless outside their domains of applications.

Very high-level languages extend current high-level languages by adding powerful abstract data types such as stacks, binary trees and sets. They also incorporate notations from formal logic, such as quantification over sets, to facilitate the formation of specifications.

Many formal specification languages are based on logic. Since logic is the most powerful and general formal description language known, it provides a good communication medium between the user and a system. However, most interesting problems in general logical systems are intractable and complex logical formulas are difficult to generate and understand. Consequently, these formal specification languages usually introduce restrictions and extensions to make logic more tractable to machine and human.

An effective way for specifying the behavior of a program is to enumerate examples describing the inputs and the corresponding outputs of the target program. This method is attractive because it is an easy and natural means for communicating with an automatic programming system. Moreover, the user can modify the specification easily by changing the examples. Other people can also effortlessly understand the behavior of the program generated by examining the set of examples.

The mechanisms employed to generate programs can be classified into procedural, deductive, inductive, transformational, knowledge-based, and inspection methods.

Procedural methods require some programmers to write a special-purpose program satisfying the specification provided by the user. Although they are the simplest and the most successful methods, they fail to automate the process of program generation.

If the specification can be formulated as a theorem stating the relation between the inputs and the corresponding outputs, then the problem of generating a program satisfying the specification is equivalent to finding a constructive proof of the satisfiability of the specification. Thus, any method of automated deduction can be used to support automatic programming. Deductive methods search for an inference path from some initial states to a goal representing the specification. Since the search space is extremely large and the current deductive systems cannot control the search process effectively, these systems cannot discover complex proofs.

Inductive methods perform program induction from partial specifications, such as examples. They perform inductive inference which generalizes partial specifications to produce computer programs that can produce the desired behavior for a given set of situations. For example, if the program to be induced is a pure function and the specification is represented as a set of inputs and the corresponding set of outputs. A program induction system must search for programs having the same behavior of the target function in a search space of all possible programs. The function can be represented in any programming languages such as C, Lisp, ML, and Prolog or in mathematical logic such as lambda calculus and first-order logic.

One major dimension to classify program induction systems is by the kinds of information employed in the specifications (Olsson 1995). At one extreme are systems that use traces of computation or sets of positive and negative examples. Biermann (1972) demonstrated that flowcharts and Turing machines can be induced from example traces. Summers (1977), Biermann and Smith (1979) described systems that create programs in Lisp from examples of their behaviors. Inductive logic programming systems (Muggletion 1992) induce logic programs from examples. At the other extreme are genetic programming systems that use specifications represented as fitness functions to drive the evolution of programs in Lisp (Koza 1992; 1994, Kinnear 1994b).

Transformational methods apply a sequence of transformations to convert a specification represented in a very high-level language into a low-level implementation. The three components of a transformation are a pattern, a set of logical applicability conditions, and an action. When an instance of the pattern is found in the specification, the conditions are checked to determine whether the transformations can be employed.

If the conditions are satisfied, the action is evaluated to compute a new section of code, which is used to replace the code matched by the pattern.

A sequence of transformations forms a transformational rewrite cycle. At each step, a transformation is selected and applied to a specification to produce a modified specification. The above process is repeated until some condition is satisfied. Transformal methods search for a sequence of transformations from the initial specification to a satisfactory low-level implementation. Since the search space is extremely large and the current transformational systems cannot control the search process effectively, these systems suffer from the same problem of deductive and inductive systems.

Knowledge-based methods improve the efficiency of software development by providing knowledge-based software assistance and the software development becomes a knowledge-intensive process (Goldberg 1986). A knowledge-based assistant provides an interactive interface for the development process and enforces the semantic consistency of the program generated from the specification provided by the user. It encodes knowledge of the programming process and domain-specific knowledge to assist the software developer. A knowledge-based assistant compiles a formal, high-level specification into an efficient low-level implementation by the repeated application of correctness-preserving transformations. Since the software developer can make decision on how to perform transformations, the search problem is partially solved.

The idea of inspection methods is to produce programs by inspection rather than by reasoning from first principles. If knowledge of programming clichés are available. A program can be constructed by recognizing clichés in the specification and then choosing among various implementations of the identified cliché. The three components of a cliché are: a skeleton, roles whose contents vary from one occurrence to the others, and constraints on what can fill the roles. The process of identifying clichés in the specification and selecting implementation of the cliché can be viewed as a search problem in a very large search space. Consequently, a fully automatic, inspection system suffers from the same control problem in deductive, inductive, and transformational systems. For this reason, existing inspection systems are only semi-automatic (Rich and Waters 1990).

## 1.2. Motivation

As described in the previous section, search is an important research topic in program induction in particular and automatic programming in general. Search methods in Artificial Intelligence can be classified into weak and strong methods. Weak methods encode search strategies that are task independent and consequently less efficient. Strong methods are rich in task-specific knowledge that a programmer or knowledge engineer places explicitly into the search mechanism. Strong methods tend to be narrowly focused but fairly efficient in their abilities to identify domain-specific solutions. Strong methods often use one or more weak methods working underneath the task-specific knowledge. Since the knowledge to solve the problem is usually represented explicitly within the problem solver's knowledge base as search strategies and heuristics, there is a direct relation between the quality of knowledge and the performances of strong methods (Angeline 1993; 1994).

Different strong methods have been introduced to guide the search for the desired programs. However, these strong methods may not always work because they may trap the program induction systems in local maxima. In order to overcome this problem, weak methods or backtracking will be invoked if the systems find that they encounter troubles in the process of searching for satisfactory programs. The problem is that these approaches are very inefficient.

The alternatives are evolutionary algorithms, a kind of weak methods, which conducts parallel searches. Evolutionary algorithms perform both exploitation of the most promising solutions and exploration of the search space. It is featured to tackle hard search problems and thus it may be applicable to program induction. Although there are many researches in evolutionary algorithms, there is no study in representing domain-specific knowledge for evolutionary algorithms to produce evolutionary strong methods for the problem of program induction.

Moreover, existing program induction systems are limited in the programming languages in which the induced programs are expressed. For example, Koza proposed (1992; 1994) Genetic Programming (GP) systems which can only induce programs represented as S-expressions in Lisp. Inductive Logic Programming (ILP) systems can only produce logic programs (Muggletion 1992). Since the formalisms of these two approaches are so different, these two approaches cannot be integrated easily although their properties and goals are similar. If they can be combined in a common framework, then many of the techniques and theories obtained in one approach can be applied in the other one. The combination can  greatly enhance the overall problem solving power and the information exchange between these fields.

These observations lead us to propose and develop a framework combining GP and ILP that employs evolutionary algorithms to induce programs. The framework is driven by logic grammars and is powerful enough to represent context-sensitive information and domain-specific knowledge that can accelerate the learning of programs. It is also very flexible and programs in various programming languages such as Lisp, Prolog, Fuzzy Prolog and C can be induced.

## 1.3. Contributions of the research

The contributions of the research are listed here in the order that they appear in the thesis:

- The Genetic Logic Programming System (GLPS) is a novel system developed to combine the implicitly parallel search power of GP and knowledge representation power of first-order logic. GLPS can learn function free first-order logic programs with constants. It takes the advantages of existing ILP and GP systems while avoids the disadvantages of them. The experiments demonstrate that GLPS is a promising alternative to other ILP systems. Since GLPS uses the same representation of other ILP systems, it is possible to combine GLPS with them.
- The work in GLPS leads to the idea that a logic program can be represented as a forest of AND-OR trees. This representation method facilitates the generation of the initial population of logic programs and the operations of various genetic operators such as crossover and reproduction. A representation-dependent but domain-independent crossover operator is also introduced.
- From the experience gained in developing and applying GLPS, we propose a novel, flexible and general framework based on a formalism of logic grammars. A system called LOGENPRO (The LOgic grammar based GENetic PROgramming system) is developed. It is found that programs in different programming languages can be expressed as derivation trees. This representation method facilitates the generation of the initial population of programs and the operations of various genetic operators such as crossover and mutation. We introduce two effective

and efficient genetic operators which guarantee only valid offspring are produced.

- We have demonstrated that LOGENPRO can emulate traditional GP (Koza 1992) easily. Traditional GP has a limitation that all the variables, constants, arguments for functions, and values returned from functions must be of the same data type. This limitation leads to the difficulty of inducing even some rather simple and straightforward functional programs. It is found that knowledge of data type can be represented easily in LOGENPRO to alleviate the above problem. An experiment is performed to show that LOGENPRO can find a solution much faster than GP and the computation required by LOGENPRO is much smaller than that of GP. Another advantage of LOGENPRO is that it can emulate the effect of Strongly Type Genetic Programming (STGP) effortlessly (Montana 1993).
- Automatic discovery of problem representation primitives is one of the most challenging research areas in Genetic Programming. We have illustrated how to apply LOGENPRO to emulate Automatically Defined Functions (ADF) proposed by Koza. ADF is one of the approaches that have been proposed to acquire problem representation primitives automatically (Koza 1992; 1994). We have performed an experiment to demonstrate that, by employing various knowledge about the problem being solved, LOGENPRO can find a solution much faster than ADF and the computation required by LOGENPRO is much smaller than that of ADF. This experiment also shows that LOGENPRO can emulate the effects of STGP and ADF simultaneously and effortlessly.
- Knowledge discovery systems induce knowledge from datasets which are frequently noisy (incorrect), incomplete, inconsistent, imprecise (fuzzy) and uncertain (Leung and Wong 1991a; 1991b; 1991c). We employ LOGENPRO to combine evolutionary algorithms and a variation

of FOIL, BEAM-FOIL, in learning logic programs from noisy datasets. Detailed comparisons between LOGENPRO and other ILP systems have been conducted using the chess endgame problem. It is found that LOGENPRO outperforms these ILP systems significantly at most noise levels.

- Imprecise and uncertain examples are frequent in real world environment, because many everyday examples are denoted in linguistic terms which are essentially imprecise and uncertain. However, there are very few studies on the issue of inducing knowledge from imprecise and uncertain datasets. We have used LOGENPRO to acquire knowledge from imprecise and uncertain training examples stored in a fuzzy relational database. The induced knowledge is represented as a program in Fuzzy Prolog (Li and Liu 1990). To the knowledge of the authors, LOGENPRO is currently the only system that can learn program in Fuzzy Prolog.
- We have formulated the problem of adaptive inductive logic programming and proposed an adaptive ILP system that can evolve during learning based on evolutionary algorithms. The logical organization of this system have been designed and a prototype has been developed.
- We have demonstrated the meta-level learner, a component of our adaptive ILP system, can be implemented by applying LOGENPRO. The meta-level learner induces search biases which affect the performance of the adaptive ILP system in learning logic programs. It has been demonstrated that the induced biases are better than that of FOIL on many standard learning tasks. This result is surprising because the search biases of the Adaptive ILP system are initialized by a random process. These biases are normally poor, but the process of natural selection and evolution can successfully evolve a good bias.

## 1.4. Outline of the thesis

In chapter 2, we will first introduce a class of weak methods called evolutionary algorithms. Subsequently, four kinds of these algorithms, namely Genetic Algorithms (GAs), Genetic Programming (GP), Evolution Strategies (ES), and Evolutionary Programming (EP), will be discussed in turn.

We will describe another approach of program induction, Inductive Logic Programming (ILP), that investigates the construction of logic programs from training examples and background knowledge in chapter 3. A brief introduction to inductive concept learning will be presented first. Then, two approaches of the ILP problem will be discussed followed by an introduction to techniques and methods of ILP.

The Genetic Logic Programming System (GLPS) will be described in chapter 4. The results of some applications will also be presented. The material of this chapter have been published in a number of papers (Wong and Leung 1994a; 1994b; 1995b).

A novel, flexible and general framework that can combine GP and ILP will be described in chapter 5. A high-level description of LOGENPRO (The LOgic grammar based GENetic PROgramming system) will be presented. We will also discuss the representation method of programs, the crossover operator, and the mutation operator.

We will illustrate how to apply LOGENPRO to emulate GP and GLPS in chapter 6. Furthermore, we will demonstrate that LOGENPRO can induce programs in imperative programming languages such as C.

Three applications of LOGENPRO in acquiring knowledge from databases will be discussed in chapter 7. The knowledge acquired can be expressed in different knowledge representations such as decision tree, decision list, production rule, first-order logic and Fuzzy Prolog. In the first application, LOGENPRO is used to induce knowledge represented in decision trees from a real-world database. In the second application, we apply LOGENPRO to combine genetic search methods and FOIL to induce knowledge from noisy datasets. The acquired knowledge is represented as a logic program. The performance of LOGENPRO is evaluated on the chess endgame problem and detailed comparisons to other ILP systems are given. In the third application, LOGENPRO is employed to acquire knowledge from imprecise and uncertain training examples stored in a fuzzy relational database. The induced knowledge is represented as a program in Fuzzy Prolog.

In chapter 8, we will describe an adaptive ILP system that employs LOGENPRO to improve itself during the problem solving process. The problem to be solved here is to induce logic programs from training examples. The definition of the problem of adaptive inductive logic programming will be formulated first. We will then present a generic top-down ILP algorithm, a meta-level learner that induces search biases, and the results of the experiments conducted. The material of this chapter have been published in a paper (Wong and Leung 1995a). Finally, we will summarize the results and the original contributions of this thesis in the last chapter. A number of suggestions for future research will also be given.

# Chapter 2
# An Overview on Evolutionary Algorithms

In the previous chapter, We have presented the problem of program induction as conducting a search in the space of all possible programs. The search can be accomplished by various techniques including general weak methods and domain-specified strong methods. In this chapter, we will first introduce a class of general weak methods called evolutionary algorithms. Subsequently, four kinds of evolutionary algorithms, namely Genetic Algorithms (GAs), Genetic Programming (GP), Evolution Strategies (ES), and Evolutionary Programming (EP), will be discussed in turn.

## 2.1. Evolutionary algorithms

Evolutionary algorithms are weak search and optimization techniques inspired by natural evolution (Angeline 1993; 1994). Weak methods are a category of problem solving methods studied in the field of Artificial Intelligence (AI). In contrast to strong methods, weak methods are more general and widely applicable in different domains (Nilson 1980, Newell and Simon 1972). Weak methods do not employ problem-dependent search operators and make no commitment to specific credit assignment methods.

Problem solving methods conduct their tasks by traversing the search space of the problem. They should identify blame and/or credit (credit assignment) on the components of each search point encountered in the search space (Minsky 1963, Winston 1992). This information evaluates the qualities of all components of a search point, their interaction, and their impact on the overall quality of the search point. Problem solving methods apply this information to determine how to combine and manipulate different components from the current or past search points to produce the next search point. Thus, good credit assignment methods direct the future search towards promising regions. An efficient problem solving method embodies an excellent credit assignment method for the problem and manipulates components of various search points to traverse the search space. However, it is often difficult to design an appropriate credit assignment method for a particular problem represented in a specific representation.

On the other hand, strong methods employ domain-dependent credit assignment techniques, search strategies, and heuristics to strengthen the efficiency and ability of problem solving. They contains a significant amount of domain-specific knowledge. This knowledge can be represented procedurally or declaratively. A procedural problem solver finds an analytic solution for a problem by executing a sequence of hard-wired instructions. Thus, its knowledge is represented procedurally. A knowledge-based system (Buchanan and Shortliffe 1984) solves a problem by performing inferences. The inferences are carried out by the inference engine of the system according to the knowledge stored declaratively in the knowledge base of the system. The knowledge usually takes the forms of heuristic rules, frames, semantic nets and first-order logic (Leung and Wong 1990). This specific knowledge allows the problem solvers to find accurate solutions quickly.

Traditional weak methods are inspired by observations of human performance (Newell and Simon 1972, Winston 1992, Pearl 1984). They includes depth-first search, breadth-first search, best-first search, generate and test, hill climbing, mean-ends analysis, constraint satisfaction, and problem reduction.

On the other hand, evolutionary algorithms are inspired from the idea of achieving intelligent behavior of humans through a search and learning method (Angeline 1993; 1994). They employ the principle of natural selection and evolution to achieve the goals of function optimization and machine learning. In general, evolutionary algorithms include any population-based algorithm that uses selection and recombination operators to generate new search points in a search space. They include genetic algorithms (Holland 1992, Goldberg 1989, Davis 1991), genetic programming (Koza 1992; 1994, Kinnear 1994b), evolutionary programming (Fogel et al. 1966, Fogel 1992), and evolution strategies (Schewefel 1981, Bäck et al. 1991).

The various kinds of evolutionary algorithms differ mainly in the evolution models assumed, the evolutionary operators employed, the selection methods, and the fitness functions used (Fogel 1994). Genetic Algorithms (GAs) and Genetic Programming (GP) model evolution at the level of genetic. They emphasize the acquisition of genetic structures at the symbolic level and regularities of the solutions. On the other hand, the idea of optimization is used in Evolution Strategies (ES) and the structures being optimized are the individuals of the population. Various behavioral properties of the individuals are parametrized and their values evolved as an optimization process. Evolutionary Programming (EP) uses the highest level of abstraction by emphasizing the adaptation of behavioral properties of various species. The following sections describe the four kinds of evolutionary algorithms.

## 2.2. Genetic Algorithms (GAs)

Genetic algorithms (GAs) are general search methods that use the analogies from natural selection and evolution. These algorithms encode a potential solution to a specific problem in a simple string of alphabets called a chromosome and apply reproduction and recombination operators to these chromosomes to create new chromosomes. The applications of GAs include function optimization, problem solving, and machine learning (Goldberg 1989). The elements of a genetic algorithm are listed in table 2.1.

- an encoding mechanism for solutions to the problem,
- a population of chromosomes representing the solutions,
- a mechanism to generate the initial population of solutions,
- an evaluation function that rates the solutions in terms of their fitness values,
- a probabilistic selection mechanism that models Darwin's survival of the fittest principle,
- genetic operators that alter the composition of the offspring during reproduction, and
- parameter values such as the population size, and the probabilities of applying genetic operators that control a GA.

**Table 2.1: The elements of a genetic algorithm**

### 2.2.1. The canonical genetic algorithm

Consider a parameter optimization problem where we must optimize a set of variables either to maximize some targets such as profits, or to minimize costs or some measures of errors. The goal is to maximize or minimize some functions, say $F(X_1, X_2, ..., X_n)$, by varying the parameters. The encoding mechanism is essential in genetic algorithms because it determines the means of representing the optimization problem's variables. In the Canonical Genetic Algorithm (CGA), binary bit strings are used to represent values of various parameter variables being optimized. Thus, the variables are discretied and the range of the discretiation corresponds to some power of 2. The discretization should have enough resolution to represent the solution adequately. If the optimization problem involves real variables, the value of each real variable is first linearly mapped to an integer defined in a specified integer range encoded using a fixed number of binary bits. The binary codes of all variables are concatenated to form a binary string. This binary string is also called the genotype or the chromosome while the set of encoded parameters is called the phenotype of the individual.

The CGA for solving this kind of optimization problems is shown in table 2.2. The algorithm starts with an initial population Pop(0). Each chromosome of the population will be a binary string of length L which corresponds to the problem encoding (Holland 1992, Schaffer 1987). The initial population is usually generated randomly using a uniform distribution.

```
•  Assign 0 to generation t.
•  Initialize a population of chromosomes Pop(t).
•  Evaluate the fitness of each chromosome in the Pop(t).
•  While the termination function is not true do
      •  Select chromosomes from Pop(t) and store them into
         Pop(t') according to a scheme based on the fitness
         values.
      •  Recombine the chromosomes in Pop(t') and store the
         produced offspring into Pop(t").
      •  Perform simple mutation to the chromosomes in Pop(t")
         and store the mutated chromosomes into Pop(t+1).
      •  Evaluate the fitness of each individual in the next
         population P(t+1)
      •  Increase the generation t by 1.
•  Return an individual as the answer. Usually, the best
   individual will be returned.
```

**Table 2.2: The canonical genetic algorithm**

Each chromosome in Pop(0) is then evaluated and assigned a fitness value by a fitness function. The fitness function is sometimes called the evaluation function or the objective function. It provides a measure of performance (fitness value) of a chromosome by evaluating the set of parameters represented in the chromosome. The fitness function first decodes the parameter values encoded in the chromosome to form the phenotype of the individual. The problem-dependent phenotype is then evaluated by the fitness function to determine the fitness of the corresponding chromosome. The evaluation of a chromosome representing a set of parameters is independent of the evaluation of any other chromosome. In the CGA, relative fitness is defined as $f_i / \bar{f}$ where $f_i$ is the fitness value associated with chromosome i and $\bar{f}$ is the average fitness of all the chromosomes in the population.

Each generation of the CGA is a three stage process which starts with the current population Pop(t). Selection is applied to the current population to create an intermediate population Pop(t'). Recombination (crossover) is then applied to the Pop(t') to create another intermediate population Pop(t"). Then mutation is employed to create the next population Pop(t+1) from the intermediate population P(t"). The process starting from the current population Pop(t) to the next population Pop(t+1) establishes one generation in the execution of the genetic algorithm. This basic implementation of genetic algorithms is also referred to as a Simple Genetic Algorithm (SGA) by Goldberg (1989). For the first generation, the current population Pop(t) is also the initial population Pop(0). It produces the next population Pop(1) and the execution proceeds to the next generation. This process iterates until the termination function is satisfied. During each generation, the relative fitness values $f_i / \bar{f}$ of all chromosomes are first evaluated, and then selection is carried out.

The selection process models Darwin's survival of the fittest principle. In the CGA, a fitter chromosome reproduces a higher number of offspring and thus has a higher chance of propagating its genetic materials to the subsequent generation. In fitness proportionate selection scheme, a chromosome with a relative fitness value $f_i / \bar{f}$ is allocated $f_i / \bar{f}$ offspring. Thus a chromosome with a fitness value higher than the average is allocated more than one offspring, while a chromosome with a fitness value smaller than the average is allocated less than one offspring. The relative fitness value represents the expected number of offspring of a chromosome. Since it is impossible to produce fractional numbers of offspring, some chromosomes have to produce a higher number of offspring than their relative fitness values and some less than their relative fitness values. The current population Pop(t) can be viewed as a mapping onto a roulette wheel, where each chromosome is represented by a slice of the roulette wheel that corresponds proportionally to its relative fitness value. By repeatedly spinning the roulette wheel, chromosomes are chosen using stochastic sample with replacement to fill the intermediate population Pop(t'). The spinning process iterates until it has generated the entire Pop(t'). Thus, this selection scheme is also called the roulette wheel selection. This method generates a large sampling errors because the final number of offspring allocated to a chromosome may vary significantly from its relative fitness. The allocated number of offspring approaches the expected number only if the population size is very large.

After selection has been carried out, the construction of the intermediate population Pop(t') is completed and recombination can occur. This can be viewed as generating another intermediate population Pop(t") form Pop(t'). Crossover is applied to randomly paired chromosomes with a crossover probability denoted as $p_c$. In other words, a pair of chromosomes is first picked randomly, these chromosomes are then recombined with probability $p_c$ to produce two offspring that are inserted into the intermediate population Pop(t"). If recombination has not been performed, copies of the two picked chromosomes are inserted into Pop(t").

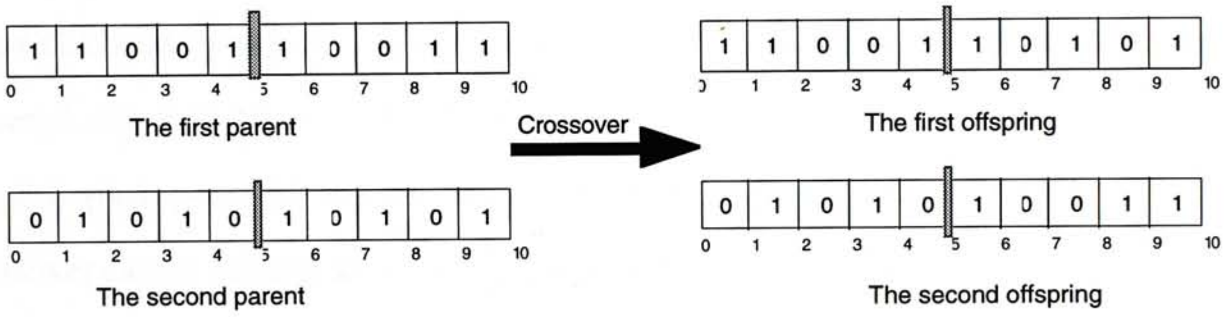


**Figure 2.1: Crossover of CGA. A one-point crossover operation is performed on two parent, 1100110011 and 0101010101, at the fifth crossover location. Two offspring, 1100110101 and 0101010011 are produced.**

Consider the two chromosomes 1100110011 and 0101010101. These chromosomes may represent possible solutions to some parameter optimization problem. For one-point crossover, a single crossover location is selected randomly. Since the length L of the chromosomes in this example is 10, a crossover location can assume values in the range between 1 to 9 (L-1). Assume the fifth location of chromosomes is chosen as the crossover location. By swapping the fragments between the two parents, the crossover operator produces the two offspring 11001:10101 and 01010:10011 where the symbol ":" is used here to denote the crossover location (figure 2.1).

After recombination is performed, other genetic operations are applied to the intermediate population Pop(t") to generate the next population Pop(t+1). In the CGA, only simple mutation can be applied. For each bit of each chromosome in the Pop(t"), it is mutated with some low probability $p_m$. There are two different implementations of mutation. The first mutation flips the bit value from 1 to 0 or vice versa, while the second one randomly selects a value from 0 and 1 to fill the mutated bit. Thus, for the latter one, there is only 0.5 probability that the bit value is really modified even if it has been selected for mutation. The mutated chromosome is then placed in the Pop(t+1). The CGA treats mutation as a secondary operator with the role of restoring lost genetic material. For example, suppose all the chromosomes in a population have converged to a 1 at a given position, and the optimal solution has a 0 at that position. In this case, crossover cannot regenerate a 0 at that position but mutation can. Figure 2.2 depicts that the chromosome 1100110101 is modified to 0100110100 by flipping the first and the last bits.

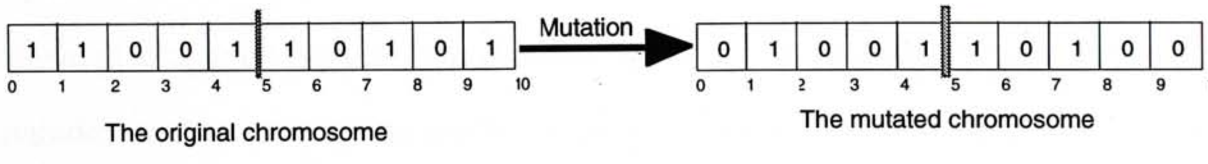


**Figure 2.2: Mutation of CGA. A mutation operation is performed on a parent 1100110101 at the first and the last bits. The offspring 0100110100 is produced.**

The about evolution process iterates until the termination criteria are satisfied. The algorithm is terminated after a fixed number of generations are attempted, the available computational resources are consumed, or satisfactory solutions are found.

GAs can be viewed as performing both exploration of new regions in the search space and exploitation of already sampled regions. The question is then on the balance between these two competing methods. The performance of GAs is significantly affected by the choice of different parameter values such as the crossover and mutation rates and the population size. The optimal choice of parameter values has been investigated extensively using empirical and analytical techniques. Grefenstette (1986), DeJong and Spears (1990) respectively propose two different sets of parameter values that are competent in general.

In addition to fitness proportionate selection, one-point crossover, and simple mutation described above, other techniques have been investigated in other genetic algorithms. The following sub-sections present these techniques.

#### 2.2.1.1. Selection methods

In fitness proportionate selection, the expected number of offspring is usually not an integer, but only integer numbers of offspring can be allocated. Thus, there is an intrinsic discrepancy between the allocated and the expected number of offspring. The remainder stochastic sampling method is proposed to achieve a distribution of offspring very close to the corresponding expected number of offspring.

In the remainder stochastic sampling method, the relative fitness value $f_i / \bar{f}$ of each chromosome i is evaluated first. If this value is greater than 1.0, the integer portion of this number indicates how many copies of that chromosome are directly placed in the intermediate population Pop(t'). All chromosomes (including those with relative fitness less then 1.0) then place additional copies of themselves in the intermediate population Pop(t') with a probability corresponding to the fractional portion of their relative fitness values. This selection method is unbiased and is efficiently implemented using a technique known as Stochastic Universal Sampling (Baker 1987).

Fitness proportionate selection has other problems. In the first few generations, the population typically has a low average fitness value, but it is common to have a few extraordinary chromosomes. Fitness proportionate selection allocates a large number of offspring to these chromosomes. These dominant chromosomes cause premature convergence. A different situation appears in the later stages when the population average fitness value is close to the best fitness value. There may be significant diversity within the population, but approximately equal numbers of offspring are allocated to all chromosomes because the variance in their fitness values is very small. Fitness scaling techniques, rank-based selection, and tournament selection can overcome these problems.

Fitness scaling techniques readjust fitness values of chromosomes (Grefenstette 1986, Goldberg 1989). Forrest (1990) presents a survey of current scaling techniques which include linear scaling, sigma truncation and power law scaling. Linear scaling computes the scaled fitness value as $f'_i = af_i + b$ where $f_i$ is the fitness value of the i$^{th}$ chromosome, $f'_i$ is the scaled value, and $a$ and $b$ are appropriate constants. In each generation, $a$ and $b$ are calculated to ensure that the maximum value of the scaled fitness values is a small number, say 1.5 or 2.0, times of the average fitness value of the population. Then the maximum number of offspring allocated to a particular chromosome is 1.5 or 2.0. Sometimes the scaled fitness values may become negative for chromosomes that have fitness values far smaller than the average fitness value of the population. In this case, $a$ and $b$ must be recomputed to avoid negative fitness values.

The sigma truncation scheme calculates the scaled fitness value as $f'_i = f_i - (\bar{f} - c\sigma)$ where $\bar{f}$ is the average fitness value of the population, $\sigma$ is the standard derivation of the fitness values in the population, and $c$ is a small constant typically ranging from 1 to 3. Chromosomes whose fitness values are less than $c$ standard deviations from the $\bar{f}$ are discarded.

The power law scaling finds some specified power of the fitness $f_i$. The scaled fitness is $f'_i = f_i^k$. The $k$ value is in general problem-dependent and may be modified during a run to stretch or shrink the range of fitness values.

The problem of fitness scaling techniques is that some parameter values ($a$, $b$, $c$, or $k$) must be determined in order to use them effectively. However, it is not trivial to decide these values. Baker (1985) proposes rank-based selection that is non-parametric. In this method, the chromosomes of a population is sorted according to their fitness values. Each chromosome is allocated the number of offspring that is a function of its rank. Usually, the number of offspring varies linearly with the rank of a chromosome. Whitley (1989) shows that significant improvements can be obtained with the selection method.

Tournament selection approximates the behavior of ranking. In a m-ary tournament, m chromosomes are selected randomly using a uniform distribution from the current population after evaluation. The best of the m chromosomes is then placed in the intermediate Pop(t'). This process is repeated until Pop(t') is filled. Goldberg and Deb (1991) show analytically that 2-ary tournament selection is the same in expectation as ranking using a linear 2.0 bias. If a winner is chosen probabilistically from a tournament of 2, then the ranking is linear and the bias is proportional to the probability with which the best chromosome is selected.

### 2.2.1.2. Recombination methods

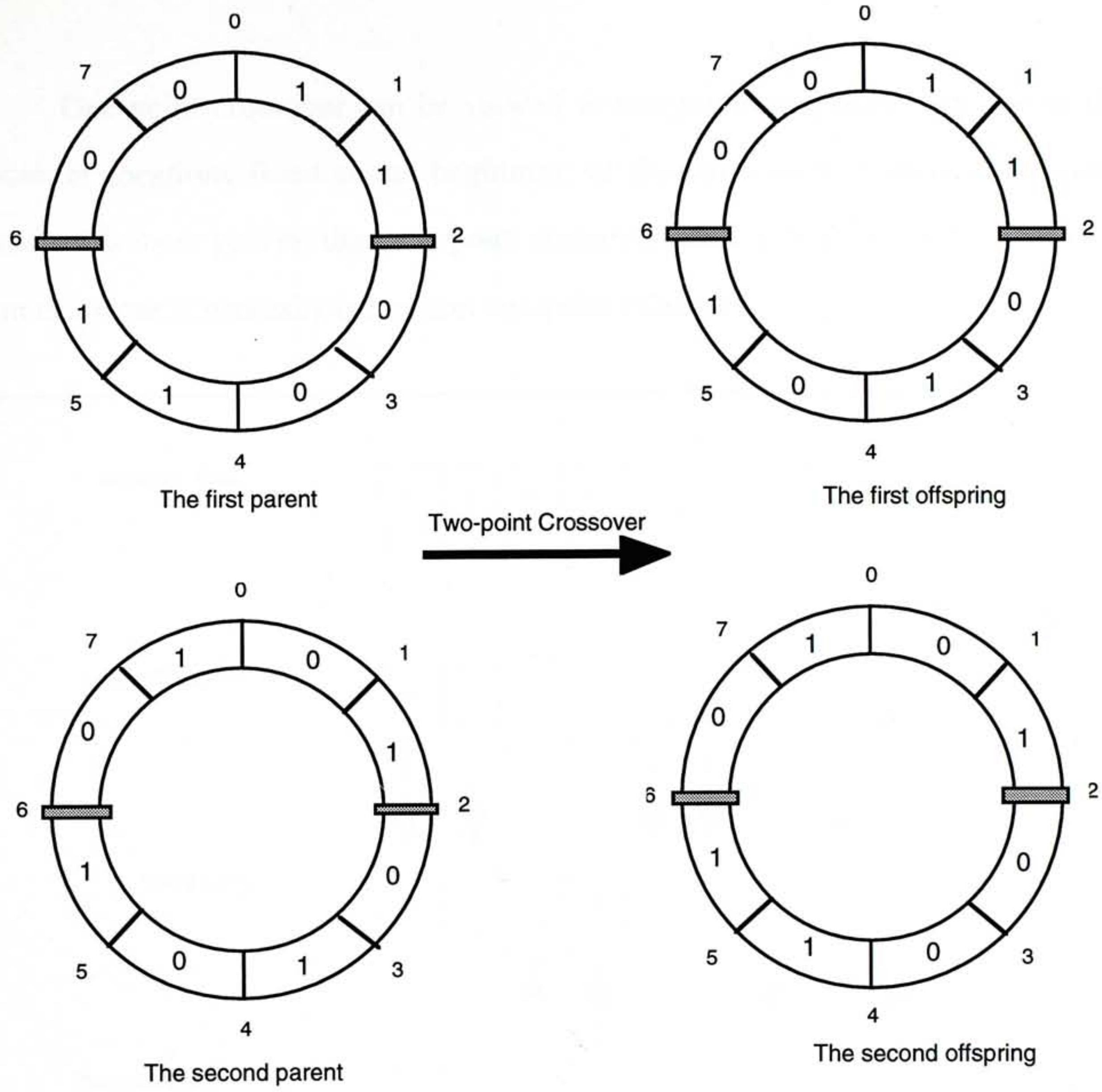


**Figure 2.3: The effects of a two-point (multi-point) crossover. A two-point crossover operation is performed on two parent, 11001100 and 01010101, between the second and the sixth locations. Two offspring, 11010100 and 01001101, are produced.**

The CGA uses one-point crossover. However, many other crossover mechanisms have been devised, often involving more than one crossover location. In two-point (multi-point) crossover, chromosomes are regarded as rings formed by joining the two ends together. To exchange a segment from one ring with that from another one requires the selection of two (multiple) crossover locations as depicted in figure 2.3.

One-point crossover can be viewed as two-point crossover with one of the crossover locations fixed at the beginning of the chromosome. Hence two-point crossover is more general than one-point crossover. Researchers now agree that two-point crossover is generally better than one-point crossover.

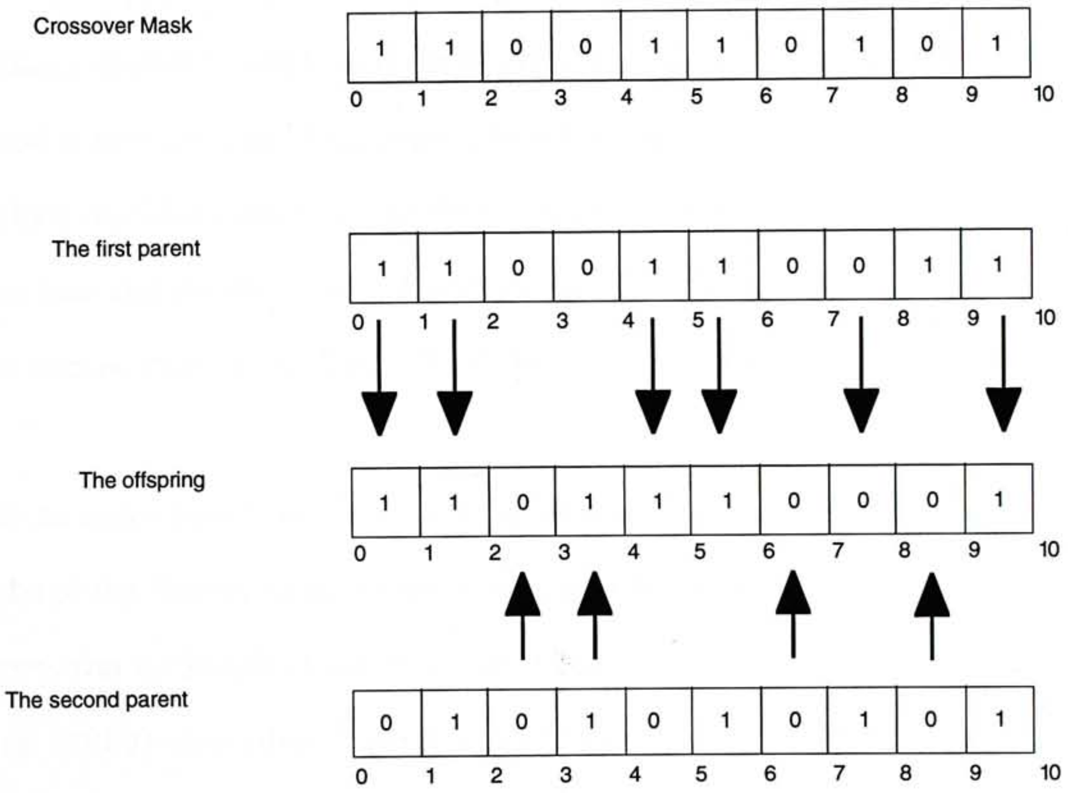


**Figure 2.4: The effects of a uniform crossover. A uniform crossover operation is performed on two parent, 1100110011 and 0101010101, and two offspring will be generated. This figure only shows one of them (1101110001).**

Uniform crossover exchanges bits of a chromosome rather than fragments. A crossover mask is first randomly generated. At each position in the offspring, the genetic material is obtained from either one of the parents. If there is a 1 in the crossover mask, the genetic material is copied from the first parent, otherwise it is obtained from the second parent. The process is repeated with the parents exchanged to produce the second offspring (figure 2.4).

An extensive comparison of different crossover methods has been performed (Eshelman et al. 1989). One-point, two-point, multi-point, and uniform crossover were theoretically analyzed in terms of positional and distributional bias, and empirically evaluated on several problems. A crossover method has positional bias if the probability that a bit is swapped depends on its position in the chromosome. The crossover method has a distributional bias if the distribution of the number of bits exchanged by the method is non-uniform. One-point crossover exhibits the maximum positional bias and the least distributional bias. At the other extreme, uniform crossover has the least positional bias and the maximum distributional bias. The empirical experiment showed that there was no more than about 20% difference in performances among the methods.

In an order-based problem, such as the traveling salesman problem, gene values are fixed and the fitness value depends on the order in which gene values appear. The above crossover techniques cannot be used because they will produce invalid offspring. Goldberg (1989) describes Partially Matched crossover (PMX) for this kind of problems. In PMX, it is the orders in which gene values appear are exchanged. Offspring have genes which inherit ordering information from each parent. This avoids the generation of offspring which violate problem constraints. Syswerda (1991b) and Davis (1991) describe other order-based operators including enhanced edge recombination, order crossover, cycle crossover, and position-based crossover. Starkweather et al. (1991) compares these operator using the traveling salesman problem and the job shop scheduling problem. They find that the effectiveness of different operators is problem-dependent.

Many other techniques have also been suggested. Several methods investigate the idea of biasing the crossover locations at some more probable chromosome positions (Schaffer and Morishma 1987, Holland 1987, Davidor 1991, Levenick 1991, Louis and Rawlins 1991). GAs learn which sites should be favored for crossover. This information is stored in a punctuation string, which is itself part of the chromosome, and so is crossed over and propagated to offspring. Thus, good punctuation strings which lead to fit offspring will be propagated through the population.

### 2.2.1.3. Inversion and Reordering

The purpose of reordering is to attempt to find gene orderings which have better evolutionary potential (Goldberg 1989). Inversion (Holland 1992) works by reversing the order of genes between two randomly selected positions in a chromosome. The operation of an inversion is illustrated in figure 2.5.

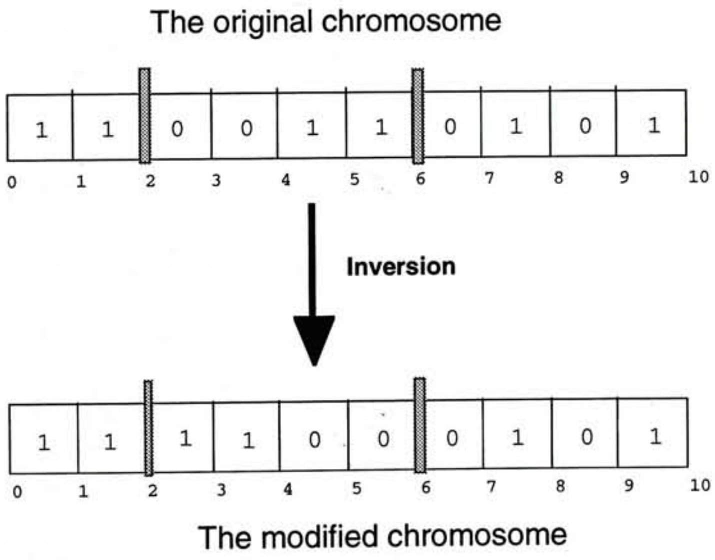


**Figure 2.5: The effects of an inversion operation. An inversion operation is performed on the parent, 1100110101, between the second and the sixth locations. An offspring, 1111000101, is produced.**

Goldberg and Bridges (1990) analyze a reordering operator on a very small task and show that it has advantages. Reordering also greatly expands the search space because GAs must also find good gene orderings. Thus, much more time is required for finding the solutions of the problem.

Meta-GAs (Grefenstette 1986) can be used to learn gene orderings. A meta-GA has a population where each member is itself a GA. Each individual GA is configured to solve the same problem, but using different gene orderings. The fitness of each individual is determined by running the GA, and examining the time required to converge. Meta-GAs are very computationally expensive to run and are worthwhile only if the results obtained can be reused many times.

### 2.2.2. Implicit parallelism and the building block hypothesis

Genetic algorithms work by implicitly sampling hyperplane partitions of a search space. This sampling process results in a robust, efficient, and complex search method (Holland 1992). Consider a problem that can be encoded in 3 bits. The search points of this problem can be represented as a cube with the string 000 at the origin (figure 2.6). The corners of this cube are numbered by binary bit strings and all adjacent corners are labeled by strings that differ by exactly one bit. The front plane of the cube contains all the points that begin with 0. If the character "*" is used to represent a wild card match symbol, then this plane can be represented by the special string 0**. Strings that contain "*" are referred to as schemata and each schema *H* represents a hyperplane in the search space. The order of a hyperplane refers to the number of actual bit values that appear in its schema. Thus, 1** is order one while 111 is order three. The order of a schema *H* is denoted as *o(H)*.

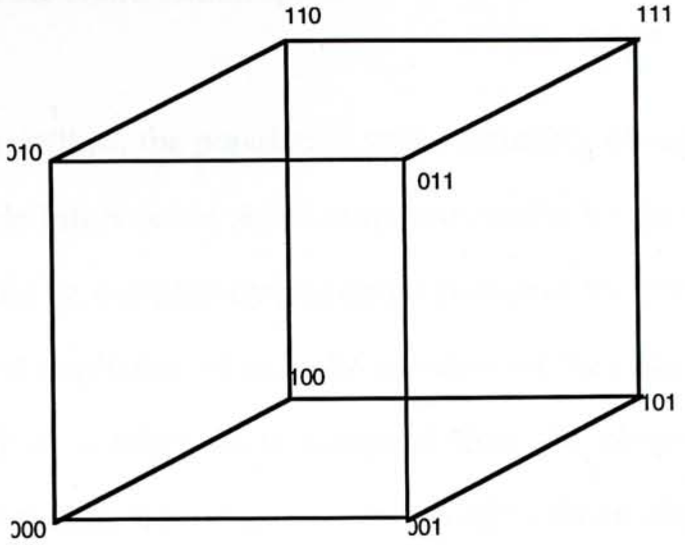


**Figure 2.6: The hyperplane space**

A chromosome or bit string matches a particular schema if that bit string can be constructed from the schema by replacing the "*" symbol with the appropriate bit value. All bit strings that match a particular schema are contained in the hyperplane represented by the schema. These bit strings are called instances of the schema. The defining bits in a schema is the bits of a schema that have values of either 0 or 1. The defining length of a schema is based on the distance between the first and the last defining bits in the schema. Let $I_l$ is the index of the position of the leftmost defining bit while $I_r$ is the index of the position of the rightmost defining bit. The defining length is $I_r$ - $I_l$. Thus, the defining length of *1**101*1* is 9 - 2 = 7. The defining length of a schema *H* is denoted by *l(H)*.

Assume that the length of a binary string is L. Since there are L positions in a particular string and each position can be either the bit value contained in the string or the "*" symbol, the number of different combinations is $2^L$. The special string of all "*" symbols represents the whole search space and is not counted as a hyperplane of the space. Thus a binary string is a member of $2^L - 1$ different hyperplanes. For each of the L positions in the scheme, the value can be either "*", 1, or 0, so there are $3^L$ different schemata. The schema of all "*" symbols is also excluded. Thus there are $3^L - 1$ different schemata in the entire search space.

In genetic algorithms, the population stores a number of sample points and these sample points provide information about numerous implicit schemata. Moreover, low order schemata should be sampled by numerous points in the population. Since many schemata are sampled implicitly when a chromosome of the population is evaluated, a much more number of schemata are sampled than the number of chromosomes contained in the population. The statistical information about any particular subset of schemata is obtained from the cumulative effects of evaluating a population of chromosomes. It results in implicit parallelism of genetic algorithms (Holland 1992). Implicit parallelism implies that many schema competitions are simultaneously solved in parallel. The theory suggests that through the process of reproduction and recombination, the competing schemata increase or decrease their instances in the population according to the relative fitness values of the chromosomes that lie in those schemata. Because genetic algorithms operate on populations of chromosomes, the number of instances of a schema can be obtained directly from the population.

The schema theorem (Holland 1992) suggests that the distribution of chromosomes in each schema should change according to the average fitness value of the chromosomes in the population that are contained in the corresponding hyperplane. Thus, even though a genetic algorithm never explicitly evaluates any particular schema, it should change the distribution of chromosomes as if it had evaluated.

Genetic algorithms process many schemata implicitly in parallel when selection acts on the population. The true average fitness value of a particular schema is the average of the fitness values of all chromosomes that lie in the hyperplane corresponds to the schema. In a given population, an estimate of the average fitness value of a schema is evaluated by averaging the fitness values of all chromosomes in the population covered by the schema. Hence, the estimate varies with the population's composition from one generation to another. According to the schema theorem, the number of copies of chromosomes that actually fall in a particular hyperplane after selection should approximate the expected number of copies that should fall in that hyperplane. Thus the estimate of the average fitness value becomes more accurate when the algorithm proceeds.

GAs can be viewed as performing simultaneous competition among schemata to increase the numbers of their instances in the population. Assume that the optimal chromosomes can be obtained by combining schemata with short defining lengths and high average fitness values. These schemata are called building blocks.

The genetic operators generate, promote, and combine building blocks to create optimal chromosomes. Crossover tends to preserve the genetic materials present in the chromosomes to be crossed. Thus, when the chromosomes to be crossed are similar, the probability of generating new building blocks reduces. Mutation is not a conservative operator and can generate radically new building blocks. Selection provides the favorable bias toward building blocks with higher fitness values and ensures that they increase their instances from generation to generation.

Crossover is the most crucial feature that differs GA from other evolutionary algorithms such as evolution strategy and evolutionary programming. GAs assume that crossover can combine good building blocks from different chromosomes to produce better offspring. This assumption is called the building block hypothesis. For some objective functions to be optimized, very bad chromosomes can be generated from good building blocks. These objective functions are referred to as deceptive functions (Goldberg 1987).

The schema theorem provides only a lower bound that holds for only a few generations into the future. Without considering what is simultaneously happening to the other schemata being processed by genetic algorithms, it is impossible to predict the number of instances of a particular schema over multiple generations. The schema theorem does not provide an accurate picture of the behavior of genetic algorithms and cannot predict how a particular schema is processed over time (Whitley 1993).

Currently, many researchers are studying the exact behavior of executable models of genetic algorithms. Goldberg (1987) analyzes the exact effects of one-point crossover on order 2 schemata using a number of equations. He also formulates the minimal deceptive problem under this framework. These equations are then generalized to handle schemata of any order (Whitley et al. 1992). These equations are based on the idea of generating gain and loss chromosomes in a systematic way. Bridges and Goldberg (1987) formalize the notion of generator for gain and loss chromosomes for one-point crossover. Vose and Liepins (1991) develop an executable model of the canonical genetic algorithm. Nix and Vose (1992) use markov chains to extend the Vose and Liepins model to include finite population models. The executable model presented by Whitley (1993) is a special case of the Vose and Liepins model. An extensive survey of different executable models is described by Vose (1993).

### 2.2.3. Steady state genetic algorithms

In a steady state genetic algorithm, two parents are selected for recombination and produce only one offspring at a time. The offspring is then placed immediately back into the population. Moreover, offspring does not replace its parents, but rather some relatively less fit member of the population. Steady state genetic algorithms have more variance than the canonical genetic algorithm with respect to the hyperplane sampling behavior. Therefore, they are more susceptible to sampling error and genetic drift. The advantage is that the best chromosomes found in the search space are maintained in the population. The search conducted by these algorithms is more aggressive and effective (Syswerda 1989; 1991a, Holland 1992).

Genitor (Whitley 1989) is an implementation of a steady state genetic algorithm. In Genitor, the worst chromosome in the population is replaced by the offspring just created. The accumulation of improved chromosomes in the population is thus monotonic. Goldberg and Deb (1991) shows that the method of replacing the worst member in the population results in a much higher selective pressure than the method of random replacement. Genitor applies rank-based selection rather than fitness proportionate selection. The advantage of rank-based selection is that it maintains a stable selective pressure over the course of search.

### 2.2.4. Hybrid algorithms

Although genetic algorithms are robust and general problem solving methods, they are usually not the most effective ones on any particular domain (Davis 1991). Therefore, combining genetic algorithms and other problem-specific strong methods may result in some general, robust, and effective problem solving systems. Many researchers use non-binary encoding and problem-specific recombination operators to strengthen the capability of traditional genetic algorithms (Davis 1991, Michalewicz 1994). Muhlenbein (1991; 1992) describes a parallel genetic algorithm that employs local hill-climbing techniques to speed up the search.

A hybrid genetic algorithm typically performs well on optimization and other search problems because it is performing local hill-climbing from multiple points in the search space. Unless the function to be optimized is severely multi-modal or the problem to be solved is highly irregular, it is likely that some points are in the basin of attraction of the global solution. In this case, hill-climbing is a fast and effective form of search. In general, the local search methods can find a small number of significant improvements of a point (chromosome) without dramatically modify its syntactic structure. Thus, a hybrid algorithm affects hyperplane sampling, but does not disrupt it completely. In this case, a hybrid algorithm takes the benefits of both the problem-specific search methods and the implicit parallelism of genetic algorithms.

## 2.3. Genetic Programming (GP)

Genetic Programming (GP) is an extension of GAs (Koza 1992; 1994). The main difference between them is the representation of the structure they manipulate and the meanings of the representation. GAs usually operate on a population of fixed-length binary strings. GP typically operate on a population of parse trees which usually represent computer programs. A parse tree is represented as a rooted, point-labeled tree. Since GP concerns with the behavior of computer programs, the definition of phenotype in GP is more abstract than that in GAs.

### 2.3.1. Introduction to the traditional GP

Most computer programs can be easily understood as performing a sequence of functions to arguments. Most language compilers first translate a given program into a parse tree and then generate a sequence of elementary assembly or machine instructions that can be executed on a target computer (Aho and Ullman 1977). Thus, parse trees are natural representations of computer programs and GP induces Lisp programs represented as parse trees.

In Lisp, a program is also called a S-expression and all operations of it are implemented as function calls. A function call consists of a list of elements enclosed by parentheses. The first element within the list is the name of the function and the other elements are arguments to the function. To represent a function call as a parse tree, the function name is the root of the parse tree while the arguments are the children at the next level down the parse tree. The arguments may be variables, constants, or other function calls. In the latter case, these function calls are again represented as parse trees and they form sub-trees of the parental parse tree. For example, the program `(* (+ X (/ Y 1.5)) (- Z 0.3))` can be represented as the parse tree in figure 2.7.

There are two sets of nodes in a parse tree. The internal nodes are called primitive functions while the leaf nodes are called terminals. In figure 2.7, the sets of primitive functions and terminals are `{+, -, *, /}` and `{X, Y, Z, 1.5, 0.3}` respectively. The terminals can be viewed as the inputs to the program being induced. They might include the independent variables and the set of constants. The primitive functions are combined with the terminals or simpler function calls to form more complex function call. The above procedure of combination iterates to produce a program. The arity of a function f, arity(f), is the number of arguments of it.

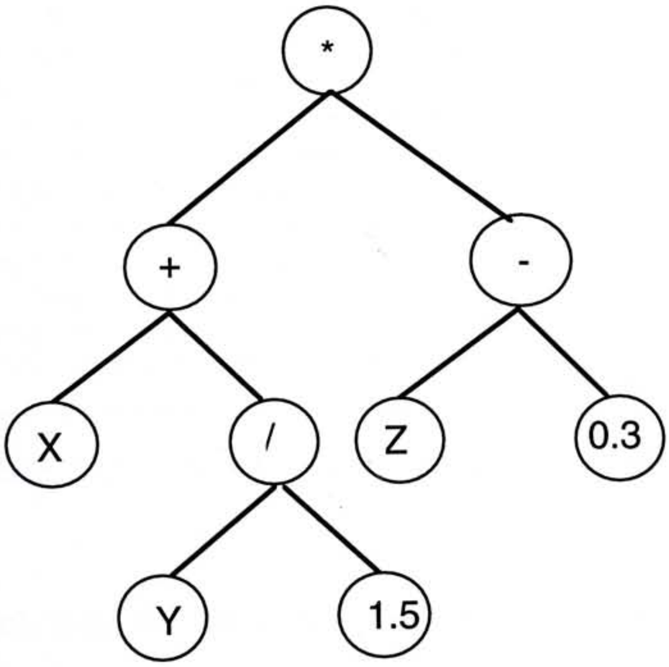


**Figure 2.7: A parse tree of the program `(* (+ X (/ Y 1.5)) (- Z 0.3))`**

The set of primitive functions might include arithmetic operators and transcendental functions. In fact, there is no limit to the complexity of the primitive functions used. Koza (1992; 1994) demonstrates iteration, functions with side-effect, and a wide variety of problem-specific functions. It is important that the function set has the closure property. That is, each primitive function should be able to accept any terminal or the output from any function as inputs. To apply GP to a problem, the user must determine:

- the set of primitive functions **F**,
- the set of terminals **T**,
- the fitness function,
- the parameters for controlling the run,
- the method for designating a result, and
- the termination function.

```
•  Assign 0 to generation t.
•  Initialize a population Pop(t) of programs composed of the
   primitive functions and terminals.
•  Evaluate the fitness of each program in the Pop(t).
•  While the termination function is not satisfied do
      •  Create a new population Pop(t+1) of programs by
         employing the selection, crossover, mutation, and
         other genetic operations.
      •  Evaluate the fitness of each individual in the next
         population P(t+1)
      •  Increase the generation t by 1.
•  Return the program that is identified by the method of
   result designation as the solution of the run
```

**Table 2.3: A high-level description of GP**

The fitness function, the controlling parameters, the method for designating a result, and the termination function are similar to those of GAs. GP usually generates an initial population of programs randomly. Programs in the population are then manipulated by various genetic operators to produce a new population of programs. These operations include crossover, mutation, permutation, editing, encapsulation, and decimation (Koza 1992). The process of proceeding from one population to the next population is called a generation. A high level description of GP is given in table 2.3.

The creation of an initial random population is a random search of the search space for computer programs. A parse tree is generated randomly by first selecting a function from **F** to be the label for the root of the tree. Whenever a point of a tree is labeled with a function f from **F**, arity(f) lines are created from that point and an element from **F** ∪ **T** is randomly selected to be the label for the endpoint of each line. If a function is selected, the above process continues recursively. Otherwise, the point becomes a leaf node of the tree and the generation process is terminated for that point. The algorithm for generating a random parse tree is shown in table 2.4.

```
Generate-tree(root?, max-depth, generation-method, F, T)
{
  if root?
     set the root of the tree to a randomly selected function from F
  else if max-depth is equal to 1 then
     set the root of the tree to a randomly selected terminal from T
  else if generation-method is "Full" then
     set the root of the tree to a randomly selected function from F
  else set the root to a randomly selected element from C = F ∪ T

  for each line go out from the root
    generate a sub-tree with the call
    Generate-tree (False, max-depth - 1, generation-method, F, T)
    and attach it to the endpoint of the line

  return he root
}

main()
{
  Generate-Tree( True, max-depth, generation-method, F, T)
}
```

**Table 2.4: An algorithm for generating a random parse tree**

The parameter *max-depth* in the above algorithm controls the maximum depth of the random tree being generated. The parameter *generation-method* can be either "Full" or "Grow" corresponding to the two different generation methods proposed by Koza (1992). For a parse tree generated by the full method, the length along any path from the root to a leaf is the same no-matter which path is taken. Parse trees generated by the grow method need not satisfy this constraint. Koza employs a method called "ramped-half-and-half" to generate an initial population. It uses the full method to generate half of the members of the population and the grow method to produce the other half. The maximum depth is varied between two and a user-specified constant MAX-INITIAL-TREE-DEPTH. This approach generates trees of different shapes and sizes.

Each program in the population is measured in terms of how well it performs in the particular problem. In GP, three measures of fitness are used. Raw fitness is the measurement of fitness that is stated in the natural terminology of the program. For example, raw fitness in a pattern recognition program can be either the number of patterns that are classified correctly or the number of misclassified patterns. Which one should be used depends on the nature of the problem.

Raw fitness is usually evaluated over a set of fitness cases. They provide a basis for evaluating the performance of a program over a number of different representative situations. For the above example, fitness cases are different patterns that are classified by a program.

The standardized fitness transforms the raw fitness so that smaller value is always a better value. Transformation can be achieved by different means. Since the standardized fitness may not lie between 0 and 1, adjustment is performed to converse it into the adjusted fitness in the desired range. The adjusted fitness is obtained by $a_i = \frac{1}{1+s_i}$ where $s_i$ is the standardized fitness of the program i and $a_i$ is the corresponding adjusted fitness. The adjusted fitness has the benefit of strengthening the selective pressure when the population converges. The same effects can be achieved by selection methods other than fitness proportionate selection such as tournament and rank-based selections. Hence, the adjusted fitness is not used for these selection methods.

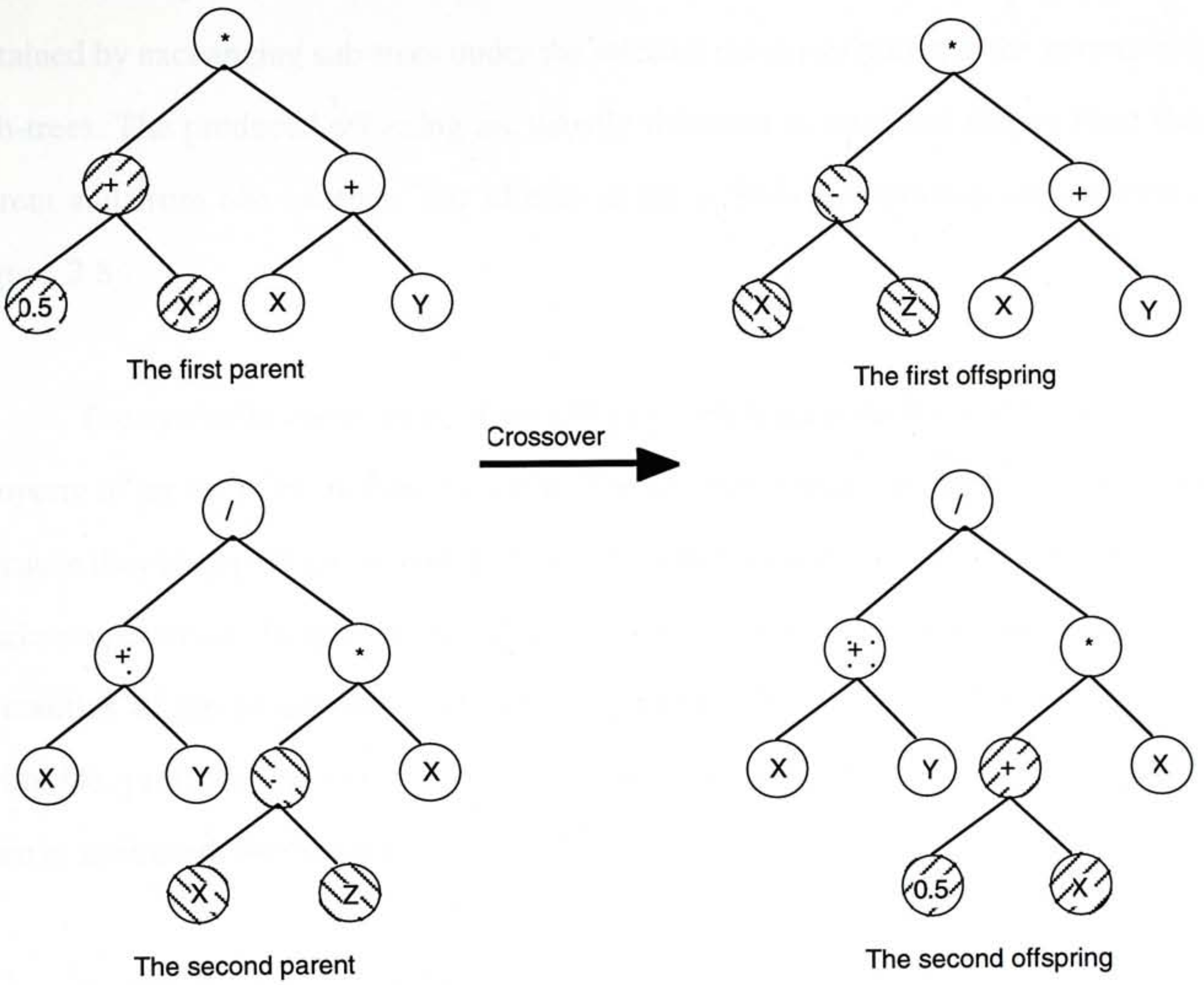


**Figure 2.8: The effects of crossover operation. A crossover operation is performed on two parental programs, `(* (+ 0.5 X) (+ X Y))` and `(/ (+ X Y) (* (- X Z) X))`. The shaded areas are exchanged and two offspring generated are: `(* (- X Z) (+ X Y))` and `(/ (+ X Y) (* (+ 0.5 X) X))`.**

The evolution process of GP is similar to that of GAs. Another key difference between them is the details of different genetic operations because these operations must now manipulate parse trees rather than fixed-length strings in GAs. Crossover of two parental trees in GP is achieved by making two duplications of the trees first to form two intermediate offspring. Then two crossover points are selected from the intermediate offspring, one within each tree. Two crossover points are required because trees are usually of different sizes and shapes from one another. The final offspring are obtained by exchanging sub-trees under the selected crossover points at the intermediate sub-trees. The produced offspring are usually different in sizes and shapes from their parent and from one another. The effects of the crossover operation are depicted in figure 2.8..

The syntactic correctness of the offspring is guaranteed because of the closure property of the set of primitives. However, the generated programs may be meaningless because they may perform semantically invalid (such as division by zero), redundant or useless operations. In order to avoid the problem of executing invalid operation, the semantics of the primitives is redefined to handle this situation. For example, the primitive, protected division %, normally returns the quotient. However, if division by zero is attempted, the function returns 1.0.

In GP, mutation is considered to be of relatively less important operation. First, a copy of a single parental tree is made. Then a mutation point is randomly selected from the copy, which will be either a leaf node or a sub-tree. The leaf node or sub-tree at the mutation point is replaced by a new leaf node or sub-tree generated randomly. The effects of the mutation operation are depicted in figure 2.9.

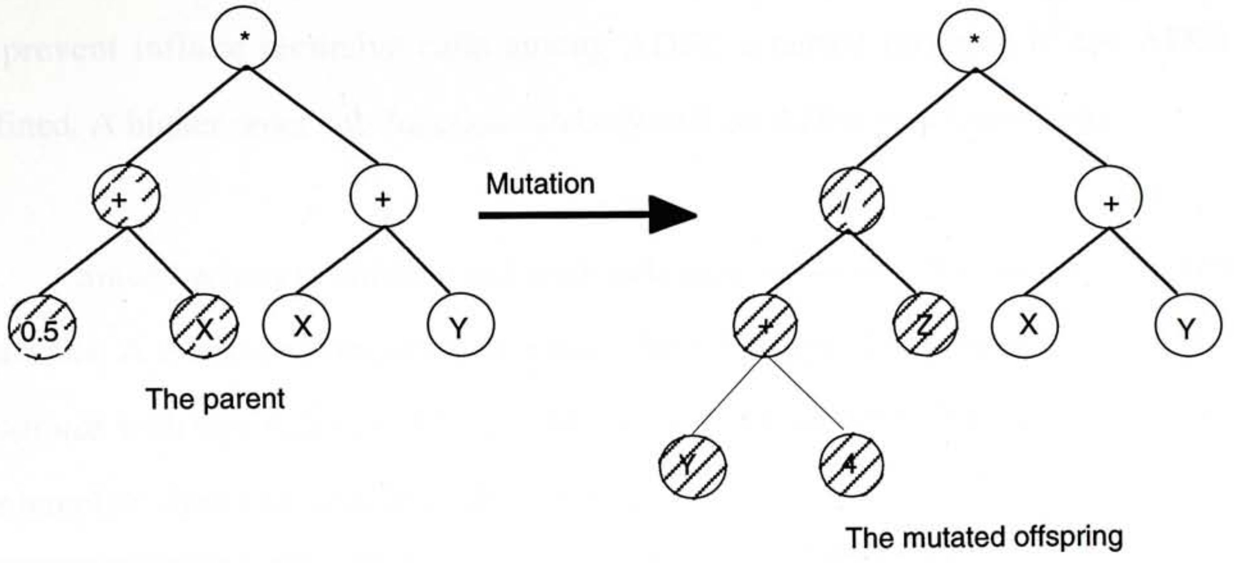


**Figure 2.9: The effects of a mutation operation. A mutation operation is performed on the program `(* (+ 0.5 X) (+ X Y))`. The shaded area of the parental program is changed to a program fragment `(/ (+ Y 4) Z)` and the offspring program `(* (/ (+ Y 4) Z) (+ X Y))` is produced.**

## 2.3.2. Automatic Defined Function (ADF)

Automatic Defined Function (Koza 1992; 1994) and module acquisition (Angeline 1993; 1994) have been proposed in GP to learn problem representation automatically. This sub-section describes Automatic Defined Function (ADF) and the next sub-section discusses Module Acquisition (MA).

Each program in the population contains multiple parts. One part, called the result-producing branch, is evaluated to produce the result of the program. Other parts are definitions of one or more sub-functions (ADFs) that may be called by the result-producing branch. These parts are called the function-defining branches. The expressions for the result-producing and the function-defining branches evolve simultaneously to find complete programs that can solve the problem. The result-producing branch can call the ADFs, and some of the ADFs can invoke others. In order to prevent infinite recursive calls among ADFs, a partial ordering of the ADFs is defined. A higher order sub-function can only call the ADFs with lower order.

Since various primitives and terminals may be used in the bodies of different branches. A template is required to restrict the evolution of programs. A template for programs with two ADFs, ADF0 and ADF1, is shown in figure 2.10. Only the parts of the template shown in bold-face are evolvable.

```
(progn
   (defun ADF0 (arg0 arg1 arg2)
      <evolvable  component  with  branch  type  1>)
   (defun ADF1 (arg0 arg1)
      <evolvable  component  with  branch  type  2>)
   (values
      <evolvable  component  with  branch  type  3>))
```

| branch type | primitive functions | terminals |
|---|---|---|
| 1 | + - * | arg0 arg1 arg2 |
| 2 | + - * ADF0 | arg0 arg1 |
| 3 | + - * ADF0 ADF1 | X Y Z |

**Figure 2.10: A template for programs with two ADFs**

The function-defining branches begin with the function symbol *defun*. In a Lisp system, the two function-defining branches will be evaluated and produce the definitions of the two sub-functions ADF0 and ADF1. The result-producing branch begins with the function symbol *values*. The expression represented in the result-producing branch will be executed and the result of the expression will be returned. If the expression invokes ADF0 and/or ADF1, the definitions of these sub-functions are applied to find the result. The function *progn* connects the two function-defining branches and the result-producing branch. It evaluates each branch in turn and returns the result obtained by executing the last branch, i.e. the result-producing branch.

The template defines different function and terminal sets that will be used in various branches of the programs. Thus ordinary crossover operator cannot be used here. In order to generate valid offspring from crossover, each branch is assigned a specific branch number called branch type and the structure-preserving crossover is used to create offspring from parents. The idea of this crossover is that any evolvable node anywhere in the whole program is randomly selected as the crossover point of the first parent. Then, the crossover point of the second parent is randomly chosen from among points of the same type. The algorithm for structure-preserving crossover is shown in table 2.5.

```
1.  Find all evolvable sub-trees of the first parental tree and store them into a global variable PRIMARY-SUB-TREES.
2.  Find all evolvable sub-trees of the second parental tree and store them into a global variable SECONDARY-SUB-TREES.
3.  If PRIMARY-SUB-TREES is not empty, select randomly a sub-tree from it using a uniform distribution. Otherwise, terminate the algorithm without generating any offspring program.
4.  Designate the sub-tree selected as SEL-PRIMARY-SUB-TREE and remove it from PRIMARY-SUB-TREES.
5.  Find a sub-tree from SECONDARY-SUB-TREES such that its type is the same as that of SEL-PRIMARY-SUB-TREE.
6.  If a sub-tree can be found in step 6, produce two offspring by exchange the two sub-trees selected. Otherwise, goto step 3.
```

**Table 2.5: Algorithm for structure-preserving crossover**

In order to use ADF, the user must determine:

- the number of function-defining branches in the overall program,
- the number of arguments possessed by each function-defining branch,
- the function and terminal sets of each function-defining branch and the result-producing branch, and
- the partial ordering of the ADFs.

The user specifies the partial ordering of the ADFs implicitly by determining the primitive function set of each function-defining branch. For example, the function set of

ADF1 of the template in figure 2.5 contains the sub-function ADF0. Thus, ADF1 can invoke ADF0. Similarly, since the function set of the result-producing branch contains the sub-functions ADF0 and ADF1, it can invoke ADF0 and ADF1. The partial ordering of the template shown in figure 2.10 is depicted in figure 2.11.

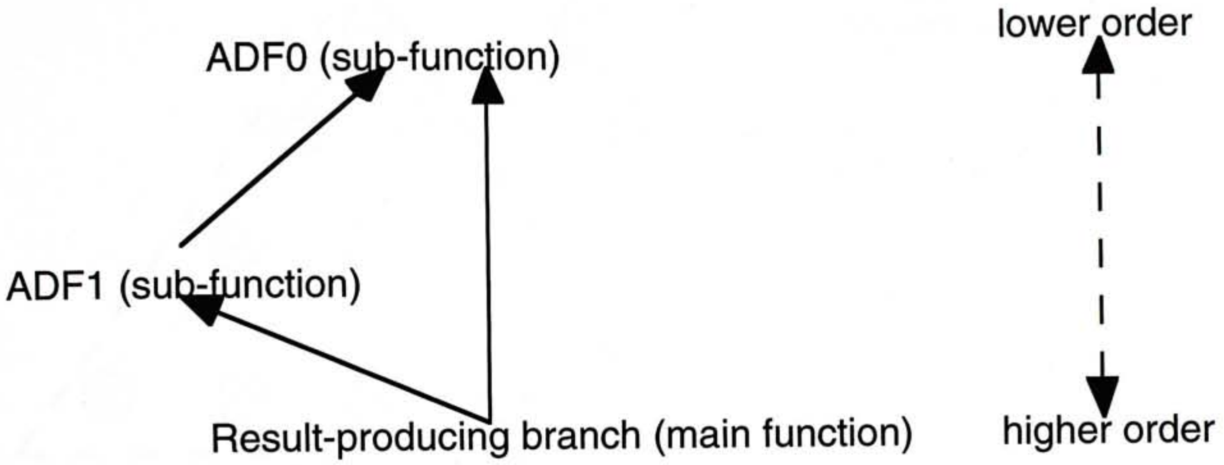
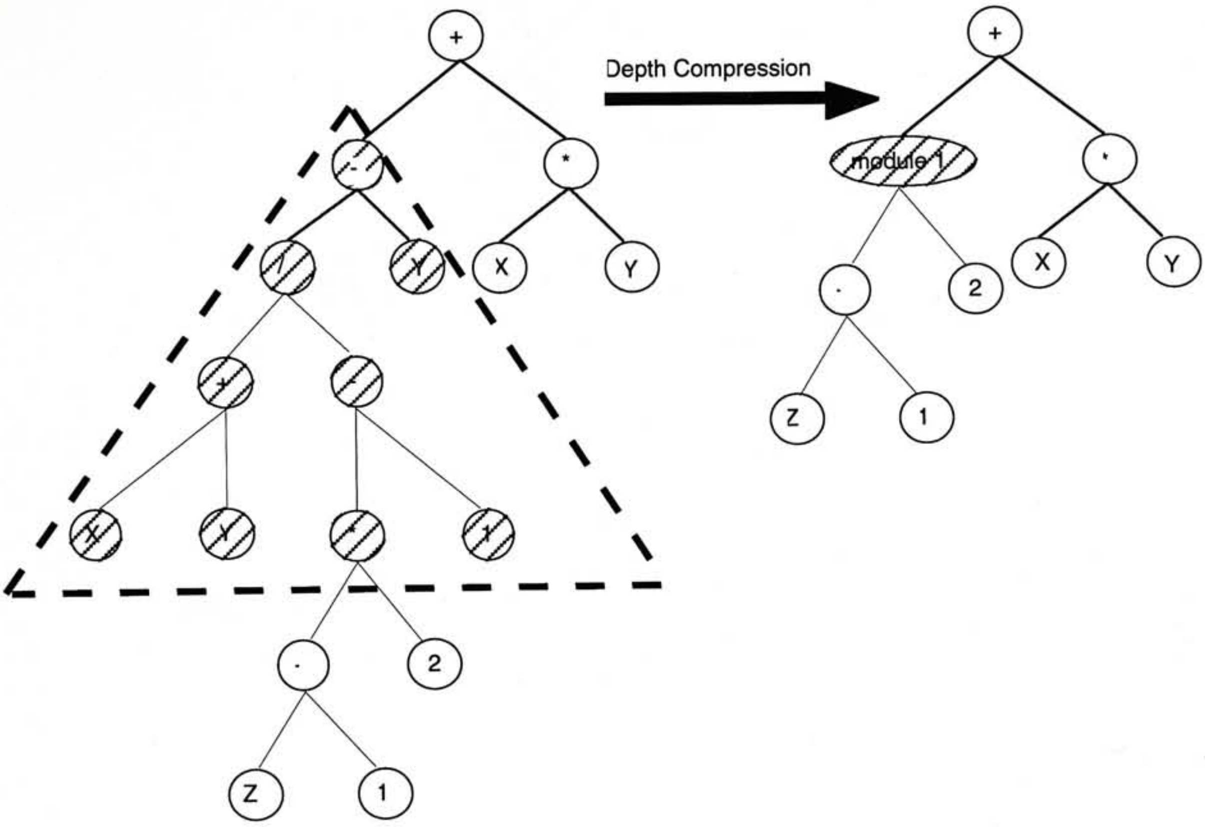


**Figure 2.11: A partial ordering of the template shown in figure 2.10**

It must be mentioned that the ADFs are local to each program. When a invocation to a particular sub-function, say ADF0, is moved by crossover from one program to another, it refers to a new ADF0 in the new program.

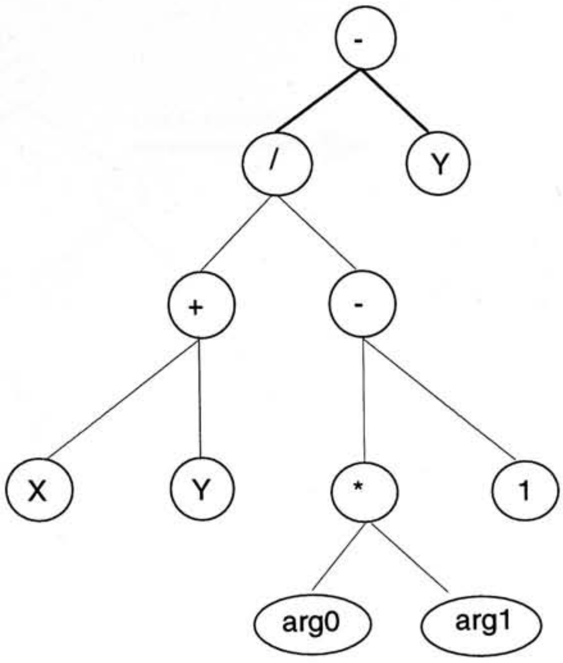
### 2.3.3. Module Acquisition (MA)

Module Acquisition (MA) is another approach of learning problem representation (Angeline 1993; 1994, Angeline and Pollack 1992; 1993). It produces a library of unique modules dynamically. These modules are globally defined and thus extend the function set of all programs. MA operates like a new genetic operator for the ordinary GP. A module is acquired by selecting a sub-tree within an existing program and defining it as a globally defined module. Two methods are proposed to extract a module from a program. In depth compression, the selected sub-tree is trimmed off a random depth to form a module. The parts of the sub-tree that are trimmed become the parameters of the module (figure 2.12).



(a)

**Figure 2.12: Module acquired by depth compression. (a) The program `(+ (- (/ (+ X Y) (- (* (- Z 1) 2) 1)) Y) (* X Y))` is compressed to `(+ (module1 (- Z 1) 2) (* X Y))`. The program fragment compressed is enclosed in dashed lines. (b) The parse tree of the module acquired by MA. (c) The corresponding lisp program of the module acquired.**



(b)

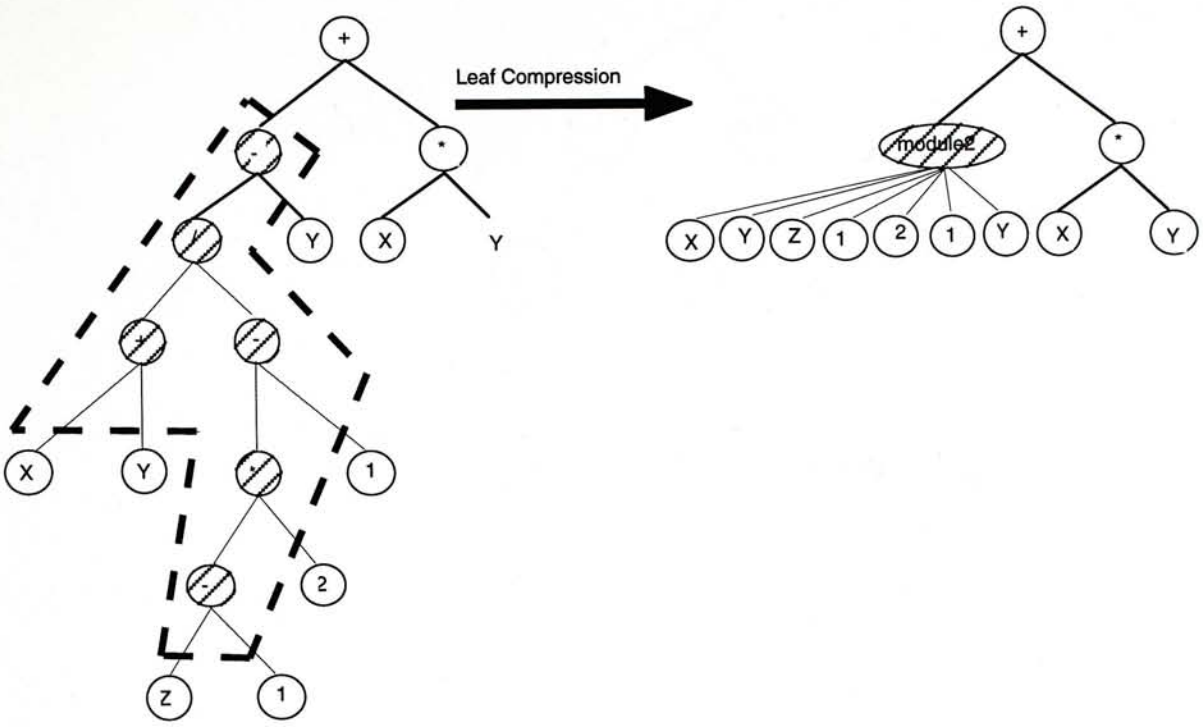
```
(defun module1 (arg0 arg1)
  (- (/ (+ X Y)
        (- (* arg0 arg1) 1))
     Y))
```

(c)

**Figure 2.12: (Cont.)**

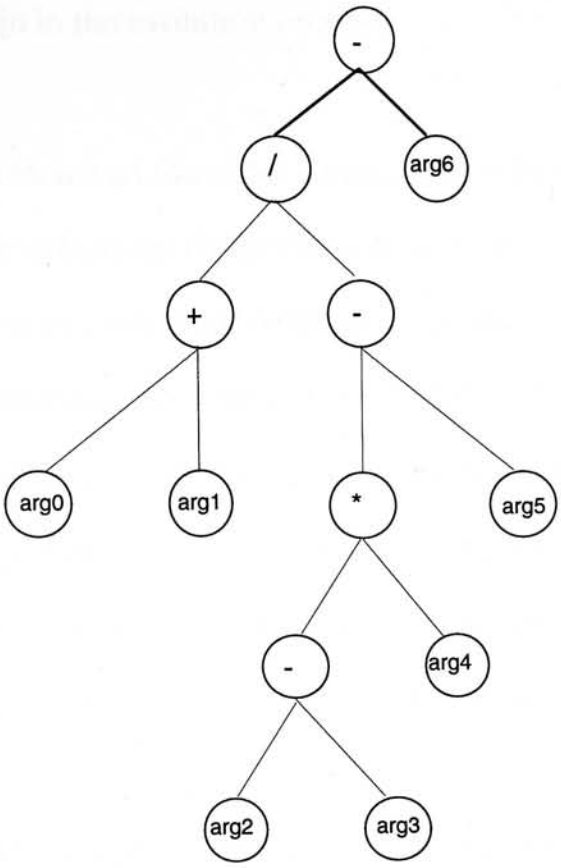
Consider the example shown in figure 2.12, the shaded sub-tree has been selected as a module. The trim depth determines the point below which the sub-trees are considered as parameters of the module. For this example, the expression `(- Z 1)` and the constant 2 are actual parameters of the module acquired. The shaded sub-tree is stored into the module library as *module1* with two formal parameters. The shaded sub-tree of the original program is replaced by an invocation of the acquired module.

Another modularization method is leaf compression. In this method, the leaf nodes of the selected sub-tree become the formal parameters of the module. The effects of if for the example in figure 2.12 are depicted in figure 2.13.



(a)

**Figure 2.13: The effects of leaf compression for the example in figure 2.12. (a) The program `(+ (- (/ (+ X Y) (- (* (- Z 1) 2) 1)) Y) (* X Y))` is compressed to `( + (module2 X Y Z 1 2 1 Y) (* X Y))`. The program fragment compressed is enclosed in dashed lines. (b) The parse tree of the module acquired by MA. (c) The corresponding Lisp program of the module acquired.**



(b)

```
(defun module2 (arg0 arg1 arg2 arg3 arg4 arg5 arg6)
  (- (/ (+ arg0 arg1)
        (- (* (- arg2 arg3) arg4) arg5))
     arg6))
```

(c)

**Figure 2.13: (Cont.)**

Modules in the library do not evolve, and are retained as long as any program applies them. Initially, there is only one reference to the module at the original program. If the module contributes good fitness to the overall program, the program would produce more offspring in the later generations and these offspring would refer to the module.

In order to modify the genetic materials of some modules, the module expansion operator takes a program and expands all the module invocations in it to create a new program with no module reference. This operation allows the genetic materials in a module to participate again in the evolution process.

Kinnear (1994) presents an intensive comparison between ADF (one template is used only) and MA. Their effects on the likelihood of evolving a correct solution to the EVEN-4-PARITY problem is contrasted. ADF has a significant improvement while MA fails to accelerate the learning. It is found that ADF creates a particular form of structural regularity that strongly increases the probability of learning a correct solution. This form of structural regularity is not present in MA. Kinnear proposes a new genetic operator based on the operators of MA. This operator, modular crossover, can produce the same kind of structural regularity for the EVEN-4-PARITY problem.

### 2.3.4. Strongly Typed Genetic Programming (STGP)

One limitation of GP is the requirement of the closure property of the set of primitive functions. In Strongly Typed Genetic Programming (STGP), all the variables, constants, arguments, and returned valued can be of any data type provided that these data types have been defined by the user (Montana 1993). One application of it is to generate a program that uses both scalars and vectors.

STGP requires that the output from each function or terminal be given a data type and that the inputs of each function take certain types. The implementation differences between GP and STGP are the generation methods of the initial population and the crossover operators. In STGP, the generation method of the initial population must comply to the type restrictions and the crossover operator must occur between functions and/or terminals of the same type.

Programs in the initial population are generated in a way such that the arguments of each function in each tree have the required data types. Crossover is implemented by randomly selecting a node from one parental tree and then randomly selecting node from the second parental tree until it is of the same type as the first node.

An extension to STGP which makes it easier to use is the concept of generic functions, which are not true strongly typed functions, but rather templates for classes of such functions. A template of a function can take a variety of different data types and return values of a variety of different types. The only constraint is that for any particular set of argument types, a generic function must return a value of a well-defined type. A generic function is instantiated to a particular instance of function by specifying a set of input argument types.

## 2.4. Evolution Strategies (ES)

In Evolution Strategies (ES), the individual model of evolution is typified (Rechenberg 1973, Schwefel 1981, Bäck et al. 1991). In these techniques, the emphasis is on the improvement of a behavior that is rated well by the fitness function rather than on the acquisition of building blocks with high fitnesses. By concentrating on optimizing the behavior, the representation and reproduction heuristics must create objects that are behaviorally similar to their parents but not necessarily structurally similar. However, the acquisition of an appropriate behavior should be easier since the effects on behavior have been modeled in the reproduction operators.

ES consider an individual to be composed of a set of traits, each of which is a feature. The interaction between the features is typically unknown. As a result, ES use fixed-length, real-valued strings to represent individuals. Each position marks a separate behavioral trait. The adherence to fixed-length strings alleviates the problem of how to manipulate the structure in order to preserve behavioral similarity between parents and their offspring. Different operators have been defined to manipulate the contents of strings to create offspring that are behaviorally similar.

```
1.   An initial population Pop(0) of m members is created.
     Each member $e_i$ is an ordered pair $(\mathbf{X}_i, \sigma_i)$ where $\mathbf{X}_i$ is a
     real-valued vector storing the object variables $x_{i,j}$ ,
     $1 \le j \le L$ for the objective function F, $\sigma_i$ is also a real-
     valued vector containing L independent strategy variables
     $\sigma_{i,j}$ , $1 \le j \le L$. The value of each object variable $x_{i,j}$ is
     selected randomly from a feasible range. The values of
     $\sigma_{i,j}$ , $1 \le j \le L$ are usually equal for all elements $e_i$,
     $1 \le i \le \mu$.
2.   Set t to 0.
3.   Create an intermediate population Pop(t') with m+1
     elements. The first m elements are obtained from Pop(t).
4.   Create a new offspring $e'_{\mu+1}$ using a recombination
     operator r on Pop(t), i.e. $e'_{\mu+1} = (\mathbf{X}'_{\mu+1}, \sigma'_{\mu+1}) = r(Pop(t))$.
5.   Create an offspring $e''_{\mu+1}$ using a mutation operator m on
     $e'_{\mu+1}$, i.e. $e''_{\mu+1} = (\mathbf{X}''_{\mu+1}, \sigma''_{\mu+1}) = m(e'_{\mu+1})$..
6.   Store $e''_{\mu+1}$ to Pop(t').
7.   Select the best m elements from Pop(t') using the
     selection operator s and store them to the new population
     Pop(t+1). Thus it contains only m elements.
8.   Increase t by 1.
9.   If the termination function is not true, goto step 3.
10.  Return an element of the last population as the result of
     the run.
```

**Table 2.6: The algorithm of ($\mu$+1)-ES**

ES originate from Germany for applications in real-valued function optimization (Rechenberg 1973, Schwefel 1981). The problem is defined as finding the real-valued vector **X** with L numbers that minimizes or maximizes an objective function F(**X**): $R^L \to R$. There are various evolution strategies that are different in their models of evolution. The one called ($\mu$+1)-ES is presented in table 2.6.

Different recombination methods have been proposed (Schewefel 1981). They can be classified into non-global and global. In the former class, two elements $e_a = (\mathbf{X}_a, \sigma_a)$ and $e_b = (\mathbf{X}_b, \sigma_b)$ are selected from the current population Pop(t) using a uniform distribution. For the simplest recombination, no actual crossover will be performed. In other words, $\mathbf{X}'_{\mu+1} = \mathbf{X}_a$ and $\sigma'_{\mu+1} = \sigma_a$.

For the discrete recombination operator, a number of uniform random values $U_j$, $1 \le j \le L$ are generated and $e'_{\mu+1}$ is obtained according to the following equation:

$$x'_{\mu+1,j} = \begin{cases} x_{a,j} & \text{if } U_j \le 0.5 \\ x_{b,j} & \text{if } U_j > 0.5 \end{cases}$$

and

$$\sigma'_{\mu+1,j} = \begin{cases} \sigma_{a,j} & \text{if } U_j \le 0.5 \\ \sigma_{b,j} & \text{if } U_j > 0.5 \end{cases}$$

where $1 \le j \le L$.

For the intermediate recombination operator, $e'_{\mu+1}$ is obtained according to the following equation:

$$x'_{\mu+1,j} = \frac{1}{2}\left(x_{a,j} + x_{b,j}\right)$$

and

$$\sigma'_{\mu+1,j} = \frac{1}{2}\left(\sigma_{a,j} + \sigma_{b,j}\right)$$

where $1 \le j \le L$.

In the global recombinations, L pairs of elements $(e_{a_j}, e_{b_j})$, $1 \le j \le L$ are selected randomly using a uniform distribution. For the global discrete recombination operator, a number of uniform random value $U_j$, $1 \le j \le L$ are created and $e'_{\mu+1}$ is obtained according to the following equations.

$$x'_{\mu+1,j} = \begin{cases} x_{a_j,j} & \text{if } U_j \le 0.5 \\ x_{b_j,\,j} & \text{if } U_j > 0.5 \end{cases}$$

and

$$\sigma'_{\mu+1,j} = \begin{cases} \sigma_{a_j,j} & \text{if } U_j \le 0.5 \\ \sigma_{b_j,j} & \text{if } U_j > 0.5 \end{cases}$$

where $1 \le j \le L$.

For the global intermediate recombination, $e'_{\mu+1}$ is obtained according to the following equations:

$$x'_{\mu+1,j} = \frac{1}{2}\left(x_{a_j,j} + x_{b_j,j}\right)$$

and

$$\sigma'_{\mu+1,j} = \frac{1}{2}\left(\sigma_{a_j,j} + \sigma_{b_j,j}\right)$$

where $1 \le j \le L$.

The mating parents for the global recombination of component $x'_{\mu+1,j}$ and $\sigma'_{\mu+1,j}$ are chosen anew from the population. Thus, it causes a high mixing of the genetic materials of the whole population. Global recombinations address the difficulty of premature convergence in ES systems.

According to the biological observation that offspring are similar to their parents and that smaller modifications occur more often than larger ones. To achieve the similar effects in ES, the element $e''_{\mu+1}$ obtained by applying mutation operation on element $e'_{\mu+1}$ is specified as:

$$x''_{\mu+1,j} = x'_{\mu+1,j} + N(0, \sigma'_{\mu+1,j}) \quad ,1 \le j \le L$$

and

$$\sigma''_{\mu+1,j} = \begin{cases} c_d \sigma'_{\mu+1,j} & \text{if } r < \frac{1}{5} \\ c_i \sigma'_{\mu+1,j} & \text{if } r > \frac{1}{5} \\ \sigma'_{\mu+1,j} & \text{if } r = \frac{1}{5} \end{cases} \quad ,1 \le j \le L$$

where N(0, σ) is a Gaussian random number with mean of zero and standard deviation σ, $c_d$ and $c_i$ are constants, and r is the ratio of successful mutations to all mutation. A mutation is successful if the mutated offspring performs better than its parent. The idea here is to change the strategy variables dynamically until r is 1/5.

Rechenberg (1973) calculated the convergence rate of a ES system for some model functions and found that the convergence rate is optimized if r is equal to 1/5. Thus, he suggested the 1/5 rule: The ratio of successful mutations to all mutation should be 1/5. If it is greater than 1/5 then increase σ by multiplying a constant $c_i$, if it is less than 1/5 then decrease $\sigma$ by multiplying a constant $c_d$. When this rule decreases the standard deviation, the search becomes more focused, with generated offspring being generally closer to their parents in value. When the standard deviation is increased, the search is broadened so that offspring might be further from their parents. Schewefel (1981) suggests that $c_d$ and $c_i$ should be 0.82 and 1/0.82 respectively.

The selection operator chooses the best $\mu$ elements from $\mu+1$ elements according to the objective function F. The termination function determines whether the optimization has been found or the computational resources are all consumed. Different methods can be used to implement the termination function and these methods are usually domain-dependent.

The simplest and oldest ES model is denoted as (1+1)-ES. The difference between it and ($\mu$+1)-ES is that the population Pop(t) contains only one element and only recombination will be performed. It can be designated as a kind of probabilistic gradient search technique. There are two main drawbacks of (1+1)-ES: The convergence rate is slow because the standard deviations are equal in each dimension; The procedure is susceptible to stagnation at local minima because of the brittleness of the gradient search.

In the ($\mu+\lambda$)-ES, the population size is still $\mu$, but $\lambda$ offspring are created at each generation from $\mu$ parents. All $\mu+\lambda$ elements compete for survival, with the best $\mu$ elements selected to survive in the next generation. Consequently, step 3 in table 2.6 is changed to:

3'. Create an intermediate population Pop(t') with $\mu+\lambda$ elements. The first $\mu$ elements are obtained from Pop(t).

In the (μ, λ)-ES, only the λ offspring compete for survival, and the μ parents are replaced each generation. The lifetime of every element is limited to a single generation. Thus, step 3 in table 2.6 is changed to:

3". Create an intermediate population Pop(t') with λ elements.

Because of the nature of this model, λ must be greater than or equal to μ. In the (μ+λ)-ES and (μ, λ)-ES, steps 4 through 6 in table 2.6 are repeated for λ times to create λ offspring. The mutation operator is also extended to allow for meta-control over the evolution process. Let $e'_{\mu+1} = (\mathbf{X}'_{\mu+1}, \sigma'_{\mu+1})$ be the offspring generated by the recombination operator. The mutation operator creates the offspring $e''_{\mu+1} = (\mathbf{X}''_{\mu+1}, \sigma''_{\mu+1})$ according to the following equations:

$$\sigma''_{\mu+1,j} = \sigma'_{\mu+1,j} \exp^{N(0,\Delta\sigma)} \qquad ,1 \le j \le L$$

and

$$x''_{\mu+1,j} = x'_{\mu+1,j} + N(0,\sigma''_{\mu+1,j}) \quad ,1 \le j \le L$$

where $\Delta\sigma$ is a meta-control parameter. It allows the user to have more control over the distribution of trials. It should be emphasized that in all models other than (1+1)-ES, more than one parent are participated in the recombination. Since the strategy variables $\sigma_{i,j}$, $1 \le j \le L$ are all stored in each element $e_i$ , $1 \le i \le \mu$, these strategy variables are also involved in the recombination and evolution. These models allow strategy variables to adapt to the landscape of the objective function and thus trials can be distributed in a more appropriate way.

## 2.5. Evolutionary Programming (EP)

Evolutionary Programming (EP) is a stochastic optimization strategy similar to GAs (Fogel et al. 1966, Fogel 1994). It emphasizes the behavioral linkage between parents and their offspring rather than seeking to emulate specific genetic operators as observed in nature. It is a useful method of optimization when other techniques such as gradient descent or direct, analytical methods are not possible. EP is suitable for difficult combinatoric and real-valued function optimization problems in which the fitness landscapes are rugged and have many locally optimal solutions.

```
• Set t to 0.
• Create an initial population of trials Pop(t) randomly.
• Each trial in the population Pop(t) is assessed by computing
  its fitness.
• While a threshold for iteration is not exceeded and
        a satisfying solution has not been found do
     • Each solution in Pop(t) can produce one or more
       offspring. Each of these offspring is mutated
       according to a distribution of mutation types, ranging
       from minor to extreme with a continuum of mutation
       types in between. The severity of mutation is judged
       on the basis of the functional change imposed on the
       parent. The mutated offspring are stored in the
       intermediate population Pop(t')
     • A stochastic tournament is usually held to determine N
       solutions to be retained for the Pop(t+1) of
       solutions. Occasionally,  selection is performed
       deterministically.  There is no requirement that the
       population size be held constant,
     • Each trial in the population Pop(t+1) is assessed by
       computing its fitness.
     • Increase t by 1.
• Return an element of the last population as the result of
  the run. Usually the best one is returned.
```

**Table 2.7: A high-level description of EP**

EP employs a model of evolution at a higher abstraction than GAs, GP, and ES. It models the reproductive relationship between species behavior in successive generations (Fogel 1994). The reproductive operators used in EP are a form of mutations that attempt to preserve behavioral similarity between offspring and their parents (Fogel 1992). The motivation for behavioral similarity is taken directly from biology where an offspring is generally similar to its parent at the behavioral level with only slight variations. These variations can be modeled by assuming that the distribution of potential offspring resembles a normal distribution around the parent's behavior in the fitness landscape. On the other hand, GAs cannot guarantee such a distribution because it emphasizes on structural similarity.

For EP, there is an underlying assumption that a fitness landscape can be characterized in terms of variables, and that there is an optimum solution in terms of these variables. A high-level description of EP is depicted in table 2.7.

There are two important differences between EP and GAs. Firstly, there is no constraint on the representation. The CGA involves encoding the problem solutions as fixed-length binary strings. In EP, the representation follows from the problem. For example, a neural network can be represented in the same manner as it is implemented. Thus the mutation operation does not demand and assume any particular encoding method.

Secondly, the mutation operators simply change aspects of the parent according to a statistical distribution. Minor modifications in the behavior of the offspring occur more frequently than substantial variations in the behavior of the offspring. Furthermore, the severity of mutations is often reduced as the global optimum is approached. In the Meta-Evolutionary technique, the variance of the mutation distribution is subject to modification by a fixed variance mutation operator and evolves along with the solutions (Fogel 1994).

EP and ES share many similarities. Typically, they operate on the real-valued representations when solving real-valued function optimization problems  Multivariate zero mean Gaussian mutations are applied to each parent in a population and a selection mechanism is applied to determine which solutions are maintained. Both approaches encode information on how to distribute new trials in the elements of the population and allow evolution to adapt this information. Most of the theoretical results on asymptotic convergence and convergence rate developed for ES or EP also apply directly to the other.

There are two main differences between ES and EP. Firstly, EP typically uses stochastic tournament selection while ES typically uses deterministic selection in which the worst solutions are eliminated from the population based directly on their fitness values. Secondly, EP is an abstraction of evolution at the level of reproductive populations (i.e., species) and thus recombination mechanisms are generally inapplicable because recombination does not occur between species. In contrast, ES is an abstraction of evolution at the level of individual behavior. Since genetic information can be encoded in an individual to affect its behavior, recombination is reasonable. Different recombination operators of ES have been discussed in the previous section. The effectiveness of such recombination operators depends on the problem at hand.

# Chapter 3
# Inductive Logic Programming

In the previous chapter, we present an overview on evolutionary algorithms. Another approach of program induction is Inductive Logic Programming (ILP) that investigates the construction of logic programs from training examples and background knowledge. ILP is a new research field that combines the techniques and theories from inductive concept learning and logic programming. ILP systems are more powerful than traditional attribute-value based learning systems because the formers use an expressive first-order logic framework to represent the concepts acquired and employ background knowledge to facilitate the learning. ILP has strong theoretical foundation from computational learning theory and logic programming. It has very impressive applications in scientific discovery, knowledge acquisition and logic program synthesis (Muggletion 1994, Bratko and King 1994). In this chapter, we will present a brief introduction to inductive concept learning first. Two approaches for ILP are discussed in the second section followed by an introduction to techniques and methods of ILP.

## 3.1. Inductive concept learning

The goal of machine learning is to develop techniques and tools for building intelligent learning machines. In other words, learning machines can improve themselves to perform more efficiently and/or more accurately. They can also increase their abilities to process more problems. Symbol-level learning is used to characterize the kind of learning that increases the efficiency of the system while knowledge-level learning improves the accuracy and/or coverage of the system (Dietterich 1986). Machine learning paradigms include inductive, deductive, genetic-based and connectionist learning (Michalski et al. 1983; 1986b, Kodratoff and Michalski 1990, Shavlik and Dietterich 1990, Carbonell 1989). Multistrategy learning integrates several learning paradigms (Michalski and Tecuci 1994). This chapter focuses on supervised, inductive learning of a single concept. If **U** is a universal set of observations, a concept **C** is formalized as a subset of observations in **U**. Inductive concept learning finds descriptions for various target concepts from positive and negative training instances of these concepts. In single concept learning, a target concept description is induced from training instances labeled positive and negative. In multiple concept learning, more than one target concept are being learned simultaneously, training examples are labeled by various concept names representing their categories.

In machine learning, formal languages for describing observations and concepts are called object and concept description languages respectively. Typically, object description languages are attribute-value pair descriptions and first-order languages of Horn clauses. Concepts can be described extensionally or intensionally. A concept is described extensionally by listing the descriptions of all of its instances (observations). Thus extensional concepts are represented in the object description language. On the other hand, intensional concepts are expressed in a separate concept description language that permits compact and concise concept descriptions. Typical concept description languages are decision trees, decision lists, production rules, and first-order logic.

Inductive concept learning can be viewed as searching the space of hypothesis descriptions. A bias is a mechanism employed by a learning system to constrain the search for target hypotheses. A search bias determines how to conduct the search in the hypothesis space while a language bias determines the size and structure of the hypothesis space.

A strong search bias, such as the hill-climbing search strategy, employs existing knowledge about the size and structure of the hypothesis space to exploit promising solutions of the space, thus it can find the target concept quickly. But it may trap the system in a local maximum. A weak search bias, such as depth-first and breath-first search, explores the space completely; the learner is guaranteed to find the target concept that can be represented by the concept description language. Nevertheless, a weak bias is very inefficient. In other words, the search bias introduces the efficiency/completeness tradeoff into a learning system.

A strong language bias defines a less expressive description language such as the propositional logic. The hypothesis space created by the bias is comparatively smaller and the learning can be performed more efficiently. But the learner may fail to find the target concept which is not contained in the small hypothesis space. A weak bias defines a larger space and thus the target concept is more likely to be expressible in the space. The disadvantage is that the learner is less efficient. The language bias introduces the efficiency/expressiveness tradeoff into a learning system.

Given:

- -A set **E** of positive $\mathbf{E}^+$ and negative $\mathbf{E}^-$ examples of a concept **C**.
- -Concept description language **L**.
- -Search and language bias.
- -Background knowledge **B**.

Find:

A complete and consistent hypothesis **H** represented in the language **L**.

A hypothesis **H** is complete if every positive example $e \in \mathbf{E}^+$ is covered by it with respect to(w.r.t.) **B**.

A hypothesis H is consistent if no negative example e $\in \mathbf{E}^-$ is covered by it w.r.t. **B**.

**Table 3.1: Supervised inductive learning of a single concept**

Background knowledge **B** is a prior knowledge that can be used by either the search bias to direct the search more efficient, or the language bias to express the hypothesis space in a more natural and concise way. If a learning system is not provided with some a prior knowledge about the learning problem, it must learn exclusively from training examples. However, difficult learning problems typically require a lot of knowledge. The task of supervised inductive learning of a single concept **C** is formulated in table 3.1.

## 3.2. Inductive Logic Programming (ILP)

Relational concept learning induces a new relation for the target concept (i.e., the target predicate) from training examples and known relations from the background knowledge. An ILP system is a relational concept learner. The training examples, the hypothesis space, and the background knowledge are represented in first-order Horn clause languages (Muggleton and Feng 1990). Tradeoffs between expressiveness and efficiency are introduced by some additional restrictions on these languages. This section describes two approaches of ILP, interactive and empirical ILP. Muggletion and De Raedt (1994) present a comprehensive introduction of theory and methods of ILP. Before presenting these approaches, the terminology of logic programming is described first (Lloyd 1987).

The alphabet of a first-order language contains predicate symbols, function symbols and variables. A predicate symbol is a lower case letter followed by a string of lower case letters and/or digits. A function symbol is a lower case letter followed by a string of lower case letters and/or digits. A variable is an upper case letter followed by a string of lower case letters and/or digits.

A term is a variable or a function. A function is a function symbol immediately followed by a sequence of terms enclosed in a pair of parentheses. The number of terms in the sequence is the arity of the function. For example, `f(g, h(X, Y), X)` is a function of arity 3 where `f`, `g`, and `h` are function symbols, `X` and `Y` are variables. A constant is a function of arity 0. Thus `g` is a constant.

An atomic formula, or atom, is a predicate symbol immediately followed by a sequence of terms enclosed in a pair of parentheses. The number of terms in the sequence is the arity of the atomic formula. For example, `mother(X, Y)` is an atom of arity 2 where `mother` is a predicate symbol and `X` and `Y` are variables.

A literal can be classified as either a positive literal or a negative literal. A positive literal L is an atomic formula while a negative literal $\neg L$ is the symbol $\neg$ followed by an atomic formula. A clause is a formula of the form $\forall X_1, X_2, \ldots, X_m(L_1 \vee L_2 \vee \ldots \vee L_n)$ where $L_i$, $1 \leq i \leq n$ are literals and $X_1, X_2, \ldots, X_m$ are variables occurring in the clause. A clause $\forall X_1, X_2, \ldots, X_m(L_1 \vee L_2 \vee \ldots \vee L_i \vee \neg L_{i+1} \vee \neg L_{i+2} \vee \ldots \vee \neg L_n)$ can be represented as $L_1 \vee L_2 \vee \ldots \vee L_i \leftarrow L_{i+1} \wedge L_{i+2} \wedge \ldots \wedge L_n$. The previous clause can be written as $L_1, L_2, \ldots, L_i \leftarrow L_{i+1}, L_{i+2}, \ldots, L_n$ where commas on the left-hand side of $\leftarrow$ denote disjunctions while commas on the right-hand side represent conjunctions.

A definite program is a set of definite program clause. A definite program clause, $\forall X_1, X_2, \ldots, X_m(T \vee \neg L_1 \vee \neg L_2 \vee \ldots \vee \neg L_n)$, is a clause which contains exactly one positive literal. It can be represented as the form $T \leftarrow L_1, L_2, \ldots, L_n$ where T and $L_i$, $1 \leq i \leq n$ are atomic formulae. The positive literal T in a definite program clause is called the head or goal of the clause. The sequence of literals $L_i$, $1 \leq i \leq n$ is called the body of the clause. A Horn clause is a clause which contains at most one positive literal. Thus a Horn clause can be either a definite program clause or a definite goal: a clause with no positive literal. A definite goal can be represented as the form $\leftarrow L_1, L_2, \ldots, L_n$ where $L_i$, $1 \leq i \leq n$ are atomic formulae. A positive unit clause is a definite program clause with an empty body. It is called a fact in Prolog and is denoted simply as T.

A normal program is a set of program clauses. A program clause is a clause of the form $T \leftarrow L_1, L_2, \ldots, L_n$ where T is an atom and $L_i$, $1 \leq i \leq n$ are positive or negative literals. In the programming language Prolog, literals of the form `not L` are allowed in the body of a clause, where `L` is an atom and `not` is interpreted under the negation-as-failure rule (Clark 1978).

A predicate definition is a set of program clauses with the same predicate symbol (and arity) in their heads. A set of clauses is called a theory and represents the conjunction of the clauses. A well-formed formula is a literal, a clause and a theory. A well-formed formula or term is ground if and only if there is no variable in the formula or term.

### 3.2.1. Interactive ILP

Interactive ILP is often used in incremental and interactive theory revision (De Raedt 1992). An Interactive ILP system is provided with six inputs: 1) a set of correct examples **E** that has been examined before, 2) correct background knowledge **B**, 3) an incorrect theory **T**, 4) a concept description language **L**, 5) a new positive or negative training example e, and 6) a teacher that can answer questions generated by the system. The system modifies the definition of **T** and creates a new theory **T'** such that it is complete and consistent with respect to all examples seen (i.e. $\mathbf{E} \cup \{e\}$) and the background knowledge **B**.

Shapiro (1983) introduced the idea of refinement operators in the MIS system which is used to structure the search space of program clauses. The system searches the space in a breadth-first top-down manner. CLINT (De Raedt 1992, De Raedt and Bruynooghe 1989; 1992) generates its own learning examples and asks questions about their classifications. It is featured with the applications of integrity constraints and its ability in changing concept description language dynamically.

Most interactive ILP systems are based on special forms of the general theory of inverse resolution introduced in CIGOL (Muggleton and Buntine 1988, Muggleton 1992). The three operators of CIGOL are absorption, intraconstruction and truncation. Absorption generalizes program clauses, intraconstruction learns definitions of new predicates and truncation generalizes unit clauses. The concept of absorption is first introduced by Sammut and Baneji (1986) in their MARVIN system. Wirth (1989) suggests two operators which are similar to absorption and intraconstruction. Rouveirol (1991; 1992) introduces a saturation procedure which overcomes some problems of absorption and truncation.

### 3.2.2. Empirical ILP

The task of empirical ILP is usually concerned with learning a single target concept from a given set of training examples and background knowledge. The task of empirical ILP is formulated in table 3.2.

Given:

-A set **E** of positive $\mathbf{E}^+$ and negative $\mathbf{E}^-$ training examples of the target predicate p. Training examples are represented as ground atoms

-A concept description language **L**

-Search and language bias.

-Background knowledge **B**

Find:

A definition **H** for the target predicate p expressible in **L** such that **H** is complete and consistent with respect to (w.r.t.) the training examples **E** and the background knowledge **B**

**H** is complete if every positive example $e^+$ in $\mathbf{E}^+$ is covered by **H** w.r.t. the background knowledge **B**. i.e.
$\mathbf{B} \cup \mathbf{H} \models e^+$

**H** is consistent if no negative example $e^-$ in $\mathbf{E}^-$ is covered by **H** w.r.t. the background knowledge **B**. i.e.
$\mathbf{B} \cup \mathbf{H} \not\models e^-$

**Table 3.2: Definition of Empirical ILP**

The background knowledge **B** provides definitions of known predicates $q_i$ which can be used in the definition of the target predicate p. It also provides additional information to ease the search of the definition of p. This information includes argument types, symmetry of predicates in pairs of arguments, input/output modes, rule models, predicate sets, parametrized languages, integrity constraints, determinations and any knowledge that can modify the operation of the search and language bias.

In the definition, a training example is covered by **H** given background knowledge **B** if e is a logical consequence of $\mathbf{B} \cup \mathbf{H}$. This notion of coverage is called intensional coverage (Lavrac and Dzeroski 1994). It allows the background knowledge **B** to include normal clauses and ground facts. For a particular concept description language **L**, an appropriate proof procedure must be used to check whether an example is entailed by $\mathbf{B} \cup \mathbf{H}$. The SLD-resolution proof procedure with bounded or unbounded depth is usually employed to determine whether a training example is entailed (Lloyd 1987). In depth-bounded SLD-resolution, unresolved goals in the SLD-proof tree at depth h are not expanded and are treated as failed. MIS (Shapiro 1983) and CIGOL (Muggleton and Buntine 1988) use this proof procedure to prevent infinite loops.

On the other hand, extensional coverage can also be used. In this case, extensional background knowledge **B** containing only ground facts must be employed to determine whether an example e is covered (Shapiro 1983). A hypothesis **H** extensionally covers an example e with respect to an extensional background knowledge **B** if there exists a clause $T \leftarrow L_1, L_2, \ldots, L_n$ in **H** and a substitution $\theta$ such that $T\theta = e$ and $\{L_1, L_2, \ldots, L_n\}\theta \subseteq \mathbf{B}$. If the background knowledge **B** provided by the users contains non-ground clauses, the empirical ILP systems have to transform it into a ground model of the background knowledge. The model contains all true ground facts that can be derived from the background knowledge by a SLD-proof tree of depth less than the depth-bound h (Shapiro 1983).

Empirical ILP systems include FOIL (Quinlan 1990; 1991), GOLEM (Muggleton and Feng 1990), LINUS (Lavrac and Dzeroski 1994), mFOIL (Lavrac and Dzeroski 1994), RX (Tangkitvanich and Shimura 1992), MOBAL (Morik et al. 1993), and ML-SMART (Bergadano et al. 1991). FOCL (Pazzani and Kibler 1992) is an extension of FOIL that combines ILP and explanation based learning. CHAM (Kijsirikul et al. 1992a) is an improvement of FOIL by applying a better search heuristic. CHAMP (Kijsirikul et al. 1992b) is an extension of CHAM that can invent useful predicates in learning relations. CHILLIN (Zelle et al. 1994) combines learning methods of GOLEM, FOIL, and CHAMP.

## 3.3. Techniques and methods of ILP

An existing empirical ILP system can be classified into either a bottom-up or top-down learner. Bottom-up systems search for program clauses by considering generalizations. They start from the most specific clause that covers a positive training example and then generalize the clause until it cannot be further generalized without covering some negative examples. Two common generalization techniques are relative least general generalization (rlgg) introduced by Plotkin (1970) and inverse resolution proposed by Muggletion and Buntine (1988). Muggletion (1992) introduces a unifying framework covering both relative least general generalization and inverse resolution, based on the notion of a most specific inverse resolvent.

A successful representative of this class is GOLEM (Muggletion and Feng 1990). GOLEM is based on the construction of relative least-general generalizations which forces the background knowledge to be expressed extensionally as a set of ground facts. This ground model of background knowledge can be excessively large, and the clauses constructed from such models can grow explosively. To tackle this problem, Muggleton and Feng (1990) introduce the notion of ij-determination and employ the language bias of inducing only ij-determinate clauses. GOLEM is also sensitive to the distribution of training examples. If only a random sample of positive training examples is presented, the induced hypothesis of the target predicate is incomplete. Thus, GOLEM may fail to produce general and accurate hypotheses.

Top-down methods apply specialization operators to learn program clauses by searching from general to specific. A specialization operator s produces a set of clauses **C'** permitted by the language bias from a clause c. It typically computes only the set of most general specializations of a clause c under $\theta$-subsumption (Plotkin 1970). Most general specializations can be obtained by performing syntactic and/or semantic operations on the clause c (Shapiro 1983). Two basic syntactic operations on a clause are:

- applying a substitution $\theta$ to the clause, and
- adding a literal to the body of the clause.

One of the most famous empirical top-down ILP system is FOIL (Quinlan 1990; 1991, Cameron-Jones and Quinlan 1993; 1994). It employs the techniques and methods applied in traditional attribute-value based learning systems. It also borrows the idea of specialization operators from MIS (Shapiro 1983) and the method of determining coverage of examples from ML-SMART (Bergadano et al. 1991).

FOIL is restricted to learning function-free program clauses. In other words, constants and functions cannot appear in the induced clauses. The body of a clause is a conjunction of positive or negative literals. Literals in the body have either a predicate symbol $q_i$ from the background knowledge **B**, or the target predicate symbol p. This implies that recursive clauses can be learned. When learning clauses with recursive literals, care must be taken to avoid infinite recursion. FOIL deals with this issue by attempting to establish an ordering on the arguments which may appear in a literal. Many sophisticated methods of finding an ordering on the arguments have been proposed (Cameron-Jones and Quinlan 1993; 1994). For each literal in the body of a clause, at least one of the variables in the arguments of the literal must appear in the head of the clause or in one of the literals to its left.

Training examples are function-free ground facts represented as a set of constant tuples. Background knowledge **B** consists of extensional predicate definitions. Each extensional predicate definition is a finite set of constant tuples representing the concept of the predicate. FOIL uses extensional background knowledge for efficiency reasons. Top-down algorithms can easily use intensionally defined background predicates to evaluate various competing hypotheses. An extension of FOIL, FOCL (Pazzani and Kibler 1992), allows background knowledge to be represented intensionally.

The FOIL algorithm is composed of three main phases. In the first phase, FOIL generates negative examples by applying the closed-world assumption if no negative example is provided. The second phase is the example covering loop. It implements the covering algorithm of AQ and INDUCE (Michalski 1983). The loop constructs a hypothesis by repeatedly performing the following operations:

- construct a clause,
- refine the clause by removing irrelevant literals from the clause,
- add the refined clause to the hypothesis **H**, and
- remove the positive examples covered by the clause from the set of positive training examples

until all the positive examples are covered or no more clause can be constructed. The last phase further refines the induced hypothesis **H** by eliminating irrelevant clauses from the hypothesis. The definitions of irrelevant literal and irrelevant clause are presented by Quinlan (1990).

The procedure that constructs a clause is the most important one in the FOIL algorithm. It starts from the most general clause and repeatedly specializes it by adding a literal to the body of the clause. The clause construction loop continues until a consistent clause covering at least one remaining positive example is found or no more specialization can be performed. During each iteration of the loop , a clause c can be refined by appending different literals to it. FOIL determines which one to be used by employing an information-based heuristics.

If the training examples are imperfect, FOIL may fail to find a consistent clause that covers some positive examples or it may find an overfitting clause that covers only a very few number of positive examples. Usually, these overfitting clauses cannot characterize the regularities in the training examples.

In FOIL, the noise handling mechanism is the encoding length restriction. The idea is that the number of bits required to encode the clause should never exceed the total number of bits needed to indicate explicitly the positive training examples covered by the clause. Thus, if a clause covers r positive examples out of n examples in the training set. The number of bits available to encode the clause is $\log_2(n)+\log_2(\binom{n}{r})$. If there is no bit available for adding another literal, but the clause has more than 85% accurate, it is retained in the induced set of clauses, otherwise the clause is deleted. In the latter case, the clause construction procedure fails to produce a clause and it causes the termination of the FOIL algorithm. This heuristics avoids overfitting the training examples because insignificant literals are excluded from clauses of the inducing hypothesis. The obtained hypothesis is usually smaller, simpler, more accurate, and more comprehensible. Dzeroski and Lavrac (Dzeroski and Lavrac 1993) argue that the encoding length restriction has two deficiencies. In exact domains, it sometimes prevent FOIL from learning complete description. In noisy domains, it allows very specific clauses.

FOIL has been extended to allow literals that bind a variable to a constant to appear in the body of a clause (Quinlan 1991, Cameron-Jones and Quinlan 1993;

1994). Other improvements include determinate literals, types and mode declarations of predicates, and advanced post-processing methods.

A fundamental weakness of FOIL is that recursive hypotheses are evaluated by employing the positive training examples as a model of the target predicate being learned. When the examples are incomplete over the domain of interest, they provide an incorrect model and FOIL has difficulty in learning even simple recursive concepts (Cohen 1993).

mFOIL (Lavrac and Dzeroski 1994) is largely based on the FOIL algorithm. The main difference is that mFOIL uses a different search heuristics and an improved noise-handling mechanism. Another major difference is the beam search strategy used in mFOIL as opposed to the hill-climbing search used in FOIL. To reduce its search space, mFOIL uses some additional information, such as the symmetry and different variables constraints. Several parameters are used in mFOIL, which determine the search heuristics used, the width of the beam in the beam search and the level of significance applied to the induced clauses.

mFOIL employs an accuracy estimate as its search heuristics. The accuracy estimate may be the Laplace estimate or the more sophisticated m-estimate (Cestnik 1990). Both estimates have been found to be useful in improving noise-handling abilities of attribute-value learning systems (Cestnik and Bratko 1991, Clark and Boswell 1991). If a clause c covers n(c) training examples, out of which $n^{+}(c)$ are positive, its expected accuracy can be estimated by either the Laplace estimate $A(c)=\frac{n^{+}(c)+1}{n(c)+2}$ or the m-estimate $A(c)=\frac{n^{+}(c)+m*a\text{-}prior\text{-}prob^{+}}{n(c)+m}$ where $a-prior-prob^{+}$ is the a prior probability of the positive class and is estimated by the relative frequency of positive examples in the whole training set.

It uses a beam search method to find a significant clause. The clause construction procedure starts with the a clause having an empty body. During the search, the best clause and a small set of promising clauses are stored in the beam. At each iteration of the clause construction loop, the significant refinements of each clause c in the beam are evaluated using their expected accuracy. The best of their significant improvements constitute the new beam. A significant improvement of a clause c is a refinement c' of the clause c such that A(c') > A(c) and c' passes the significance test. The search for a clause terminates when the new beam becomes empty. The best clause found so far is retained in the hypothesis if its expected accuracy is better than the default accuracy. The default accuracy, estimated from the entire training set by the relative frequency estimate, is the probability of the more frequent of the positive or negative classes.

The significance test used in mFOIL is based on the likelihood ratio statistic (Kalbfleish 1979). Assume that the training set has $n^+$ positive examples and $n^-$ negative examples. If a clause c cover n(c) examples, $n^+(c)$ of which are positive, the value of the statistic can be calculated as follows:

$$\mathrm{LikelihoodRatio} = 2 * n(c) * \left[ \mathrm{prob}^+(c) * \log\left(\frac{\mathrm{prob}^+(c)}{a - \mathrm{prior} - \mathrm{prob}^+}\right) + \mathrm{prob}^-(c) * \log\left(\frac{\mathrm{prob}^-(c)}{a - \mathrm{prior} - \mathrm{prob}^-}\right)\right]$$

where

$$\mathrm{prob}^+(c) = \frac{n^+(c)}{n(c)},$$

$$\mathrm{prob}^-(c) = \frac{n^-(c)}{n(c)},$$

$$a - \mathrm{prior} - \mathrm{prob}^+ = \frac{n^+}{n^+ + n^-}, \text{ and}$$

$$a - \mathrm{prior} - \mathrm{prob}^- = \frac{n^-}{n^+ + n^-}$$

This statistic is distributed approximately as a $\chi^2$ distribution with one degree of freedom. If its value is above a specified significance threshold, the clause is significant.

The covering algorithm of AQ and INDUCE (Michalski 1983) is used in mFOIL. Program clauses are constructed repetitively. The stopping criteria of the example covering loop terminate the search for clauses when too few positive examples are left for generating a significant clause or no significant clause can be found with expected accuracy greater than the default accuracy.

We have given overviews on evolutionary algorithms and ILP in the last and the current chapters respectively. From next chapter onwards, we shall detail the original contributions of this thesis.

# Chapter 4
# Genetic Logic Programming and Applications

## 4.1. Introduction

As discussed in chapter 3, there have been increasing interests in systems that induce first-order logic programs. The task of inducing logic programs can be formulated as a search problem (Mitchell 1982) in a search space of logic programs. Various systems, such as FOIL (Quinlan 1990; 1991), FOCL (Pazzani and Kibler 1992, Pazzani et al. 1991), CIGOL (Muggleton and Buntine 1988), and GOLEM (Muggleton and Feng 1990), differ mainly in the search strategies and heuristics used to guide the search for the correct program. Most systems are based on a greedy search strategy. They generate a sequence of logic programs from general to specific (or from specific to general) until a consistent target program is found. Each program in the sequence is obtained by specializing or generalizing the previous one. For example, FOIL applies the hill climbing search strategy guided by an information-gain heuristics to search programs from general to specific ones.

However, these strategies and heuristics are not always applicable because they may trap the systems in local maxima. In order to overcome this problem, non greedy strategies should be adopted. Moreover, other learning paradigms such as reinforcement learning (Sutton 1988; 1992, Tesauro 1992, Lin 1992, Kaelbling 1993) and strategy learning cannot be achieved by ordinary ILP systems.

An alternative is Genetic Programming (GP), a very general and domain-independent program induction method. It has impressive applications in symbolic regression, learning of control and game playing strategies, evolution of emergent behavior, evolution of subsumption, automatic programming, concept learning, induction of subroutines and hierarchy of a program, and meta-level learning (Koza 1992; 1994, Kinnear 1994b, Wong and Leung 1995a). Although it is very general, it has little theoretical foundation. Even the most well-established theory of Genetic Algorithms, the schema theory (Holland 1992, Goldberg 1989), cannot be applied directly to GP. The shortcomings of GP are summarized as follows:

- The semantics of the program created are unclear because (a) the semantics of some primitive functions such as LEFT, RIGHT and MOVE (Koza 1992) are difficult to define, and (b) various execution models can be used to execute the programs generated. Thus the semantics of the programs depends on the underlying execution model. It is possible to create two identical programs with different semantics because the underlying execution models are different.
- The underlying execution model must be defined before programs can be created. It means that the users must have some ideas of the solutions.
- It is difficult, if not impossible, to generate recursive programs
- The sub-functions inventing mechanism is restrictive (Koza 1994). In Automatic Defined Function (ADF), the user must decide how many sub-functions should be created, the number of formal arguments in each sub-function and whether these sub-functions can invoke one another.
- A special execution model must be used to run programs with iteration. This model imposes a restriction on where iterations can be introduced in the final programs. This requirement implies that the user must know in advance that the programs being found have iteration.

Since ILP and GP have their own pros and cons, this observation motivates the integration of the two approaches. The Genetic Logic Programming System (GLPS) is the first attempt (Wong and Leung 1994a; 1994b; 1995b) to achieve this goal. It is a novel framework for combining the implicitly parallel search power of GP and knowledge representation power of first-order logic. The shortcomings mentioned above could also be alleviated or eliminated. Currently, GLPS can learn function free first-order logic programs with constants. Section 4.2 presents a description of the mechanism used to generate the initial population of programs. One of the genetic operators, crossover, is detailed in section 4.3. Section 4.4 presents a high level description of GLPS. The results of some applications are presented in section 4.5.

## 4.2. Representations of logic programs

GLPS uses first-order logic to represent background knowledge and training examples and can induce logic programs by genetic search. In this section, we present the representation method of logic programs. GLPS allows atomic formula with variables and constants but does not allow them to contain function symbols.

In GLPS, populations of logic programs are genetically bred using the Darwinian principle of survival and reproduction of the fittest along with a genetic crossover operation appropriate for mating logic programs. The fundamental difficulty in GLPS is to represent logic programs appropriately so that initial population can be generated easily and the genetic operator such as crossover and reproduction can be performed effectively. A logic program can be represented as a forest of AND-OR trees. The leaves of an AND-OR tree are positive or negative literals generated using the predicate symbols and terms of the problem domain. For example, consider the following logic program:

```
C1:  cup(X) :-          insulate_heat(X), stable(X),
                        liftable(X).
C2:  cup(X) :-          paper_cup(X).
C3:  stable(X) :-       bottom(X, B), flat(B).
C4:  stable(X) :-       bottom(X, B), concave(B).
C5:  stable(X) :-       has_support(X).
C6:  liftable(X) :-     has(X, Y), handle(Y).
C7:  liftable(X) :-     small(X), made_from(X, Y),
                        low_density(Y).
```

In this chapter, the syntax of the logic programming language Prolog is used to present logic programs. In comparison with the definitions of logic programs discussed in section 3.2, the symbol ← is replaced by the symbol :- and every clause of a Prolog program must be ended with a full stop. The labels such as C1 and C2 before colons are names used to identify the clauses. These labels and colons are not parts of the logic program.

For the above example, the set of predicate symbols is {cup, insulate_heat, stable, liftable, paper_cup, bottom, flat, concave, has_support, has, handle, small, made_from, low_density} and the set of terms is {X, Y, B}. This program can be represented as in figure 4.1.

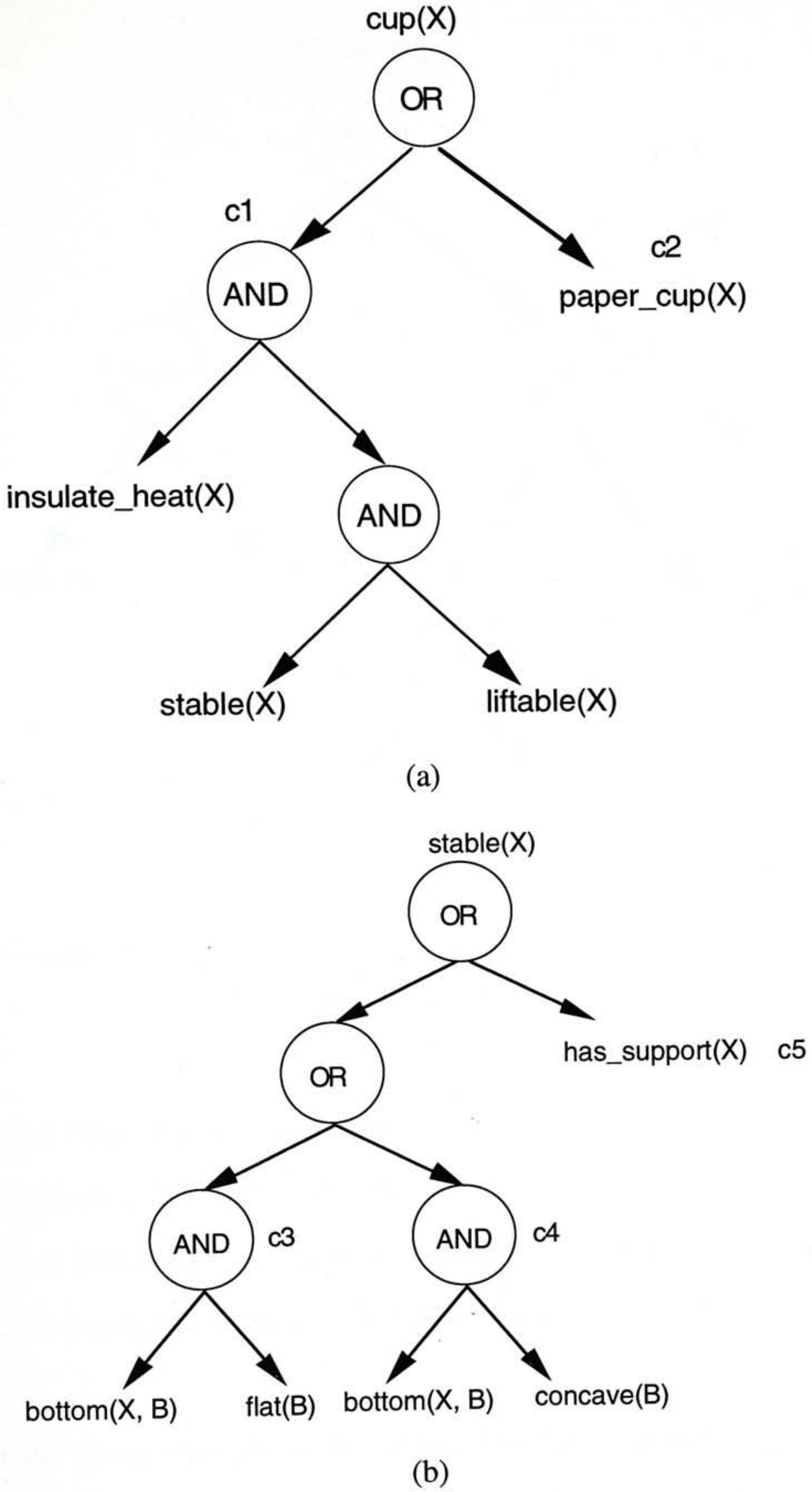
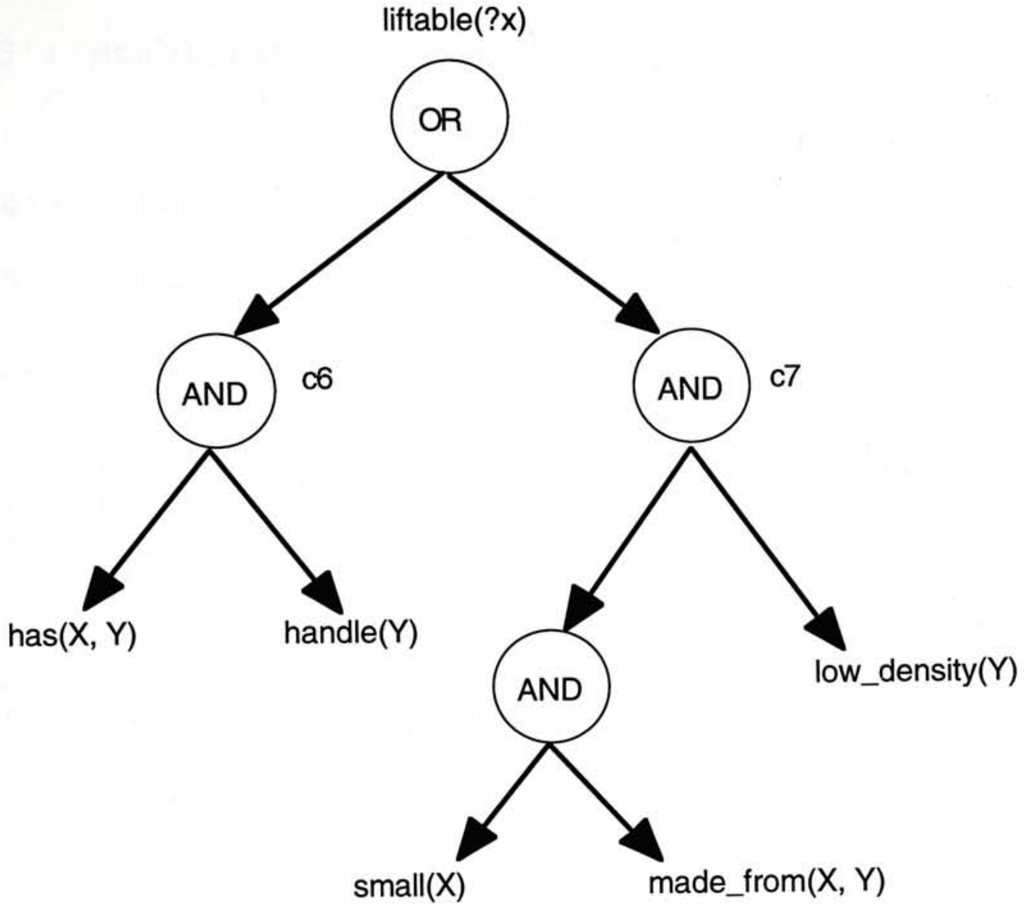


**Figure 4.1: A forest of AND-OR trees that represents a logic program. (a) The representation of the predicate `cup`. (b) The representation of the predicate `stable`. (c) The representation of the predicate `liftable`.**



(c)

**Figure 4.1: (Cont.)**

Since a logic program can be represented by a forest of AND-OR trees, we can randomly generate a forest of AND-OR trees for the program and randomly fill the leaves of these trees with literals of the problem. The high-level description of the algorithm used to generate an initial population is depicted in table 4.1.

For the above example, if the target predicate symbol is `cup`, the predicate symbols for sub-concepts are `stable` and `liftable`, and the set of terms is `{X, Y, B}`. The algorithm generates the following logic programs:

```
C1': cup(X) :-       bottom(Y), handle(B).
C2': cup(X) :-       small(X), insulate_heat(Y).
C3': stable(B) :-    cup(B), paper_cup(X), flat(Y),
                     flat(X).
C4': liftable(X) :-  liftable(Y).
C5': liftable(Y) :-  concave(Y).
```

```
Input:
   Preds:    The set of predicate symbols such as {$p_1$, $p_2$, ...., $p_n$}
   Terms:    The set of terms such as { $t_1$, $t_2$, ..., $t_m$}
   Target:   A special predicate symbol in Preds that indicates the
             target concept to be induced
   Sub:      A set of predicate symbols in Preds that indicate the
             sub-concepts to be learned. If there is no sub-concept in
             the target logic program, then Sub is an empty set.
   Depth:    It specifies the maximum depth of the AND-OR trees to be
             generated.
   Balance:  It is a parameter that controls whether balance or
             unbalance AND-OR trees will be generated.

Output:
   A forest of AND-OR trees representing a logic program.

Comment:
   All predicate symbols represent operational concepts that must be
   defined by either extensional tuples or built-in operations.

Function Generate-Trees(Preds, Terms, Target, Sub, Depth, Balance)
{
   •  Let ALL-CONCEPTS := {Target} ∪ Sub.

   •  Initialize FOREST to an empty set.

   •  Generate a set of literal LITERALS using the predicate symbols
      in Preds and terms in Terms.

   •  For all concepts C in ALL-CONCEPTS do
         •  Create an AND-OR tree for the current concept C. The
            leaves of the AND-OR tree are selected from LITERALS.
         •  Store the AND-OR tree for the current concept C into
            FOREST.

   •  Return (FOREST).
}
```

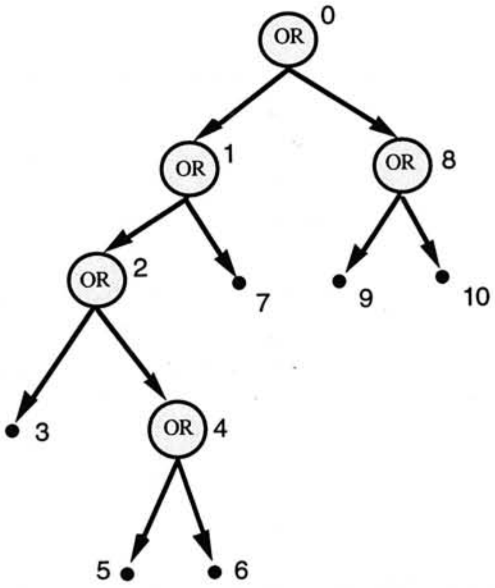
**Table 4.1: The algorithm for generating an initial program randomly.**

Alternatively, an initial population of logic programs can be induced by other learning systems, such as FOIL (Quinlan 1990), using a portion of the training examples. If there are more than one representation for a logic program, one of them will be selected randomly.

## 4.3. Crossover of logic programs

We can apply crossover to the components of a logic program including the whole logic program, the rules, the clauses, and the antecedent literals. In GLPS, the terms of literals cannot be exchanged. Thus crossover components are referred to by a list of numbers. The list can have at most three elements:

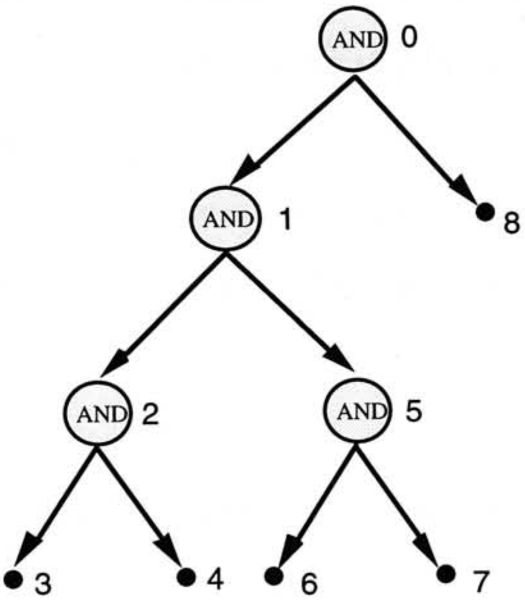
1. { } refers to the whole logic program.
2. {m} refers to the mth rule in the program. A rule has one or more clauses.
3. {m, n'} refers to a clause or a number of clauses of the mth rule in the program where n' is a node number of the corresponding sub-tree. For instance, let the mth rule has $N_m$ clauses which are arranged in an OR-tree as follows:



Each leaf in the tree represents a clause. In the example, the tree has six clauses, i.e. $N_m = 6$. There are 11 nodes in the tree, and the number of nodes is denoted by $N'_m$. n' in the list {m, n'} is between 0 and $N'_{m-1}$.

Thus, {m, n'} represents a clause if n' corresponds to a leaf node. It refers to a set of clauses if n' corresponds to an internal node in the tree.

4. {m, n, l'} refers to a literal or a set of literals of the $n^{th}$ clause of the $m^{th}$ rule where l' is also a node number of the corresponding sub-tree. For example, let the clause has $L_{m,n}$ antecedent literals. These literals are arranged in an AND-tree as follows:



Each leaf in the tree represents an antecedent literal and there are 5 antecedent literals, i.e. $L_{m,n} = 5$. Let the number of nodes in an AND tree be $L'_{m,n}$ which is 9 for the above tree. The third number in {m, n, l'} can have value between 0 and $L'_{m,n} - 1$. {m, n, l'} represents a literal if l' refers to a leaf node. It is a set of literals if l' refers to an internal node.

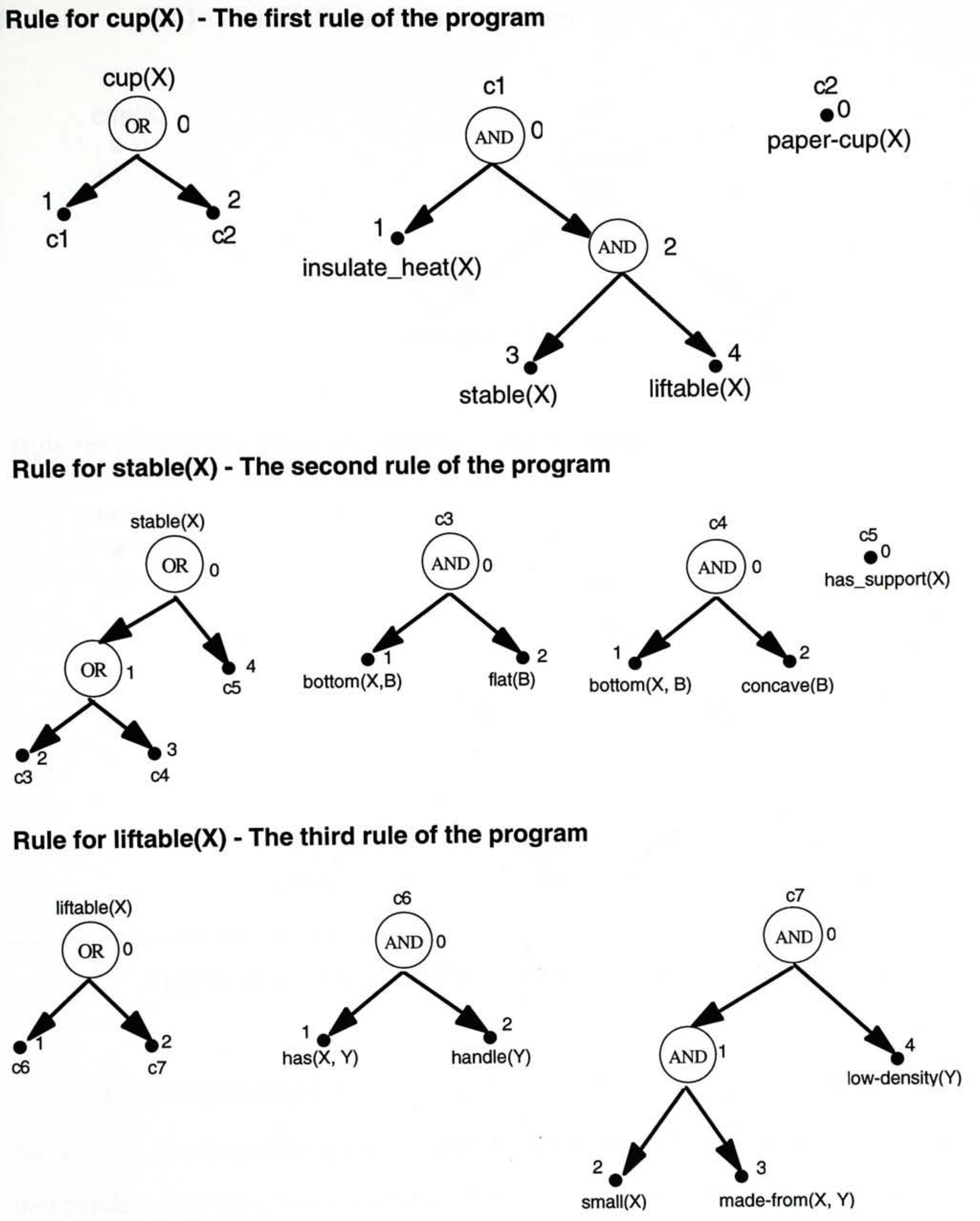
There are four kinds of crossover points represented by the above lists of numbers. Two crossover points are compatible if their representations (i.e. lists) have the same number of elements. In GLPS, crossover between two parental programs can only occur at compatible crossover points. Consider the logic program $Prog_1$ represented in Horn clauses:

```
C1:   cup(X) :-          insulate_heat(X), stable(X),
                         liftable(X).
C2:   cup(X) :-          paper_cup(X).
C3:   stable(X) :-       bottom(X, B), flat(B).
C4:   stable(X) :-       bottom(X, B), concave(B).
C5:   stable(X) :-       has_support(X).
C6:   liftable(X) :-     has(X, Y), handle(Y).
C7:   liftable(X) :-     small(X), made_from(X, Y),
                         low_density(Y).
```

and the logic program Prog$_2$:

```
C1': cup(X) :-           insulate_heat(X), stable(X).
C2': stable(X) :-        bottom(X, B), flat(B),
                         concave(B), has_support(X).
```

The forests of AND-OR trees of Prog$_1$ and Prog$_2$ are depicted respectively in figures 4.2 and 4.3.

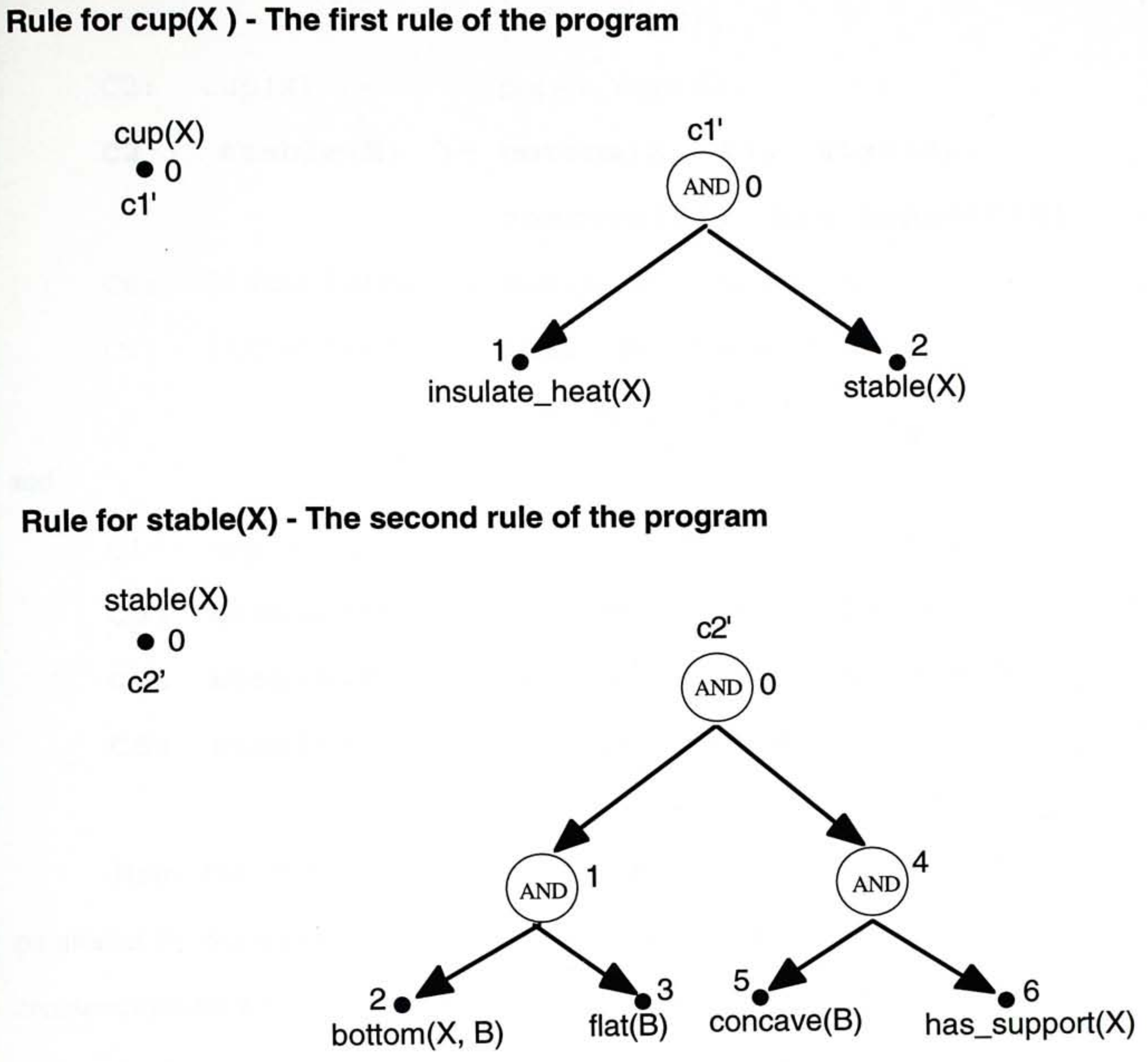


Rule for cup(X) - The first rule of the program

Rule for stable(X) - The second rule of the program

Rule for liftable(X) - The third rule of the program

**Figure 4.2: The And-Or trees of the program $Prog_1$.**



Rule for cup(X ) - The first rule of the program

Rule for stable(X) - The second rule of the program

**Figure 4.3: The And-Or trees of the program $Prog_2$.**

If the crossover points are empty lists {}, the offspring are identical to their parents and the crossover operation degenerates to reproduction. Thus, GLPS has no independent reproduction operation. There is a parameter $P_0$ which controls the probability of reproduction.

The parameter $P_1$ controls the probability of a list with only one element being generated. For instance, if the crossover points are {2} and {2}, the offspring are:

```
C1:   cup(X) :-          insulate_heat(X), stable(Y),
                         liftable(X).
C2:   cup(X) :-          paper_cup(X).
C2':  stable(X)    :-    bottom(X,  B),  flat(B),
                         concave(B),  has_support(B).
C6:   liftable(X) :-     has(X, Y), handle(Y).
C7:   liftable(X) :-     small(X), made_from(X, Y),
                         low_density(Y).
```

and

```
C1':  cup(X) :-          insulate_heat(X), stable(X).
C3:   stable(X)    :-    bottom(X,  B),  flat(B).
C4:   stable(X)    :-    bottom(X,  B),  concave(B).
C5:   stable(X)    :-    has_support(X).
```

Here, the exchanged program fragments are depicted in bold-face. The parameter $P_2$ determines the probability that a list of two elements is generated. If the crossover points are {2, 1} for $Prog_1$ and {2, 0} for $Prog_2$, the offspring are:

```
C1:   cup(X) :-          insulate_heat(X), stable(X),
                         liftable(X).
C2:   cup(X) :-          paper_cup(X).
C2':  stable(X)    :-    bottom(X,  B),  flat(B),
                         concave(B),  has_support(X).
C5:   stable(X) :-       has_support(X).
C6:   liftable(X) :-     has(X, Y), handle(Y).
C7:   liftable(X) :-     small(X), made_from(X, Y),
                         low_density(Y).
```

and

```
C1': cup(X) :-        insulate_heat(X), stable(X).
C3:  stable(X)  :-  bottom(X, B),  flat(B).
C4:  stable(X)  :-  bottom(X, B),  concave(B).
```

The parameter $P_3$ determines the probability that a list of three elements is created. If the crossover points are {2, 3, 0} for $Prog_1$ and {2, 0, 1} for $Prog_2$, the offspring are:

```
C1:  cup(X) :-          insulate_heat(X), stable(X),
                        liftable(X).
C2:  cup(X) :-          paper_cup(X).
C3:  stable(X) :-       bottom(X, B), flat(B).
C4:  stable(X) :-       bottom(X, B),  flat(B).
C5:  stable(X) :-       has_support(X).
C6:  liftable(X) :-     has(X, Y), handle(Y).
C7:  liftable(X) :-     small(X), made_from(X, Y),
                        low_density(Y).
```

and

```
C1': cup(X) :-          insulate_heat(X), stable(X).
C2': stable(X) :-       bottom(X, B),  concave(B),
                        concave(B), has_support(X).
```

Hence, the crossover operation has many effects depending on the crossover points and only generates syntactically valid logic programs.

## 4.4. Genetic Logic Programming System (GLPS)

This section presents the evolutionary process performed by GLPS. It starts with an initial population of first-order logic programs generated randomly, induced by other learning systems, or provided by the user. The initial logic programs are composed of predicate symbols, terms and atomic formulas of the problem domain. An atomic formula can be defined extensionally as a list of tuples for which the formula is true or intensionally as a set of Horn clauses that can compute whether the formula is true. Intensional atomic formulas can also be standard built-in formulas that perform arithmetic, input/output and logical functions etc.

For concept learning (DeJong et al. 1993, Janikow 1993, Greene and Smith 1993), each individual logic program in the population is measured in terms of how well it covers positive examples and excludes negative examples. This measure is the fitness function of GLPS. Typically, each logic program is run over a number of training examples so that its fitness is measured as the total number of misclassified positive and negative examples. Sometimes, if the distribution of positive and negative examples is extremely uneven, this fitness function is not good enough to focus the search. For example,  assume that there are 2 positive and 10000 negative examples, if the number of misclassified examples is used as the fitness function, a logic program that deduces everything are negative will have very good fitness. Thus, in this case, the fitness function should be a weighted sum of the total numbers of misclassified positive and negative examples. GLPS can also learn logic programs computing arithmetic functions such as square root or factorial. In this case, the fitness function calculates the difference between the outputs found by the logic program and the target arithmetic function.

The initial logic programs in generation 0 are normally incorrect and have poor performances. However, some individuals in the population will be fitter than others. The Darwinian principle of reproduction and survival of the fittest and the genetic operation of sexual crossover are used to create new offspring population of programs from the current pôpulation. The reproduction operation involves selecting a program from the current population of programs and allowing it to survive by copying it into the new population. The selection is based on either fitness (fitness proportionate selection) or tournament (tournament selection).

```
Input:
   Preds:    The set of predicate symbols such as {$p_1$, $p_2$, ...., $p_n$}
   Terms:    The set of terms such as { $t_1$, $t_2$, ..., $t_m$}
   Target:   A special predicate symbol in Preds that indicates the
             target concept to be induced
   Sub:      A set of predicate symbols in Preds that indicate the
             sub-concepts to be learned. If there is no sub-concept in
             the target logic programs, then Sub is an empty set.
   Depth:    It specifies the maximum depth of the AND-OR trees to be
             generated.
   Balance:  It is a parameter that controls whether balance or
             unbalance AND-OR trees will be generated.
   t         The termination function.
   f         The fitness function.

Output:
   A logic program induced by GLPS.

Function GLPS(Preds, Terms, Target, Sub, Depth, Balance, t, f)
{
   •  generation := 0.

   •  Initialize a population Pop(generation) of logic programs. They
      are generated by calling the function
      Generate-Trees(Preds, Terms, Target, Sub, Depth, Balance),
      provided from the user, or generated by other learning systems.

   •  Execute each logic program in the Pop(generation) and assign it
      a fitness value according to the fitness function f. The
      fitness value of a program measures how well it covers positive
      examples and excludes negative examples.

   •  While the termination function t is not satisfied do
         •  Create a new population Pop(generation+1) of programs by
            employing the crossover and the mutation. The operations
            are applied to logic programs chosen by either fitness
            proportionate selection or tournament selection.
         •  Evaluate the fitness of each individual in the next
            population Pop(generation+1)
         •  generation := generation + 1.
   •  Return the best logic program found in any generation of a run.
}
```

**Table 4.2: The high-level description of GLPS.**

The genetic process of crossover is used to create two offspring programs from the parental programs selected by either fitness proportionate selection or tournament selection. The parental programs are usually of different sizes and shapes and the offspring programs are composed of the clauses and the literals from their parents.

These offspring programs are typically of different sizes and shapes from their parents. The new generation replaces the old generation after the reproduction and crossover operations are performed on the old generation. The fitness value of each program in the new generation is estimated and the above process is iterated over many generations until the termination criterion is satisfied.

The algorithm will produce populations of programs which tend to exhibit increasing average fitness in producing correct answers for the training examples. GLPS returns the best logic program found in any generation of a run as the result. A high-level description of GLPS is presented in table 4.2.

## 4.5. Applications

An implementation of GLPS is completed. It is implemented in CLOS (Common Lisp Object System). It has been tested on various CLOS implementations and different hardware platforms including CMU Common Lisp on a SparcStation, Lucid Common Lisp on a DecStation and MCL on a Macintosh.

Three applications on learning solved by GLPS are given below as demonstrations, namely, the Winston's arch problem (Winston 1975), the modified Quinlan's network reachability problem (Quinlan 1990), and the factorial problem. Five runs are performed for each problem. The parameters $P_0$, $P_1$, $P_2$ and $P_3$ are 0.0, 0.1, 0.3 and 0.6 respectively. The maximum number of generations of each run is 50 for the first two problems and 20 for the third problem.

### 4.5.1. The Winston's arch problem

In this learning task, the objective is to learn the nature of arches from examples (Winston 1975). The domain has several operational relations. A relation is operation if it is represented extensionally. The operational relations are as follows:

- `supports(A, B)` -- A supports B
- `left-of(A, B)` -- A is on the left of B
- `touches(A, B)` -- A touches B
- `brick(A)` -- A is a brick
- `wedge(A)` -- A is a wedge
- `parallel-piped(A)` -- A is a brick or a wedge.

The non-operational relation `arch(A, B, C)` contains all tuples <A, B, C> that form an arch with lintel A. There are 2 positive and 1726 negative training examples. Since the number of negative examples is much larger than that of positive examples, the standardized fitness is the weighted sum of the number of misclassified examples. Each misclassified positive example has weight 863 while the negative one has weight 1. The predicate symbols are the operational and non-operational predicates described. The set of terms is `{A, B, C}` and the population size is 1000. The maximum number of generations is 50. GLPS can find a near correct program within 2 generations. One of the best programs induced is:

```
arch(A, B, C) :-    left-of(C, B), wedge(C).
arch(A, B, C) :-    left-of(B, C), supports(B, A).
```

The standard solution of this problem is:

```
arch(A, B, C) :-    left-of(B, C), supports(B, A),
                    not touches(B, C).
```

and it is similar to the second clause of the program induced. The completely correct program cannot be induced by GLPS because negative antecedent literals are not allowed in the preliminary implementation. Figure 4.4 delineates the best, average, and worst standardized fitnesses for increasing generations.

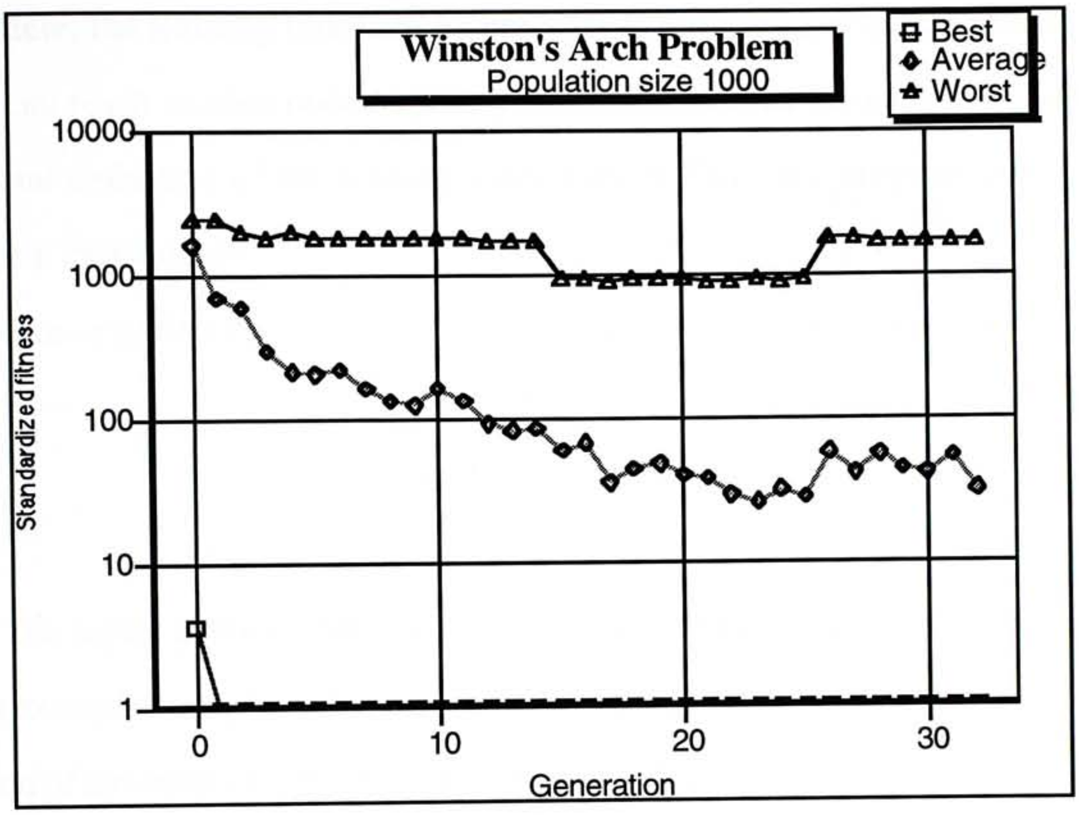
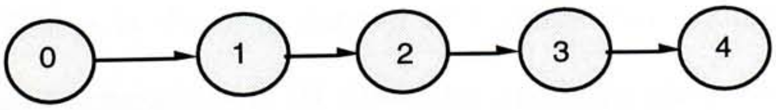


**Figure 4.4: Performance for the Winston's Arch problem.**

## 4.5.2. The modified Quinlan's network reachability problem

The network reachability problem is originally proposed by Quinlan (Quinlan 1990), the domain involves a directional network such as the one depicted as follows:



The structural information of the network is represented by the literal `linked-to(X, Y)` denoting that a node X is directly linked to a node Y. The extension of linked-to(X, Y) is:

```
linked-to(X, Y) = {<0, 1>, <1, 2>, <2, 3>, <3, 4>}
```

Here, the learning task is to induce a logic program that determines whether a node X can reach another node Y. This problem can also be formulated as finding the intensional definition of the relation `can-reach(X, Y)` given its extension. Its extensional definition is:

```
can-reach(X, Y) = {<0, 1>, <0, 2>, <0, 3>, <0, 4>,
                   <1, 2>, <1, 3>, <1, 4>, <2, 3>,
                   <2, 4>, <3, 4>}
```

The tuples of this relation are the positive training examples, and the negative training examples are generated using the close-world assumption. Thus the extensional definition of the relation `not can-reach(X, Y)` is:

```
not can-reach(X, Y) = {<0, 0>, <1, 0>, <1, 1>, <2, 0>,
                       <2, 1>, <2, 2>, <3, 0>, <3, 1>,
                       <3, 2>, <3, 3>, <4, 0>, <4, 1>,
                       <4, 2>, <4, 3>, <4, 4>}
```

In this experiment, the predicate symbols are `can-reach` and `linked-to`. The symbol `can-reach` represents the target concept while `linked-to` is an operational concept. The set of terms is `{X, Y, Z}`. The population size is 1000 and the standardized fitness is the total number of misclassified training examples. The maximum number of generations is 50. Since the symbol `can-reach` is in the set of predicate symbols, it is possible to evolve a non-terminating recursive program such as the following one:

```
can-reach(X, Y) :-   can-reach(Y, X).
```

In order to avoid this problem, an execution time limit is set. If an evolved logic program fails to complete within five seconds, the inference engine will terminate the program and GLPS will assign the worst standardized fitness value, 25, to it.

GLPS can find a perfect program that covers all positive examples while excludes all negative ones within a few generations. One program found is:

```
can-reach(X, Y) :-   linked-to(Z, Y), linked-to(X, Z).
can-reach(X, Y) :-   linked-to(X, Y), linked-to(X, Z).
can-reach(X, Y) :-   can-reach(X, Z), can-reach(Z, Y).
```

This program can be simplified to:

```
can-reach(X, Y) :-   linked-to(X, Z), linked-to(Z, Y).
can-reach(X, Y) :-   linked-to(X, Y).
can-reach(X, Y) :-   can-reach(X, Z), can-reach(Z, Y).
```

The first clause of this program declares that a node X can reach node Y if there is another node Z that directly connects them. The second clause declares that a node X can reach a node Y if they are directly connected. The third clause is recursive, it expresses that a node X can reach a node Y if there is another node Z, such that Z is reachable from X and Y is reachable from Z. In fact, this program is semantically equivalent to the standard solution:

```
can-reach(X, Y) :-   linked-to(X, Y).
can-reach(X, Y) :-   linked-to(X, Z), can-reach(Z, Y).
```

This experiment demonstrates that GLPS can learn recursive program naturally and effectively. Recursive functions are difficult to learn in Koza's GP (Koza 1992). This experiment shows the advantage of GLPS over GP. Figure 4.5 depicts the best, average, and worst standardized fitness values for increasing generations.

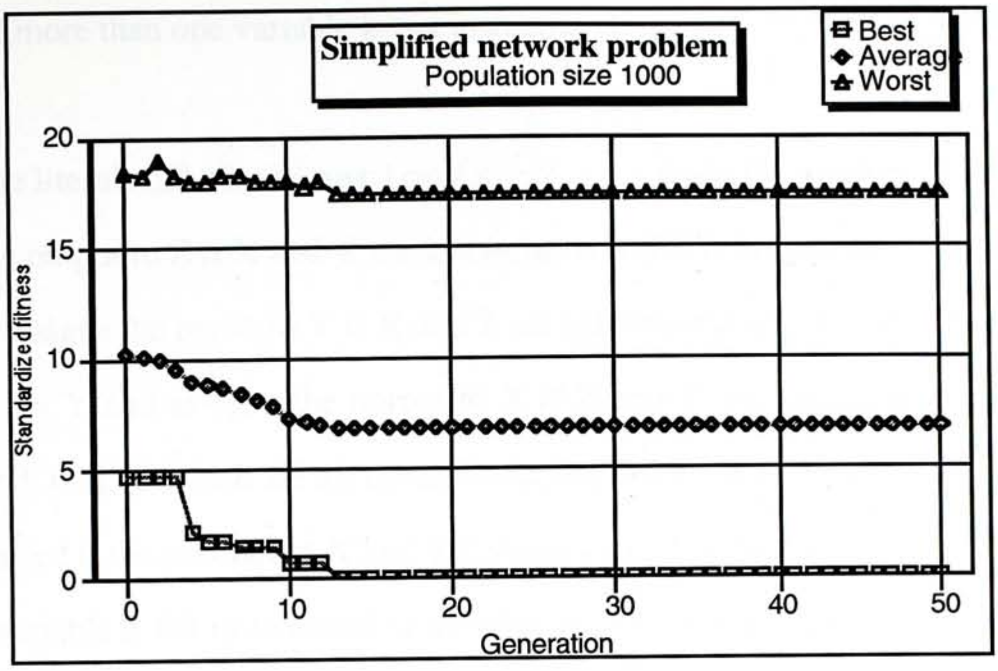


**Figure 4.5: Performance for the modified network reachability problem.**

### 4.5.3. The factorial problem

This experiment learns the relation `factorial(X, Y)` where Y is the factorial of X. The predicate symbols are `factorial`, `plus`, and `multiplication`. The symbol `factorial` represents the target concept while `plus` and `multiplication` are built-in predicates that perform arithmetic operations. The literal `factorial(X, Y)` finds the factorial of X and assigns the result to Y if X is instantiated and Y is not instantiated. It is satisfied if Y is the factorial of X if X and Y are instantiated. The literal is not satisfied if X and Y are not instantiated.

The literal `plus(X, Y, Z)` finds the sum of X and Y and assigns the output to Z if X and Y are instantiated and Z is not instantiated. It finds the difference of Z and X and assigns the result to Y if X and Z are instantiated and Y is not instantiated. It calculated the difference of Z and Y and assigns the output to X if Z and Y are instantiated and X is not instantiated. If X, Y and Z are all instantiated, the literal `plus(X, Y, Z)` is satisfied if the sum of X and Y is equal to Z. The literal is not satisfied if more than one variable is not instantiated.

The literal `multiplication(X, Y, Z)` finds the product of X and Y and assigns the output to Z if X and Y are instantiated and Z is not instantiated. It divides Z by X and assigns the result to Y if X and Z are instantiated and Y is not instantiated. It divides Z by Y and assigns the output to X if Z and Y are instantiated and X is not instantiated. If X, Y and Z are all instantiated, the literal `multiplication(X, Y, Z)` is satisfied if the product of X and Y is equal to Z. The literal is not satisfied if more than one variable is not instantiated or division by zero is attempted.

The set of terms is `{0, 1, 2, W, X, Y, Z}`. The population size is 1000 and the maximum number of generations is 20. The standardized fitness of a program is defined as follows:

$$\sum_{i} \min[1,\ abs\ (\frac{prog\_factorial(i) - factorial(i)}{factorial(i)})]$$

```
where i is the input value;
      factorial(i) returns the correct result for the input i;
      prog_factorial(i) returns the result of the logic program
      for the input i
```

In this experiment, we uses five fitness cases for i from 0 to 4. In order to prevent the problem of non-terminating recursive programs, any evolved program that fails to finish within 100 seconds will be terminated and the worst standardized fitness value, 5, is assigned to this program. A logic program is invoked through the goal `factorial(X, Y)` where X is instantiated to a value between 0 and 4 while Y must be unbound. Since the search space of this problem is extremely large, a number of incorrect initial clauses are used to create the initial population of programs. An individual program contains a random subset of clauses from these incorrect initial clauses. The clauses are as follows:

```
      factorial(0, 1) :-  plus(1, 1, 2).
      factorial(1, 1) :-  plus(1, 1, 2).
      factorial(X, Y) :-  plus(Z, 1, X), plus(X, Y Z).
      factorial(X, Y) :-  plus(Z, X, Y) factorial(Z, W),
                          multiplication(W, X, Y).
      factorial(1, 1) :-  plus(1, 1, 2),
                          multiplication(X, X, Y).
      factorial(X, Y) :-  plus(Z, 1, X),
                          multiplication(Z, Z, W),
                          multiplication(W, X, Y).
      factorial(X, Y) :-  factorial(Z, W),
                          multiplication(W, X, Y),
                          multiplication(X, Y, Z).
      factorial(X, Y) :-  plus(X, X, W),
                          multiplication(W, W, Z),
                          multiplication(Z, X, Y).
      factorial(X, Y) :-  multiplication(X, X, W),
                          factorial(W, Z), plus(Z, X, Y).
```

During one of the runs, the correct logic program is induced in the eighth generation. The program is

```
factorial(0, 1) :-   plus(1, 1, 2).
factorial(X, Y) :-   factorial(Z, W),
                     multiplication(W, X, Y),
                     multiplication(X, Y, Z).
factorial(X, Y) :-   plus(Z, X, Y), factorial(Z, W),
                     multiplication(W, X, Y).
factorial(0, 1) :-   multiplication(W, 0, 1).
factorial(X, Y) :-   multiplication(W, X, Y),
                     multiplication(W, X, Y),
                     multiplication(X, Y, Z).
factorial(1, 1) :-   plus(1, 1, 2),
                     multiplication(X, X, Y).
factorial(X, Y) :-   plus(Z, 1, X), factorial(Z, W),
                     multiplication(W, X, Y).
factorial(X, Y) :-   plus(Z, 1, X), plus(X, Y, Z).
```

It seems that the above program will execute infinitely because of the second clause. In fact, the inference engine can check immediately that this clause cannot be satisfied. The variables Z and W in the second clause are unbound when the sub-goal `factorial(Z, W)` is invoked. Since `factorial(A, B)` fails if A and B are unbound, the sub-goal `factorial(Z, W)` fails and the second clause will not cause infinite recursion.

As described previously, the logic program is invoked through the goal `factorial(X, Y)` where X is instantiated to a value between 0 and 4 while Y must be unbound. Thus, the third clause of the program will fail because two of the variables of the sub-goal `plus(Z, X, Y)` are unbound. Similarly, the fifth and the sixth clauses will also fail. To comprehend the behavior of the program, we remove these clauses and simplify the program to:

```
factorial(0, 1) :-  plus(1, 1, 2).
factorial(0, 1) :-  multiplication(W, 0, 1).
factorial(X, Y) :-  plus(Z, 1, X), factorial(Z, W),
                    multiplication(W, X, Y).
factorial(X, Y) :-  plus(Z, 1, X), plus(X, Y, Z).
```

Since the second clause in the simplified program cannot be satisfied in every situation, it is removed from the program too. Although the last clause is incorrect, it will never be used during execution, so it can be deleted too. The final program is:

```
factorial(0, 1) :-  plus(1, 1, 2).
factorial(X, Y) :-  plus(Z, 1, X), factorial(Z, W),
                    multiplication(W, X, Y).
```

which is a correct logic program to find the factorial of a number. Figure 4.6 depicts the average of the best, average, and worst standardized fitness values over 5 runs against increasing generations.

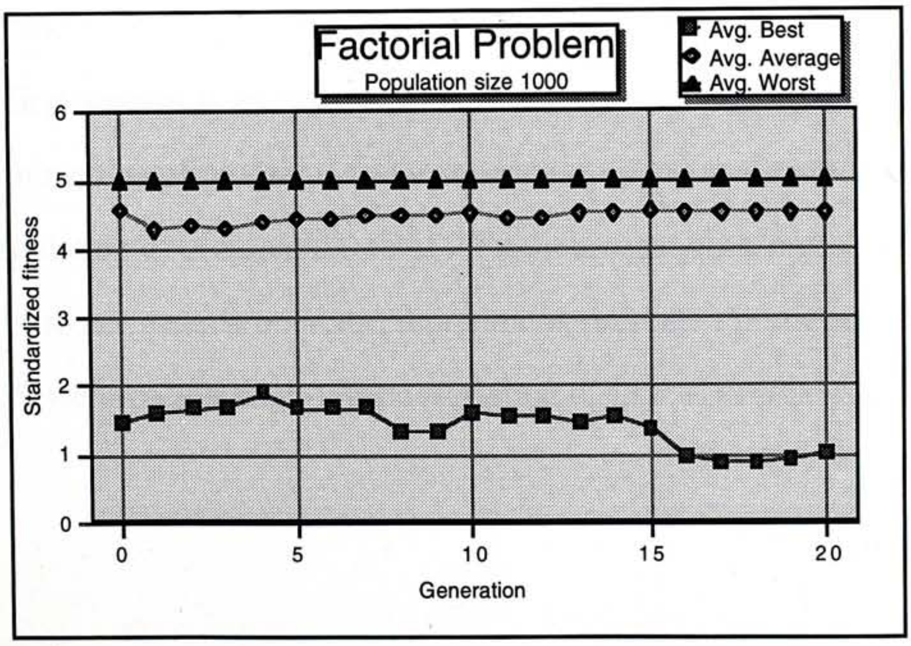


**Figure 4.6: Performance for the factorial problem.**

# Chapter 5
# The logic grammars based genetic programming system (LOGENPRO)

GLPS described in the previous chapter achieves the goal of combining GP and ILP. However, GLPS can only induce logic programs. In theory, programs in any programming language can be represented as parse trees. Hence, GP should be able to learn programs in any programming languages. In practice, the process of translating programs in some languages to the corresponding parse trees is not trivial. Since the syntax of Lisp is so simple and uniform that the translation can be done easily, programs evolved by GP are usually expressed in Lisp.

In this chapter, we propose a novel, flexible, and general framework that combines GP and ILP. This framework is based on a formalism of logic grammars and a system called LOGENPRO (The LOgic grammar based GENetic PROgramming system) is developed. LOGENPRO can learn programs in various programming languages.

The first section is an introduction to logic grammars. Section 5.2 presents a representation method of programs and a description of the mechanism used to generate the initial population of programs. One of the genetic operators, crossover, is detailed in section 5.3. Another genetic operator, mutation, is presented in the subsequent section. In section 5.5, we present a high-level description of LOGENPRO. The last section is a discussion.

## 5.1. Logic grammars

The LOgic grammars based GENetic PROgramming system (LOGENPRO) can induce programs in various programming languages such as Lisp, C, and Prolog. Thus, LOGENPRO must be able to accept grammars of different languages and produce programs in them. Most modern programming languages are specified in the notation of BNF (Backus-Naur form) which is a kind of context-free grammars (CFGs). However, LOGENPRO is based on logic grammars because CFGs (Hopcroft and Ullman 1979, Lewis and Rapadimitrion 1981) are not expressive enough to represent context-sensitive information of some languages and domain-dependent knowledge of the target program being induced. This section introduces the formalism of logic grammars.

Logic grammars are the generalizations of CFGs. Their expressivenesses are much more powerful than those of CFGs, but equally amenable to efficient execution. In this thesis, logic grammars are described in a notation similar to that of definite clause grammars (Pereira and Warren 1980, Pereira and Shieber 1987, Sterling and Shapiro 1986). The logic grammar for some simple S-expressions in table 5.1 will be used throughout this chapter. More logic grammars for S-expressions can be found in the next chapter. Grammars for some logic programming languages can be found in chapters 6 and 7.

```
1:    start          ->    [(*], exp(W), exp(W), exp(W), [)].
2:    start          ->    {member(?x,[W, Z])}, [(*], exp-1(?x),
                           exp-1(?x), exp-1(?x), [)].
3:    start          ->    {member(?x,[W, Z])}, [(+], exp-1(?x),
                           exp-1(?x), exp-1(?x), [)].
4:    exp(?x)        ->    [(/ ?x 1.5)].
5:    exp-1(?x)      ->    {random(1,2,?y)}, [(/ ?x ?y)].
6:    exp-1(?x)      ->    {random(3,4,?y)}, [(- ?x ?y)].
7:    exp-1(W)       ->    [(+ (- W 11) 12)].
```

**Table 5.1: A logic grammar**

A logic grammar differs from a CFG in that the logic grammar symbols, whether terminal or non-terminal, may include arguments. The arguments can be any term in the grammar. A term is either a logic variable, a function or a constant. A variable is represented by a question mark ? followed by a string of letters and/or digits. A function is a grammar symbol followed by a bracketed n-tuple of terms and a constant is simply a 0-arity function. Arguments can be used in a logic grammar to enforce context-dependency. Thus, the permissible forms for a constituent may depend on the context in which that constituent occurs in the program. Another application of arguments is to construct tree structures in the course of parsing, such tree structures can provide a representation of the semantics of the program.

The terminal symbols, which are enclosed in square brackets, correspond to the set of words of the language specified. For example, the terminal `[(- ?x ?y)]` creates the constituent `(- 1.0 2.0)` of a program if `?x` and `?y` are instantiated respectively to 1.0 and 2.0. Non-terminal symbols are similar to literals in Prolog, `exp-1(?x)` in table 5.1 is an example of non-terminal symbol. Commas denote concatenation and each grammar rule ends with a full stop.

The right-hand side of a grammar rule may contain logic goals and grammar symbols. The goals are pure logical predicates for which logical definitions have been given. They specify the conditions that must be satisfied before the rule can be applied. For example, the goal `member(?x, [W, Z])` in table 5.1 instantiates the variable `?x` to either `W` or `Z` if `?x` has not been instantiated, otherwise it checks whether the value of `?x` is either `W` or `Z`. If the variable `?y` has not been bound, the goal `random(1, 2, ?y)` instantiates `?y` to a random floating point number between 1 and 2. Otherwise, the goal checks whether the value of `?y` is between 1 and 2.

Domain-dependent knowledge can be represented in logic goals. For example, consider the following grammar rule:

```
a-useful-program        ->    first-component(?X),
                              {is-useful(?X, ?Y)},
                              second-component(?Y).
```

This rule states that a useful program is composed of two components. The first component is generated from the non-terminal `first-component(?X)`. The logic variable `?X` is used to store semantic information about the first component produced. The logic goal then determines whether the first component is useful according to the semantic information stored in `?X`. Domain-dependent knowledge about which program fragments are useful is represented in the logical definition of this predicate. If the first component is useful, the logic goal `is-useful(?X, ?Y)` is satisfied and some semantic information is stored into the logic variable `?Y`. This information will be used in the non-terminal `second-component(?Y)` to guide the search for a good program fragment as the second component of a useful program.

The special non-terminal `start` corresponds to a program of the language. In table 5.1, some grammar symbols are shown in bold-face to identify the constituents that cannot be manipulated by genetic operators. For example, the last terminal symbol **[)]** of the second rule is revealed in bold-face because every S-expression must be ended with a ')'. The number before each rule is a label for later discussions. It is not part of the grammar.

## 5.2. Representations of programs

The fundamental contribution of LOGENPRO is in the representations of programs in different programming languages appropriately so that initial population can be generated easily and the genetic operators such as reproduction, mutation, and crossover can be performed effectively. A program can be represented as a derivation tree that shows how the program has been derived from the logic grammar. LOGENPRO applies deduction to randomly generate programs and their derivation trees in the language declared by the given grammar. These programs form the initial population. For example, the program `(* (/ W 1.5) (/ W 1.5) (/ W 1.5))` can be generated by LOGENPRO given the logic grammar in table 5.1. It is derived from the following sequence of derivations:

```
start      =>   [(*] exp(W) exp(W) exp(W) [)]
           =>   [(*] [(/ W 1.5)] exp(W) exp(W) [)]
           =>   [(*] [(/ W 1.5)] [(/ W 1.5)] exp(W)
                  [)]
           =>   [(*] [(/ W 1.5)] [(/ W 1.5)]
                [(/ W 1.5)] [)]
           =>   [(* (/ W 1.5) (/ W 1.5) (/ W 1.5))]
```

This sequence of derivations can be represented as the derivation tree depicted in figure 5.1.

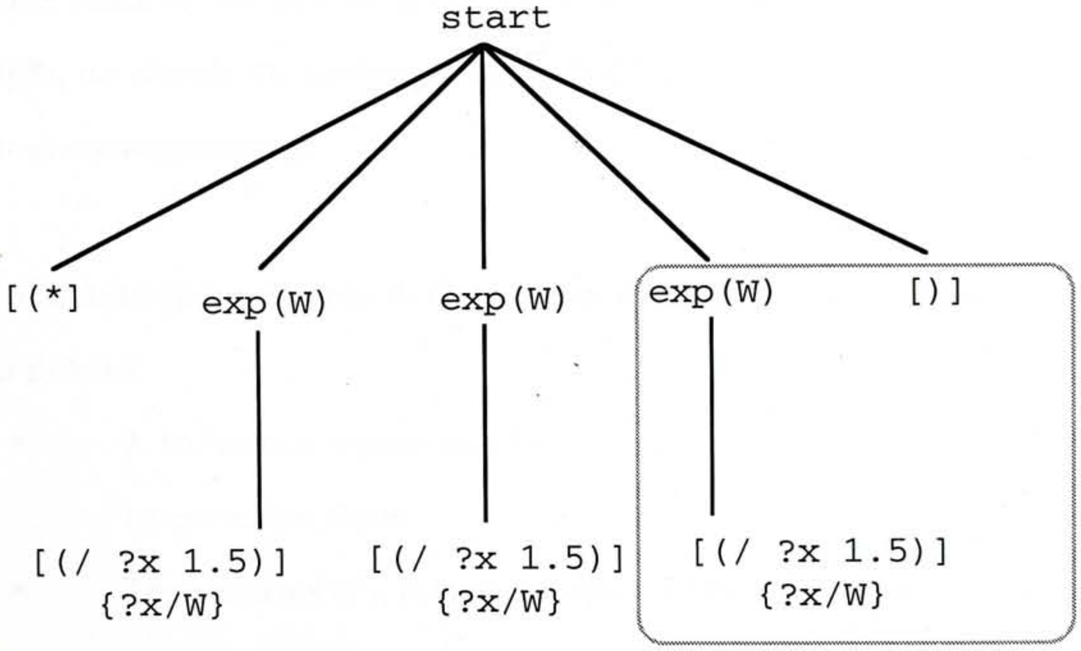


**Figure 5.1: A derivation tree of the S-expression in Lisp** `(* (/ W 1.5) (/ W 1.5) (/ W 1.5))`

In literature, the terms of derivation trees and parse trees are usually used interchangeably. However, we will use the term derivation trees to refer to the tree structures in our framework and the term parse trees to refer to those in GP. The bindings of logic variables are shown in italic font and enclosed in a pair of braces. The sub-trees enclosed in a dashed rectangular are frozen. In other words, they are generated by bold-faced grammar symbols and they cannot be modified by genetic operators.

One advantage of logic grammars is that they specify what is a legal program without any explicit reference to the process of program generation and parsing. Furthermore, a logic grammar can be translated into an efficient logic program that can generate and parse the programs in the language declared by the logic grammar (Pereira and Warren 1980, Pereira and Shieber 1987, Abramson and Dahl 1989). In other words, the process of program generation and parsing can be achieved by performing deduction using the produced logic program. Consequently, the program generation and analysis mechanisms of LOGENPRO can be implemented using a deduction mechanism based on the logic programs translated from the grammars. In the following paragraphs, we discuss the method of implementing LOGENPRO using a Prolog-like logic programming language.

The differences between the logic programming language used and Prolog are listed as follows:

- A variable is represented by a question mark ? followed by a string of letters and/or digits.
- The elements of a list can be separated by either commas or spaces. For example, `[a b c]` and `[a, b, c]` are equivalent.
- A pair of `'|'` is used to represent a frozen terminal symbol. For example, the symbol **`[)]`** in the second rule of the grammar in table 5.1 is translated into |)|.

- A pair of braces encloses a sequence of logic goals appearing in a logic grammar.
- If there are a number of clauses $C_1$, $C_2$, ..., $C_n$ that match a goal G, the ordering of evaluating these clauses is determined randomly.

Using the difference list approach (Sterling and Shapiro 1986), a grammar rule of the form:

$A_0$ -> $A_1$, $A_2$, ..., $A_n$.

is translated into a logic program clause of the form:

$A_0$' :- $A_1$', $A_2$', ..., $A_n$'.

in the logic programming language. Here, if $A_i$ , for some i between 0 and n, is a non-terminal with M arguments, then $A_i$' is a literal with M+3 arguments. The predicate symbols of $A_i$ and $A_i$' are the same. For example, $A_i$ is translated into exp(?X, ?Tree, ?$S_j$, ?$S_{j+1}$), for some j, if $A_i$ is exp(?X). The literal exp(?X, ?Tree, ?$S_j$, ?$S_{j+1}$) states that the sequence of symbols between ?$S_j$ and ?$S_{j+1}$ is a sentence of the category represented by the non-terminal symbol exp(?X). The derivation tree of the sentence is stored in the logic variable ?Tree.

A terminal symbol such as [a b c] is translated to a literal with 3 arguments: connect([a b c], ?$S_j$, ?$S_{j+1}$), for some j. The predicate connect is defined as:

```
connect(?A, ?S0, ?S1) :- append(?A, ?S1, ?S0).
```

This predicate declares that the list of symbols stored in the logic variable ?A can be found in the sequence of symbols between ?S0 and ?S1.

If $A_k$, for some k between 1 and n, is a pair of braces enclosing a sequence of pure logic goals, i.e., $A_k$ has the form of {$G_0$, $G_1$, ...., $G_m$}, then $A_k$' is obtained from $A_k$ by removing the pair of braces.

```
1':   start(tree(start, [(*], ?E1, ?E2, frozen(?E3), |)|),
             ?S0, ?S5)
            :-              connect([(*], ?S0, ?S1),
                            exp(W, ?E1, ?S1, ?S2),
                            exp(W, ?E2, ?S2, ?S3),
                            exp(W, ?E3, ?S3, ?S4),
                            connect([)], ?S4, ?S5).

2':   start(tree(start, {member(?x, [W, Z])}, [(*],
             ?E1, ?E2, frozen(?E3), |)|),?S0, ?S5)
            :-              member(?x, [W, Z]),
                            connect([(*], ?S0, ?S1),
                            exp-1(?x, ?E1, ?S1, ?S2),
                            exp-1(?x, ?E2, ?S2, ?S3),
                            exp-1(?x, ?E3, ?S3, ?S4),
                            connect([)], ?S4, ?S5).

3':   start(tree(start, {member(?x, [W, Z])}, [(+],
             ?E1, ?E2, frozen(?E3), |)|),?S0, ?S5)
            :-              member(?x, [W, Z]),
                            connect([(+], ?S0, ?S1),
                            exp-1(?x, ?E1, ?S1, ?S2),
                            exp-1(?x, ?E2, ?S2, ?S3),
                            exp-1(?x, ?E3, ?S3, ?S4),
                            connect([)], ?S4, ?S5).

4':   exp(?x, tree(exp(?x), [(/ ?x 1.5)]),?S0, ?S1)
            :-              connect([(/ ?x 1.5)], ?S0, ?S1).

5':   exp-1(?x, tree(exp-1(?x), {random(1,2,?y)},
             [(/ ?x ?y)]),?S0, ?S1)
            :-              random(1, 2, ?y),
                            connect([(/ ?x ?y)], ?S0, ?S1).

6':   exp-1(?x, tree(exp-1(?x), {random(3,4,?y)},
             [(- ?x ?y)]),?S0, ?S1)
            :-              random(3, 4, ?y),
                            connect([(- ?x ?y)], ?S0, ?S1).

7':   exp-1(W, tree(exp-1(W), [(+ (- W 11) 12)]),?S0, ?S1)
            :-              connect([(+ (- W 11) 12)], ?S0, ?S1).
```

**Table 5.2: A logic program obtained from translating the logic grammar presented in table 5.1**

This method of translating a logic grammar into a logic program is common in the field of natural language processing (Pereira and Warren 1980, Pereira and Shieber 1987, Abramson and Dahl 1989). The original idea of this approach is to rephrase the special purpose formalism of CFGs into a general purpose first-order predicate logic

(Kowalski 1979, Colmerauer 1978, Pereira and Warren 1980). This approach is further refined and generalized to Define Clause Grammars (DCGs) which can handle the properties of context-dependency of natural languages effectively.

Since DCGs, a kind of logic grammars, can be translated into efficient logic programs automatically, parsers and generators for the corresponding natural languages can be obtained easily. In other words, researchers in the field of natural language processing only declare the grammar for a particular natural language, and the translation process will produce the corresponding parser and generator for them. Moreover, for some cases, the same logic program can be used as both a parser and generator at the same time.

For example, the grammar depicted in table 5.1 can be translated into the logic program presented in table 5.2. In the clause 1' of the logic program shown in table 5.2, the compound term `tree(start, [(*], ?E1, ?E2, frozen(?E3), |)|)` indicates that it is a tree with a root labeled as `start`. The children of the root include the terminal symbol `[(*]`, a tree created from the non-terminal `exp(W)`, another tree created from the non-terminal `exp(W)`, a frozen tree generated from the non-terminal **`exp(W)`**, and the frozen terminal `|)|`.

Thus, a derivation tree can be generated randomly by issuing the following query:

```
    ?- start(?T, ?S, []).
```

This goal can be satisfied by deducing a sentence that is in the language specified by the grammar. One solution is:

```
    ?S = [(* (/ W 1.5) (/ W 1.5) (/ W 1.5))]
```

and the corresponding derivation tree is:

```
?T = tree(start, [(*],
           tree(exp(W), [(/ W 1.5)]),
           tree(exp(W), [(/ W 1.5)]),
           frozen(tree(exp(W), [(/ W 1.5)])),
           |)|)
```

This is exactly a representation of the derivation tree shown in figure 5.1. In fact, the bindings of all logic variables and other information are also maintained in the derivation trees to facilitate the genetic operations that will be performed on the derivation trees.

Alternatively, initial programs can be induced by other learning systems such as FOIL (Quinlan 1990; 1991) or given by the user. LOGENPRO analyzes each program and creates the corresponding derivation tree. If the language is ambiguous, multiple derivation trees can be generated. LOGENPRO produces only one tree randomly. For example, the program `(* (/ W 1.5) (/ W 1.5) (/ W 1.5))` can also be derived from the following sequence of derivations:

```
start     =>    {member(?x, [W, Z])} [(*] exp-1(?x)
                exp-1(?x) exp-1(?x) [)]
          =>    [(*] exp-1(W) exp-1(W) exp-1(W) [)]
          =>    [(*] {random(1, 2, ?y)} [(/ W ?y)]
                exp-1(W) exp-1(W) [)]
          =>    [(*] [(/ W 1.5)] exp-1(W) exp-1(W) [)]
          =>    [(*] [(/ W 1.5)]
                {random(1, 2, ?y)} [(/ W ?y)]
                exp-1(W) [)]
```

```
=>   [(*] [(/ W 1.5)] [(/ W 1.5)]
     exp-1(W) [)]
=>   [(*] [(/ W 1.5)] [(/ W 1.5)]
     {random(1, 2, ?y)} [(/ W ?y)] [)]
=>   [(*] [(/ W 1.5)] [(/ W 1.5)]
     [(/ W 1.5)] [)]
=>   [(* (/ W 1.5) (/ W 1.5) (/ W 1.5))]
```

The derivation tree of this sequence of derivations is depicted in figure 5.2. The ?y1, ?y2, and ?y3 in the figure are different instances of the logic variable ?y appearing in the same or different rules in the grammar.

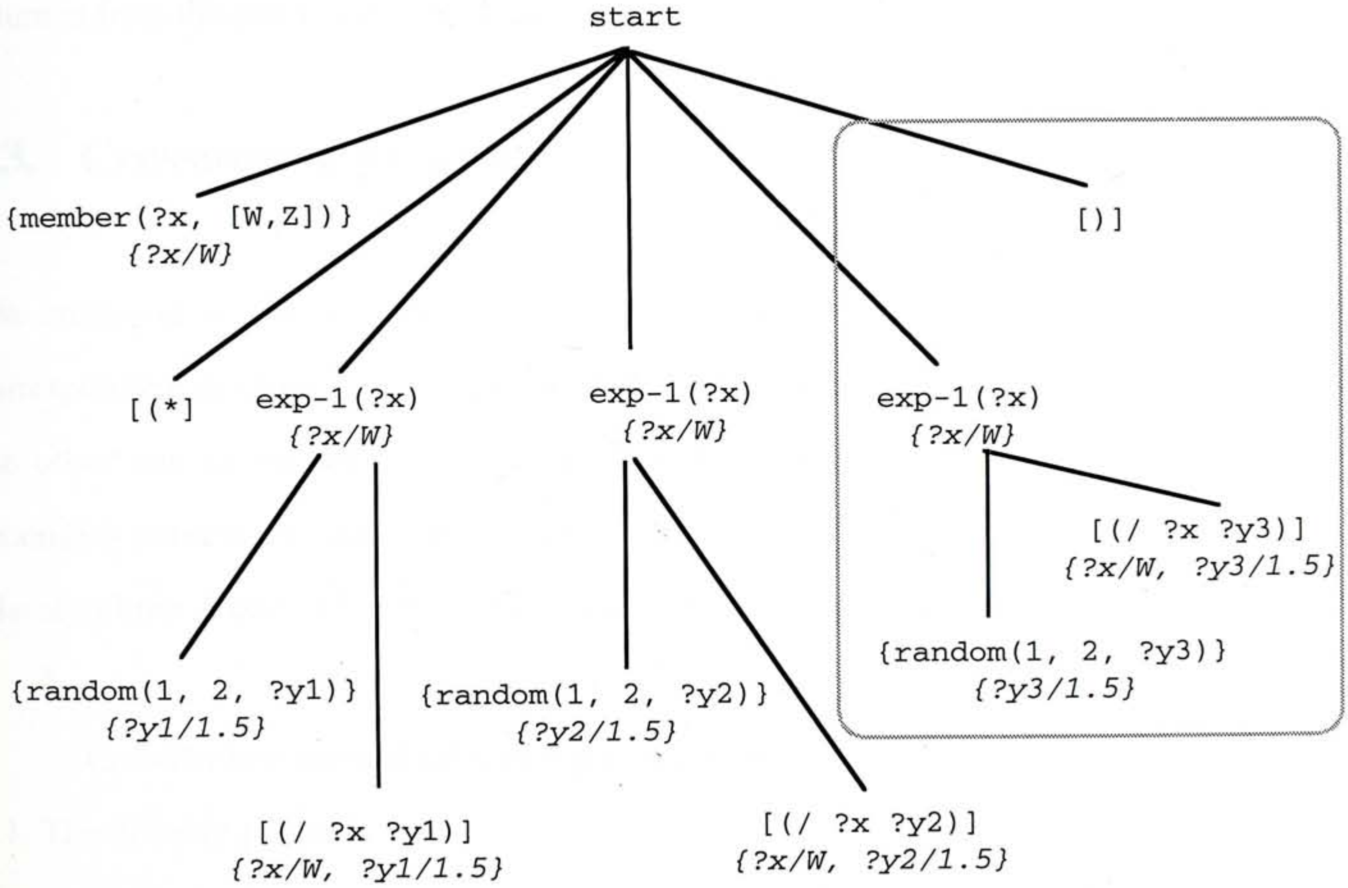


**Figure 5.2: Another derivation tree of the S-expression in Lisp** (* (/ W 1.5) (/ W 1.5) (/ W 1.5))

Using the logic program in table 5.2, a given program such as `(* (/ W 1.5) (/ W 1.5) (/ W 1.5))` can be analyzed using the following query:

```
?- start(?T, [(* (/ W 1.5) (/ W 1.5) (/ W 1.5))], []).
```

The given program is correct if the above goal can be satisfied and the corresponding derivation tree will be bound to the logic variable `?T`. As demonstrated previously, the logic grammar in table 5.1 is ambiguous and thus the corresponding logic program may produce multiple derivation trees for a given program. Since the search strategy of the underlying deduction mechanism selects randomly one clause to explore with backtracking from all unifiable clauses, the sequence of generating the derivation trees of a particular program is also random. Consequently, LOGENPRO takes the first tree returned from the query to represent the given program.

## 5.3. Crossover of programs

The crossover is a sexual operation that starts with two parental programs and the corresponding derivation trees. One program is designated as the primary parent and the other one as the secondary parent. The derivation trees of the primary and secondary parents are called the primary and secondary derivation trees respectively. The algorithm in table 5.3 is used to produce an offspring program.

Consider two parental programs generated randomly from the grammar in table 5.1. The primary parent is `(+ (- Z 3.5) (- Z 3.8) (/ Z 1.5))` and the secondary parent is `(* (/ W 1.5) (+ (- W 11) 12) (- W 3.5))`. The corresponding derivation trees are depicted in figures 5.3 and 5.4 respectively. In the figures, the shadowed numbers identify sub-trees of these derivation trees, while the underlined numbers indicate the grammar rules used in deducing the corresponding sub-trees.

```
Input:
   P:               The primary derivation tree.
   S:               The secondary derivation tree.

Output:
   Return a new derivation tree if a valid offspring can be obtained by performing crossover, otherwise return false.

Function crossover(P, S)
{
      1.    Find all sub-trees of the primary derivation tree P and store them into a global variable PRIMARY-SUB-TREES, excluding the primary derivation tree, all logic goals, and frozen sub-trees.
      2.    Find all sub-trees of the secondary derivation tree S and store them into a global variable SECONDARY-SUB-TREES, excluding all logic goals and frozen sub-trees.
      3.    If the variable PRIMARY-SUB-TREES is not empty, select randomly a sub-tree from it using a uniform distribution. Otherwise, terminate the algorithm without generating any offspring program.
      4.    Designate the sub-tree selected as the SEL-PRIMARY-SUB-TREE and the root of it as the PRIMARY-CROSSOVER-POINT. Remove the SEL-PRIMARY-SUB-TREE from the variable PRIMARY-SUB-TREES.
      5.    Copy the variable SECONDARY-SUB-TREES to the temporary variable TEMP-SECONDARY-SUB-TREES.
      6     If the variable TEMP-SECONDARY-SUB-TREES is not empty, select randomly a sub-tree from it using a uniform distribution. Otherwise, go to step 3.
      7.    Designate the sub-tree selected in step 6 as the SEL-SECONDARY-SUB-TREE. Remove it from the variable TEMP-SECONDARY-SUB-TREES.
      8.    If the offspring produced by performing crossover between the SEL-PRIMARY-SUB-TREE and the SEL-SECONDARY-SUB-TREE is invalid according to the grammar, go to step 6. The validity of the offspring generated can be checked by the procedure is-valid(P, SEL-PRIMARY-SUB-TREE, SEL-SECONDARY-SUB-TREE).
      9.    Copy the genetic materials of the primary parent P to the offspring, remove the SEL-PRIMARY-SUB-TREE from it and then impregnating a copy of the SEL-SECONDARY-SUB-TREE at the PRIMARY-CROSSOVER-POINT.
     10.    Perform some house-keeping tasks and return the offspring program.
}
```

**Table 5.3: The crossover algorithm of LOGENPRO**

```
Input:
   P:          The primary derivation tree
   P-sub-tree: The sub-tree in the primary derivation tree that is
               selected to be crossed over.
   S-sub-tree: The sub-tree in the secondary derivation tree that
               is selected to be crossed over.

Output:
   Return true if the offspring generated is valid, otherwise return
   false.

Function is-valid(P, P-sub-tree, S-sub-tree)
{
      11.   Find all siblings of the P-sub-tree in P and store them
            into the global variable SIBLINGS.
      12.   For each sub-tree in the variable SIBLINGS, perform the
            following sub-steps:
               • Store the bindings of the sub-tree to the global
                 variable BINDINGS.
               • For each logic variable in the variable BINDINGS
                 that is not instantiated by the sub-tree,  remove
                 it from the variable BINDINGS.
               • Modify the bindings of the sub-tree.
      13.   Modify the bindings of the S-sub-tree. A logic variable
            is retained only if it is instantiated in the S-sub-tree.
      14.   If there is a rule in the grammar such that:
               • it is satisfied by the sub-trees in the variable
                 SIBLINGS and the S-sub-tree,
               • the sub-trees in the variable SIBLINGS and the S-
                 sub-tree are used exactly once,
               • the sub-trees are applied in the same order as that
                 in the original rule of the primary derivation
                 tree, and
               • a consistent conclusion C is deduced from the rule.
                 The conclusion is consistent if the function
                 is-consistent(P, PARENT, C) returns true where
                 PARENT is the parent of the P-sub-tree. The
                 function is-consistent is presented in table 5.5.
            then the offspring generated will be valid. Otherwise,
            the offspring will be invalid.
}
```

**Table 5.4: The algorithm that checks whether the offspring produced by LOGENPRO is valid.**

```
Input:
   P:        The primary derivation tree.
   PARENT:   The parent of the primary sub-tree.
   C:        The conclusion.

Output:
   Return true if the conclusion C is consistent, otherwise return
   false.

Comment:
   This operation can be viewed as performing a tentative crossover
   between PARENT and C and then determining whether the tentative
   offspring produced is valid. Here, PARENT is treated as the
   primary sub-tree while C is treated as the secondary sub-tree of
   the tentative crossover operation. The main difference between
   this algorithm and that in table 5.4 is that all rule applications
   in all ancestors of PARENT must be maintained.

Function is-consistent(P, PARENT, C)
{
      15.   If PARENT is the root of P then
                  if C is labeled with the symbol start then
                        return true
                  else false.
      16.   Find all siblings of PARENT in P and store them into the
            global variable SIBLINGS.
      17.   For each sub-tree in the variable SIBLINGS, perform the
            following sub-steps:
               •  Store the bindings of the sub-tree to the global
                  variable BINDINGS.
               •  For each logic variable in the variable BINDINGS
                  that is not instantiated by the sub-tree, remove
                  it from the variable BINDINGS.
               •  Modify the bindings of the sub-tree.
      18.   Let the grammar rule applied in the parent node of PARENT
            as RULE.
            If the following conditions are satisfied:
               •  RULE is satisfied by the sub-trees in the variable
                  SIBLINGS and C,
               •  the sub-trees in SIBLINGS and C are used exactly
                  once and the ordering of applications is
                  maintained, and
               •  a consistent conclusion C' is deduced from RULE.
                  The conclusion is consistent if the function
                  is-consistent(P, GRANDPARENT, C') returns true
                  where GRANDPARENT is the parent node of PARENT.
            then
                  return true
            else
                  return false.
 }
```

**Table 5.5: The algorithm that checks whether a conclusion deduced from a rule is consistent with the direct parent of the primary sub-tree.**

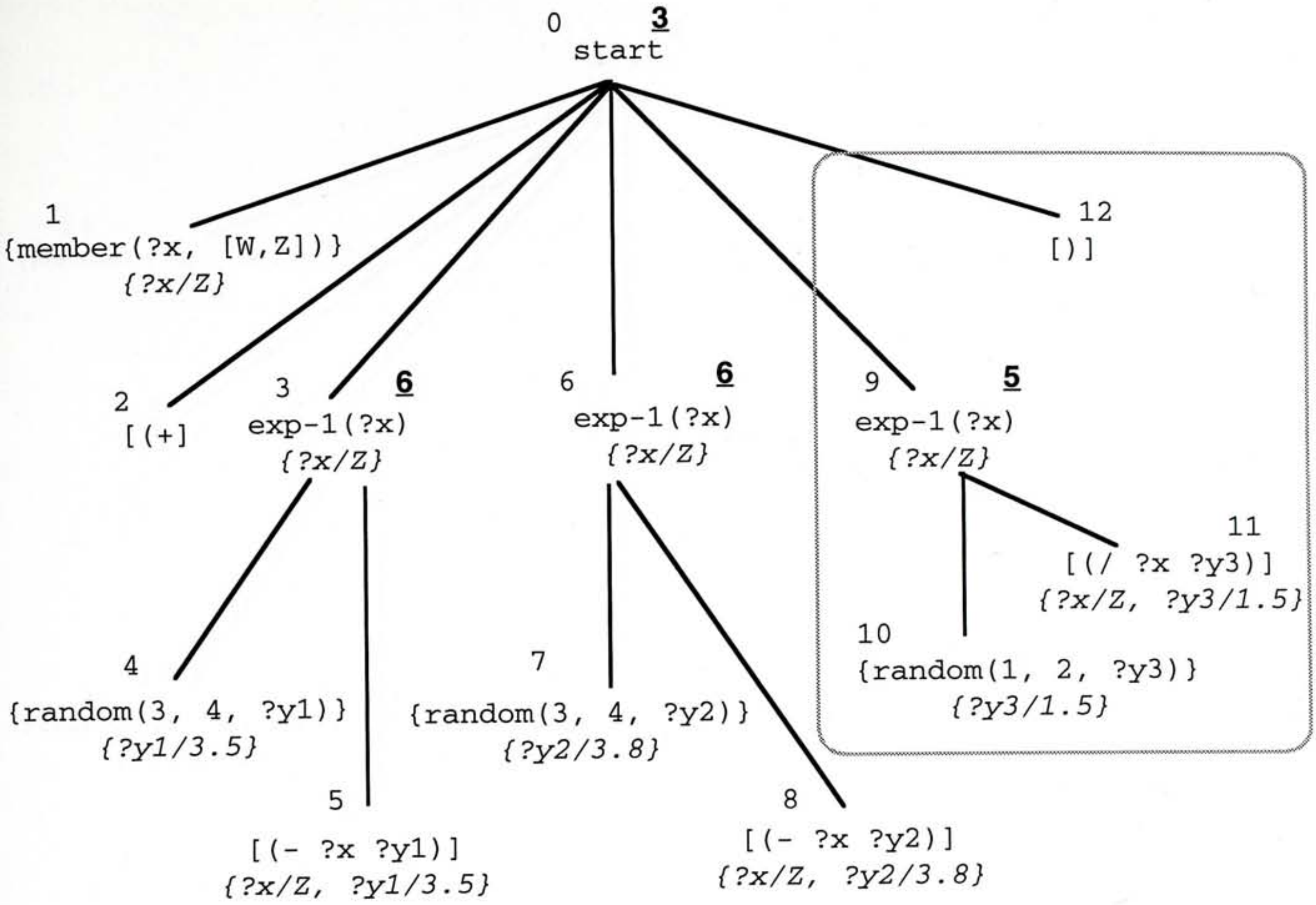


**Figure 5.3: The derivations tree of the primary parental program `(+ (- Z 3.5) (- Z 3.8) (/ Z 1.5))`**

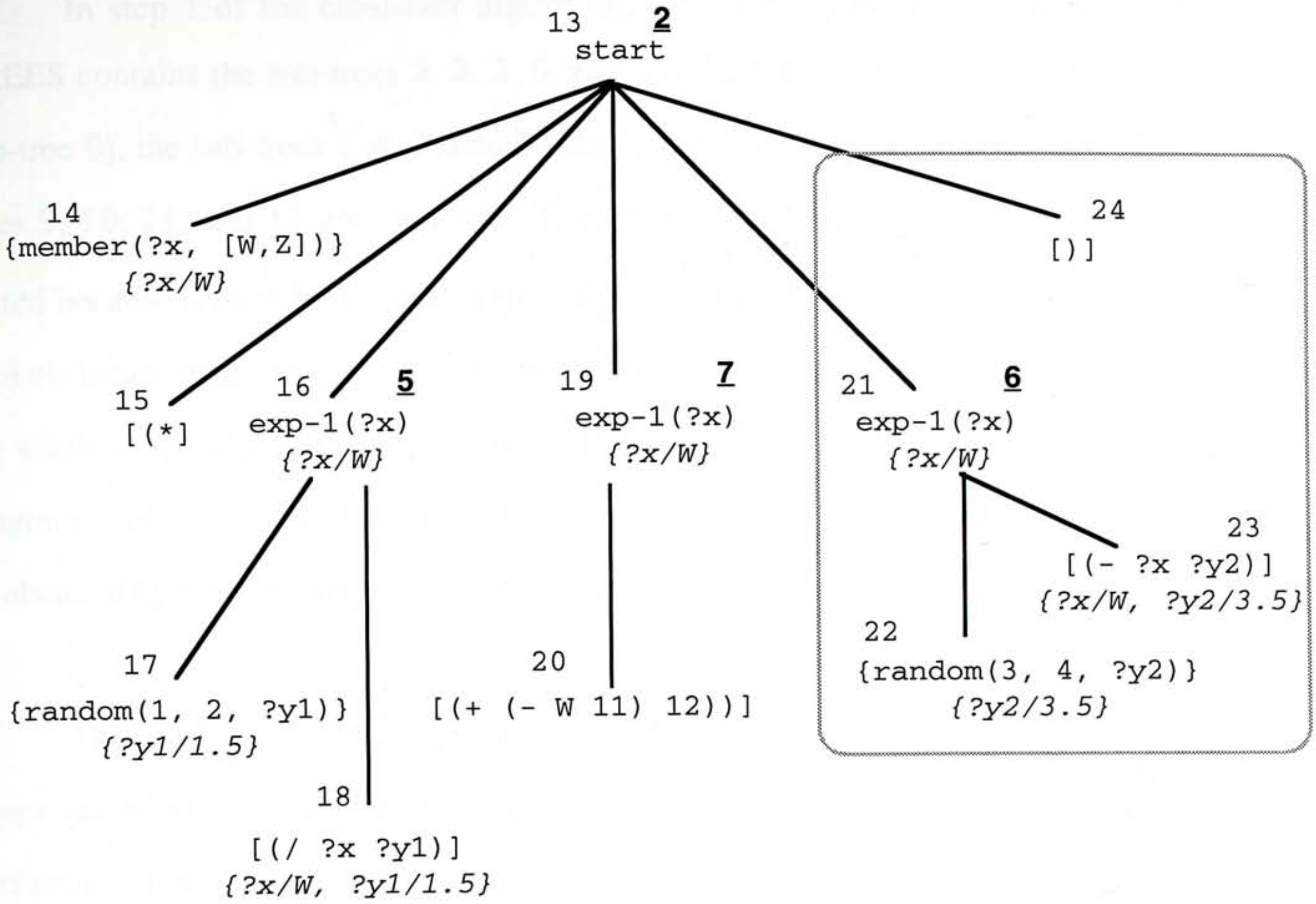


**Figure 5.4: The derivations tree of the secondary parental program**

**(* (/ W 1.5) (+ (- W 11) 12) (- W 3.5))**

In step 1 of the crossover algorithm, the global variable PRIMARY-SUB-TREES contains the sub-trees 2, 3, 5, 6, and 8,. The primary derivation tree (i.e. the sub-tree 0), the sub-trees 1, 4, 7, and 10 that contain logic goals, and the frozen sub-trees 9, 10, 11, and 12 are excluded. The whole primary derivation tree cannot be mated because it must be generated from the grammar symbol `start`. If the symbol `start` is not recursive (i.e. `start` does not appear on the right hand side of a rule), the whole secondary derivation tree must be chosen for crossover. Thus, the offspring program must be a copy of the secondary parental program. In fact, the same effect can be obtained by reproducing the secondary parental program.

The sub-trees containing logic goals are eliminated for two reasons. Firstly, the crossover algorithm can be greatly simplified if logic goals are prevented from performing crossover. Secondly, logic goals specify the conditions that must be satisfied before the rule can be applied and/or the computations that should be done. Hence, from the viewpoint of natural selection and reproduction, the interpretation of crossover between logic goals is unclear and unnatural. Thus this kind of operations is avoided.

Similarly, the sub-trees 13, 15, 16, 18, 19, and 20 are assigned to the global variable SECONDARY-SUB-TREES in step 2. In the next step, a sub-tree in the variable PRIMARY-SUB-TREES is selected randomly using a uniform distribution because the variable is not empty. Assume that the sub-tree 2, the SEL-PRIMARY-SUB-TREE, is selected. Thus, it is removed from the variable PRIMARY-SUB-TREES in step 4. A copy of the variable SECONDARY-SUB-TREES is made and stored into the global variable TEMP-SECONDARY-SUB-TREES in step 5.

Steps 6 to 8 form a loop that finds an appropriate sub-tree from the variable TEMP-SECONDARY-SUB-TREES. A sub-tree, SEL-SECONDARY-SUB-TREE, is appropriate if a valid offspring can be obtained by executing crossover between the SEL-PRIMARY-SUB-TREE and the SEL-SECONDARY-SUB-TREE. If no appropriate sub-tree can be found in this loop, the algorithm returns back to step 3 to find another SEL-PRIMARY-SUB-TREE. Assume that the sub-tree 𝟙𝟝 is chosen as the SEL-SECONDARY-SUB-TREE. Step 8 determines whether a valid offspring can be obtained. It is the most complicate procedure in this algorithm and it is delineated in table 5.4 and explained in the following paragraphs.

In step 11 of the algorithm shown in table 5.4, the sub-trees 𝟙, 𝟛, 𝟞, 𝟡, and 𝟙𝟚 are found to be the siblings of the SEL-PRIMARY-SUB-TREE 𝟚 and stored into the global variable SIBLINGS. The SIBLINGS can be thought as the context around the PRIMARY-CROSSOVER-POINT and the context's consistency has to be checked and computed. The purpose of step 12 is to remove the bindings established solely by the SEL-PRIMARY-SUB-TREE which will be deleted by the crossover operation. To achieve this goal, the bindings of each sub-tree in the variable SIBLINGS is modified so that only the bindings established by itself is retained. The bindings instantiated by a sub-tree can be found easily using the techniques of explanation-based learning (DeJong 1993, Mitchell et al 1986, DeJong and Mooney 1986). For example, the bindings {?x/Z} of the sub-tree 𝟙 need not be modified because the logic variable ?x is instantiated to the value Z by the logic goal member(?x, [W, Z]). The bindings {?x/Z} of the sub-tree 𝟛 is changed to an empty list because the logic variable ?x is bound to the value Z by the sub-tree 1. Similarly, the bindings {?x/Z} of the sub-trees 𝟞 and 𝟡 are changed to empty lists. The bindings of the sub-tree 𝟙𝟚 is not changed because it is already empty.

In step 13, the bindings of the SEL-SECONDARY-SUB-TREE is modified so that only the bindings established by itself is retained. The purpose is to identify the effect of the sub-tree on the logic variables. In this example, since the grammar symbol of the SEL-SECONDARY-SUB-TREE 𝟙𝟝 has no argument, its bindings is empty. In fact, the primary and secondary derivation trees are pre-processed by LOGENPRO using an algorithm based on the techniques of Explanation-Based Learning (EBL). The algorithm finds the bindings established solely by the corresponding sub-trees of the derivation trees. The results are stored in the sub-trees so that they can be retrieved in constant time $C_r$. Thus the time complexity of step 12 is O(n) where n is the number of sub-trees in the global variable SIBLINGS. Similarly, the time complexity of step 13 is O(1).

In step 14, the second grammar rule is satisfied by the sub-trees in SIBLINGS and the SEL-SECONDARY-SUB-TREE. Moreover, this rule reaches the conclusion `start` which is consistent with the requirement of the parent, the sub-tree 0, of the SEL-PRIMARY-SUB-TREE. Thus, the offspring generated is valid. The procedure that checks whether a conclusion is consistent is presented in table 5.5.

In step 9 of the crossover algorithm in table 5.3, the offspring is generated. In the next step, it is returned as the solution after some house-keeping tasks have been performed. The house-keeping tasks update the bindings and the rule numbers of the sub-trees of the offspring. The offspring program of this example is `(* (- Z 3.5) (- Z 3.8) (/ Z 1.5))` and its derivation tree is shown in figure 5.5. It is interesting to find that the sub-tree 27 has the rule number 2. This indicates that the sub-tree is generated by the second grammar rule rather than the third rule applied to the primary parent. The second rule must be used because the terminal symbol `[(+]` is changed to `[(*]` and only the second rule can create the terminal `[(*]`. In fact, this situation is identified in step 14 of the function `is-valid` and a record is maintained so that the rule number can be changed to 2 by the house-keeping procedure.

In another example, the same primary and secondary parents are used. Assume that the SEL-PRIMARY-SUB-TREE 3 is selected in step 3 and the SEL-SECONDARY-SUB-TREE 16 is chosen in step 7 of the crossover algorithm. Now, the siblings of the SEL-PRIMARY-SUB-TREE 𝟛 are the sub-trees 𝟙, 𝟚, 𝟞, 𝟡, and 𝟙𝟚. Although the SEL-PRIMARY-SUB-TREE has the bindings `{?x/Z}`, the instantiation of the logic variable `?x` to value `Z` is done by the sub-tree 𝟙. In other words, the SEL-PRIMARY-SUB-TREE has not established any binding. In step 12 of the function `is-valid`, the bindings `{?x/Z}` of the sub-tree 𝟙 is not modified because the logic variable `?x` is instantiated to the value `Z` by the logic goal `member(?x, [W, Z]).` The bindings of the sub-trees 𝟚 and 𝟙𝟚 are not changed because they are already empty. The bindings `{?x/Z}` of the sub-trees 𝟞 is changed to an empty list because the logic variable `?x` is bound to the value `Z` by the sub-tree 𝟙. Similarly, the bindings `{?x/Z}` of the sub-tree 𝟡 is changed to an empty list.

The SEL-SECONDARY-SUB-TREE has the bindings `{?x/W}`, but the instantiation of `?x` is performed by the sub-tree 𝟙𝟜. Thus, the bindings of the SEL-SECONDARY-SUB-TREE is changed in step 13 to an empty list (i.e. the logic variable `?x` is not instantiated). In step 14, since the third rule satisfies all requirements, a valid offspring can be created.

The offspring program is `(+ (/ Z 1.5) (- Z 3.8) (/ Z 1.5))` and its derivation tree is depicted in figure 5.6. It should be emphasized that the constituent from the secondary parent is changed from `(/ W 1.5)` to `(/ Z 1.5)` in the offspring. This must be modified because the logic variable `?x` in the sub-tree 𝟜𝟙 is instantiated to `Z` in the sub-tree 𝟛𝟡. The house-keeping procedure modifies the bindings of 𝟜𝟙 in order to achieve this effect. This example demonstrates the use of logic grammars to enforce contextual-dependency between different constituents of a program.

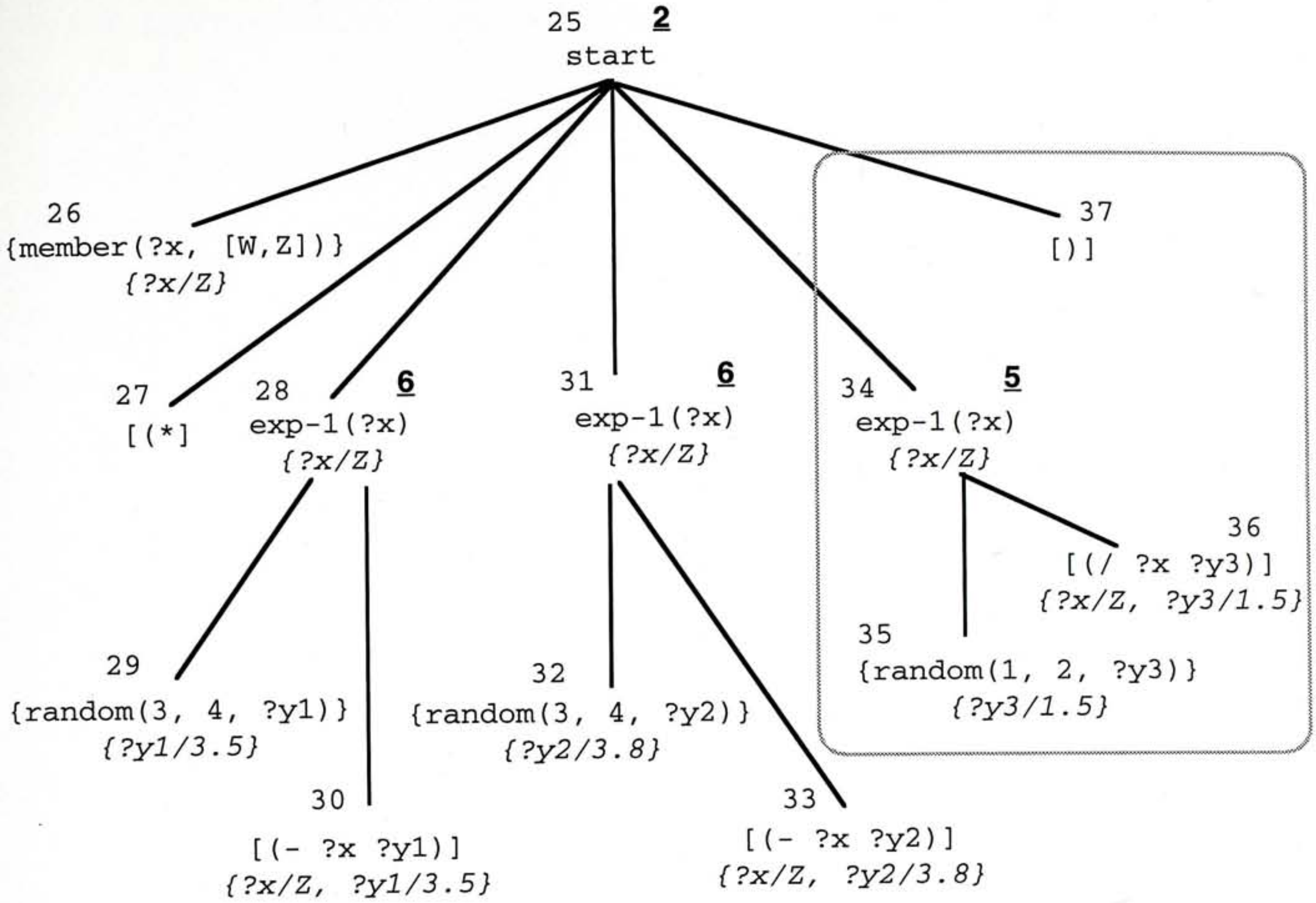


**Figure 5.5: A derivation tree of the offspring produced by performing crossover between the primary sub-tree 2 of the tree in figure 5.3 and the secondary sub-tree 15 of the tree in figure 5.4**

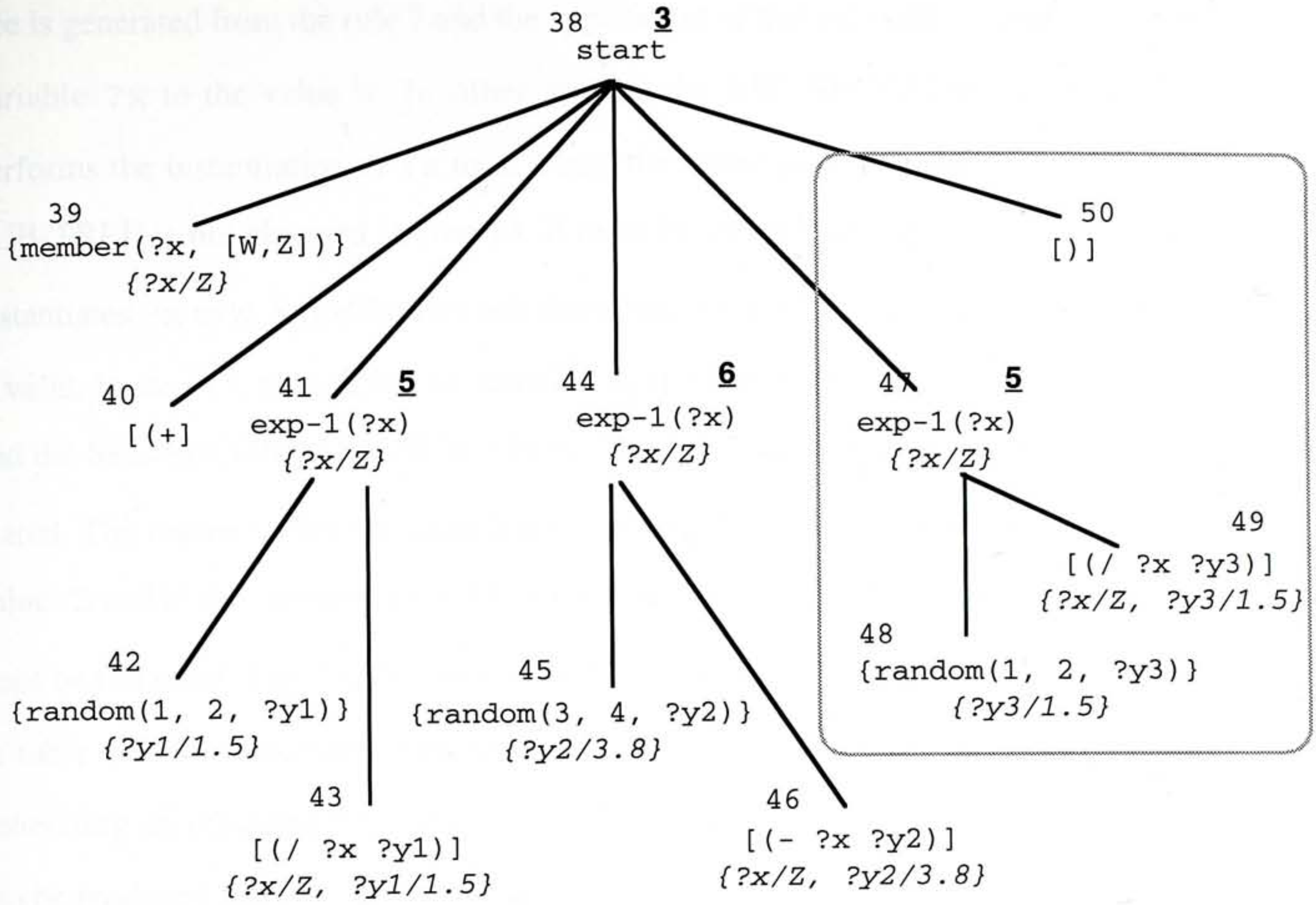


**Figure 5.6: A derivation tree of the offspring produced by performing crossover between the primary sub-tree 𝟛 of the tree in figure 5.3 and the secondary sub-tree 𝟙𝟞 of the tree in figure 5.4**

As a further example, the same primary and secondary parents are used. Assume that the SEL-PRIMARY-SUB-TREE 𝟞 is selected in step 3 of the crossover algorithm and the SEL-SECONDARY-SUB-TREE 𝟙𝟡 is chosen in step 7. The variable aSIBLINGS contains the sub-trees 𝟙, 𝟚, 𝟛, 𝟡, and 𝟙𝟚. In step 12 of the function `is-valid`, the bindings `{?x/Z}` of the sub-tree 𝟙 is not modified. The bindings of the sub-trees 𝟚 and 𝟙𝟚 are not modified because they are already empty. The bindings `{?x/Z}` of the sub-trees 𝟛 and 𝟡 are changed to empty lists because the logic variable `?x` is bound to the value `Z` by the sub-tree 𝟙.

The SEL-SECONDARY-SUB-TREE 𝟙𝟡 has the bindings {?x/W}. This sub-tree is generated from the rule 7 and the application of this rule will instantiate the logic variable ?x to the value W. In other words, the SEL-SECONDARY-SUB-TREE performs the instantiation of ?x to W. Thus, the bindings of the SEL-SECONDARY-SUB-TREE is not changed in step 13. It must be mentioned that the sub-tree 𝟙𝟜 also instantiates ?x to W. Since the two sub-trees bind ?x to the same value W, this situation is valid. In step 14, no rule can be satisfied by the sub-trees in the variable SIBLINGS and the SEL-SECONDARY-SUB-TREE. Thus, the two sub-trees 𝟞 and 𝟙𝟡 cannot be mated. The reason is that the same logic variable ?x must be instantiated to different values Z and W: the sub-tree 𝟙𝟡 requires the variable ?x to be instantiated to W while ?x must be instantiated to Z in the context of the primary parent. The function is-valid in table 5.4 can determine this situation and avoid the crossover algorithm from generating an offspring by exchanging the two sub-trees. Thus, only valid offspring can be produced and this operation can be achieved effectively.

In the following paragraphs, we estimate the time complexity of the crossover algorithm. Let the numbers of sub-trees in the primary and secondary derivation trees are respectively $N_p$ and $N_s$. The numbers of sub-trees in the global variables PRIMARY-SUB-TREES and SECONDARY-SUB-TREES are respectively $N_p'$ and $N_s'$. Assume that the depth of the primary derivation tree is $D_p$ (Depth starts from 0). Hence there are $D_p$ rule applications along the longest path from the root to the leaf node. Let R be the grammar rule having the largest number of symbols on its right hand side. Then S is the number of symbols on the right hand side of R.

Since the most time-consuming operation of the crossover algorithm is step 8 which calls the function is-valid. We concentrate on the time complexity of this step first. In the worst case, this step will calls is-valid for $N_p'*N_s'$ times. In each execution of the function is-valid, the purpose of steps 11 to 13 is to find the bindings established solely by the SEL-SECONDARY-SUB-TREE and the siblings of the SEL-PRIMARY-SUB-TREE. Since the total number of sub-trees to be examined must be equal to or smaller than S, the steps can be completed in $S*C_r$ time, where $C_r$ is the constant time to retrieve the bindings established solely by a particular sub-tree of the sub-trees being examined.

Step 14 is a loop that finds a grammar rule that can be satisfied. Suppose that the parent of the SEL-PRIMARY-SUB-TREE generates program fragments belonging to the category CAT. The loop examines all grammar rules for the category CAT. If there are $N_r$ rules for CAT, step 14 repeats for $N_r$ times.

In each iteration of step 14, the first three operations check whether the rule is satisfiable. These operations can be viewed as determining whether the SEL-SECONDARY-SUB-TREE and the sub-trees in the global variable SIBLINGS are unificable according to the rule (Mooney 1989). Since an efficient, linear time algorithm exist for unification (Paterson and Wegman 1978). These operations can be completed in O(S) time (Mooney 1989).

The last operation of step 14 applies the function `is-consistent` whose time complexity depends on the depth $D_c$ of the PRIMARY-CROSSOVER-POINT, where $D_c \leq D_p$. There are three cases to be considered. Firstly, $D_c$ cannot be equal to zero because the whole primary derivation tree cannot be crossed over with the SEL-SECONDARY-SUB-TREE. Secondly, if $D_c$ is equal to 1, the function `is-consistent` can be completed in constant time $C_1$ because step 15 will be executed. Lastly, if $D_c$ is greater than or equal to 2, the function `is-consistent` will recursively check the rules from the grandparent of the SEL-PRIMARY-SUB-TREE to the root of the primary derivation tree, to determine whether the rules are satisfied. As described previously, steps 16 and 17 can be completed in $S*C_r$ time and each rule can be checked in O(S) time. In the worst case, the recursive process iterates for $D_c$ times.

Hence the function `is-consistent` can be completed in $[(D_c - 1) * (O(S) + S * C_r) + C_l]$ time.

In summary, each execution of the function `is-valid` requires $T_{is\text{-}valid}$ time which is presented in follows:

$$T_{is\text{ - }valid} = S * C_r + N_r * [O(S) + ((D_c - 1) * (O(S) + S * C_r) + C_l)]$$

In the worst case, the depth $D_c$ of the PRIMARY-CROSSOVER-POINT is equal to $D_p$. Then the worst case time complexity of the function `is-valid` is:

$$T_{is\text{ - }valid} = S * C_r + N_r * [O(S) + ((D_p - 1) * (O(S) + S * C_r) + C_l)]$$

and the worst case time complexity of the crossover algorithm is:

$$T_{crossover} = N_p{}' * N_s{}' * T_{is\text{-}valid} + T_1 + T_2 + T_3 + T_4$$

where $T_1$ is the time used to perform steps 1 and 2, $T_2$ is the time employed to execute steps 3 and 4, $T_3$ is the execution time for steps 5 to 7, and $T_4$ is the time consumed by steps 9 and 10.

Obviously, $T_1$ depends on the sizes of the primary and secondary derivation trees, thus its complexity is $O(N_p + N_s)$. If the sub-trees in the variable PRIMARY-SUB-TREES are permuted randomly using an $O(N_p{}')$ algorithm (Cormen et al. 1990) before executing steps 3 and 4, these steps can be completed in $T_2 = O(N_p{}')$ time. Similarly, steps 5, 6, and 7 can be completed in $T_3 = O(N_p{}' * N_s{}')$ time. $T_4$ depends on the sizes of the primary and secondary derivation trees, thus its complexity is $O(N_p + N_s)$.

Assume that the first term of the above equation is much larger than the other terms, then the worst case time complexity is approximated by the following equation:

$$T_{crossover} \cong O(N_p{}' * N_s{}' * D_p * S * N_r).$$

If the primary derivation tree is a complete m-ary tree, then $\frac{m^{(D_p+1)}-1}{m-1} = N_p$. In other words, $D_p$ is of the order of $\log_m(N_p)$. Furthermore, S and $N_r$ are fixed for a given grammar. Thus, the worst case time complexity of the crossover algorithm is:

$$T_{crossover} \cong O(N_p{'}*N_s{'}*\log_m(N_p)).$$

Since the computation time consumed by performing crossover is insignificant when compare with the time used in evaluating the fitness of each program in the population. The issue of computational complexities of various crossover algorithms has not been addressed by other researchers in the field of Genetic Programming. In fact, it is easy to calculate that the worst case time complexity of the structure-preserving crossover algorithm (table 2.5) of ADF (Koza 1994) is $O(N_{p1}*N_{p2})$, where $N_{p1}$ and $N_{p2}$ are respectively the sizes of the parental parse trees. Similarly, the crossover algorithm of STGP (Montana 1993) has the same complexity. Although the crossover algorithm of LOGENPRO is slightly slower than other algorithms by $O(\log_m(N_p))$, it is much more general and powerful than other algorithms.

## 5.4. Mutation of programs

The mutation operation in LOGENPRO introduces random modifications to programs in the population. Mutation is asexual and operates on only one program each time. A program in the population is selected as the parental program. The selection is based on various methods such as fitness proportionate and tournament selections. The algorithm in table 5.6 is used to produce an offspring program by mutation.

```
Input:
   P:        The derivation tree of the parental program

Output:
   Return a new derivation tree if a valid offspring can be obtained
   by performing mutation, otherwise return false.

Function mutation(P)
{
     1.     Find all sub-trees of the derivation tree P of the
            parental program and store them into a global variable
            SUB-TREES, excluding all frozen sub-trees, logic goals,
            and terminal symbols
     2.     If SUB-TREES is not empty, select randomly a sub-tree
            from the SUB-TREES using a uniform distribution.
            Otherwise, terminate the algorithm without generating any
            offspring.
     3.     Designate the sub-tree selected as MUTATED-SUB-TREE. The
            root of the MUTATED-SUB-TREE is called the MUTATE-POINT.
            Remove the MUTATED-SUB-TREE from the variable SUB-TREES.
            The MUTATED-SUB-TREE must be generated from a non-
            terminal symbol of the grammar. Designate this non-
            terminal symbol as NON-TERMINAL. The NON-TERMINAL may
            have a list of arguments called ARGS.
     4.     For each argument in the ARGS, if it contains some logic
            variables, determine whether these variables are
            instantiated by other constituent of the derivation tree.
            If they are, bind the instantiated value to the variable.
            Otherwise, the variable is unbounded. Store the modified
            bindings to a global variable NEW-BINDINGS.
     5.     Create a new non-terminal symbol NEW-NON-TERMINAL from
            the NON-TERMINAL and the bindings in the variable NEW-
            BINDINGS.
     6.     Try to generate a new derivation tree NEW-SUB-TREE from
            the NEW-NON-TERMINAL using the deduction mechanism
            provided by LOGENPRO.
     7.     If a new derivation tree can be successfully created, the
            offspring is obtained by deleting the MUTATED-SUB-TREE
            from a copy of the parental derivation tree P and then
            impregnating the NEW-SUB-TREE at the MUTATE-POINT.
            Otherwise, go to step 3.
}
```

**Table 5.6: The mutation algorithm**

For example, assume that the program being mutated is `(+ (- Z 3.5) (- Z 3.8) (/ Z 1.5))` and the corresponding derivation tree is depicted in figure 5.3. In step 1 of the mutation algorithm, the global variable SUB-TREES contains the sub-trees 𝟘, 𝟛, and 𝟞. The frozen sub-trees 𝟡, 𝟙𝟘, 𝟙𝟙, and 𝟙𝟚 are excluded. The sub-trees 𝟙, 𝟜, and 𝟟 are also excluded because they contain logic goals of the grammar and thus should not be modified by genetic operations. The sub-trees 2, 5, and 8 containing terminal symbols are eliminated for two reasons. First, the mutation algorithm is significantly simplified if terminal symbol need not be modified. Second, the effect of mutating terminal symbols can be emulated by the crossover operation. Recalling the example described in the previous section, the primary sub-tree 2 are crossed with the secondary sub-tree 15 to generate the offspring (* (- Z 3.5) (- Z 3.8) (/ Z 1.5)). This offspring can be seen as the result of mutating the terminal symbol [(+] to the [(*].

In step 2, a sub-tree in the variable SUB-TREES is selected randomly using a uniform distribution if the SUB-TREES is not empty. Otherwise, the mutation algorithm terminates without generating any modified program because no valid mutation can be found. In normal situation, this should not occur because it is almost always possible to select the whole derivation tree as the one to be mutated. The whole tree cannot be chosen only if it is frozen. The effect of mutating the whole tree, the sub-tree 0 in this example, is equivalent to generating a new program from scratch. A new program can be created successfully if the language specified by the grammar contains at least one program (this must be true for a grammar to be useful) and enough computational resources such as computer memory are available. Thus, the algorithm will fail to find a mutation only if the whole derivation tree is frozen or not enough computational resources are available.

Assume that the sub-tree 3 is selected as the MUTATED-SUB-TREE in step 2. In the next step, the sub-tree 3 is removed from the variable SUB-TREES. The NON-TERMINAL and the ARGS are exp-1(?x) and {?x} respectively. Since the logic variable ?x is instantiated to Z in the sub-tree 1 by the logic goal member(?x, [W, Z]), the bindings {?x/Z} are stored into the variable NEW-BINDINGS in step 4.

In step 5, the new non-terminal NEW-NON-TERMINAL `exp-1(Z)` is created. Using this mechanism, contextual-dependent information can be transmitted between different parts of a program. In step 6, a new derivation tree for the S-expression `(/ Z 1.9)` can be obtained from the non-terminal symbol `exp-1(Z)` using the fifth rule of the grammar. This derivation tree is displayed in figure 5.7.

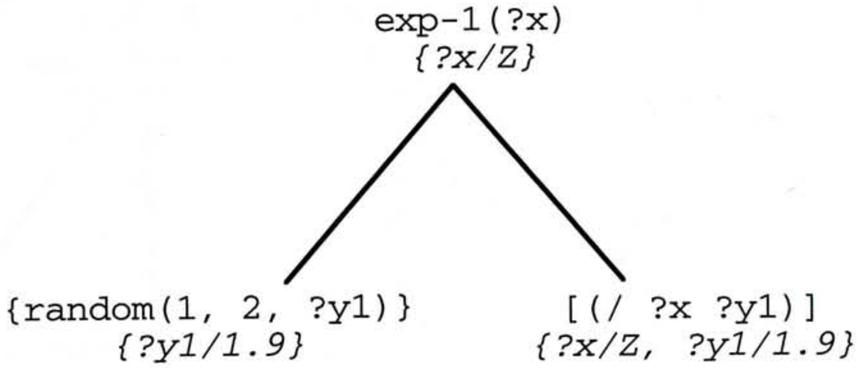


**Figure 5.7: A derivation tree generated from the non-terminal exp-1(Z)**

Since the NEW-SUB-TREE can be found, a new offspring is obtained by duplicating the genetic materials of its parental derivation tree, followed by deleting the MUTATED-SUB-TREE from the duplication, and then pasting the NEW-SUB-TREE at the MUTATE-POINT. The derivation tree of the offspring `(+ (/ Z 1.9) (- Z 3.8) (/ Z 1.5))` can be found in figure 5.8.

LOGENPRO has an efficient implementation of the mutation algorithm. Moreover, an inference engine has been developed for deducing derivation trees (or programs) from a logic grammar given. Thus, only valid mutations can be performed and this operation can be achieved effectively and efficiently.

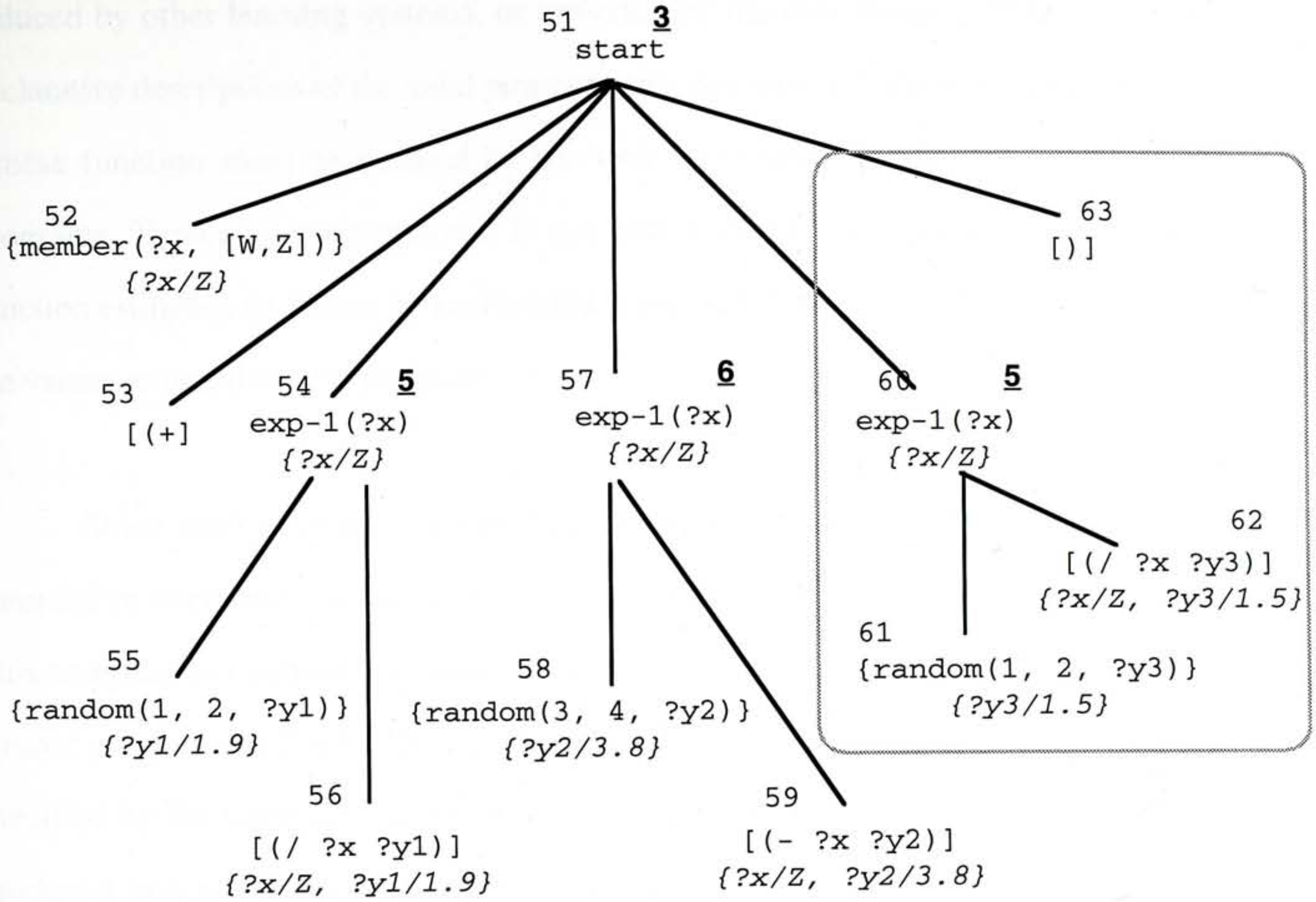


**Figure 5.8: A derivation tree of the offspring produced by performing mutation of the tree in figure 5.3 at the sub-tree 3**

## 5.5. The evolution process of LOGENPRO

The problem of inducing S-expressions or logic programs can be formulated as a search for a highly fit program in the space of all possible programs (Mitchell 1982). In GP, this space is determined by the syntax of S-expressions in Lisp and the sets of terminals and functions. The search space of ILP is determined by the syntax of logic program and the background knowledge. Thus, the search space is fixed once these elements are decided. On the other hand, the search space can be specified declaratively under the framework proposed because the space is determined by the logic grammar given.

LOGENPRO starts with an initial population of programs generated randomly, induced by other learning systems, or provided by the user. Logic grammars provide declarative descriptions of the valid programs that can appear in the initial population. A fitness function must be defined by the user to evaluate the fitness values of the programs. Typically, each program is run over a set of fitness cases and the fitness function estimates its fitness by performing some statistical operations (e.g. average) to the values returned by this program.

Since each program generated in the evolution process must be executed. A compiler or interpreter for the corresponding programming language must be available. This compiler or interpreter is called by the fitness function to compile or interpret the created programs. LOGENPRO can guarantee only that valid programs in the language specified by the logic grammar will be generated. However, it cannot ensure that the produced programs can be successfully compiled or interpreted if the appropriate compiler/interpreter is not provided by the user. Thus, the user must be very careful in designing the logic grammar and the fitness function.

The evolution process of LOGENPRO is similar to that of GLPS described in chapter 4. A high-level algorithm of LOGENPRO is presented in table 5.7. The main difference between the two systems is in their genetic operators. The crossover operator of LOGENPRO generates at most one offspring from two parental programs selected. The mutation operator creates a modified offspring program from a parental program selected. Moreover, a particular logic grammar is used to restrict the offspring programs that can be produced by these genetic operators.

```
Input:
   Grammar: It is a logic grammar that specifies the search space of
            programs.
   t        The termination function.
   f        The fitness function.

Output:
   A logic program induced by LOGENPRO.

Function LOGENPRO(Grammar, t, f)
{
   •  Translate the Grammar to a logic program.

   •  generation := 0.

   •  Initialize a population Pop(generation) of programs. They are
      generated by issuing the query ?-start(?Tree, ?S , []),
      provided from the user, or generated by other learning systems.
      If a program, Prog,  is provide by the user or generated by
      other learning systems, the program is translated to a
      derivation tree using the query ?-start(?Tree, ?P, []) where ?P
      contains the program Prog.

   •  Execute each program in the Pop(generation) and assign it a
      fitness value according to the fitness function f.

   •  While the termination function t is not satisfied do
         •  Create a new population Pop(generation+1) of programs by
            employing the reproduction, the crossover and the
            mutation. The operations are applied to programs chosen
            by either the fitness proportionate or tournament
            selections.
         •  Evaluate the fitness of each individual in the next
            population Pop(generation+1)
         •  generation := generation + 1.
   •  Return the best program found in any generation of the run.
}
```

**Table 5.7: A high-level algorithm of LOGENPRO**

## 5.6. Discussion

We have proposed a framework for combining GP and ILP. This framework is based on a formalism of logic grammars. The formalism can represent context-sensitive information and domain-dependent knowledge. It is also very flexible and programs in various programming languages such as Lisp, Prolog, Fuzzy Prolog, and C can be induced.

Since the framework is very flexible, different representations employed in other inductive learning systems can be specified easily. It facilitates the integration of LOGENPRO with other learning systems. One approach is to incorporate the learning techniques of other systems into LOGENPRO. These techniques include information guided hill-climbing (Quinlan 1990; 1991), explanation-based generalization (DeJong and Mooney 1986, Mitchell et al. 1986, Ellman 1989), explanation-based specialization (Minton 1989) and inverse resolution (Muggleton 1992). LOGENPRO can also invoke these systems as front-ends to generates the initial population. The advantage is that they can quickly find important and meaningful components (genetic materials) and embody these components into the initial population. The following chapters will illustrate some of these points clearly.

# Chapter 6
# Applications of LOGENPRO

LOGENPRO is a flexible and general program induction system. In the first section, the method of emulating Genetic Programming (GP) using LOGENPRO is illustrated. In section 6.2, it is demonstrated that the learning problems solved by GLPS can also be handled by LOGENPRO. In the last section, we illustrate that LOGENPRO can induce programs in imperative programming languages such as C. Experimental results by LOGENPRO are presented and compared with similar methods where appropriate.

## 6.1. Learning functional programs

It seems that the framework proposed in the previous chapter is rather complicated but powerful. Consequently, the question of whether this framework can be applied easily arises. In the first sub-section, we show that this framework can emulate GP (Koza 1992; 1994) easily in learning S-expressions. A template is provided to facilitate the application of the framework. It must be emphasized that the example used in the first sub-section is deliberately constructed as simple as possible to illustrate the point. More realistic applications can be found in the following sub-sections.

### 6.1.1. Learning S-expressions using LOGENPRO

A logic grammar template for learning S-expressions using the framework is depicted in table 6.1. To apply the template for a particular problem, various sets of terminals and primitive functions will substitute for the identifiers in italics.

```
10:   start        ->    function.
11:   s-exp        ->    term.
12:   s-exp        ->    function.

13a:  function     ->    function-0.
13b:  function     ->    function-1.
13c:  function     ->    function-2.
  ...
  ...
13n:  function     ->    function-n.

14a:  function-0   ->    [(], op-0, [)].
14b:  function-1   ->    [(], op-1, s-exp, [)].
14c:  function-2   ->    [(], op-2, s-exp, s-exp, [)].
  ...
  ...
14n:  function-n   ->    [(], op-n, s-exp, ..., s-exp,[)].

15:   term         ->    { member(?w, <TERMINAL SET>) }, [?w].

16a:  op-0         ->    { member(?w, <FUNCTION SET-0>) }, [?w].
16b:  op-1         ->    { member(?w, <FUNCTION SET-1>) }, [?w].
16c:  op-2         ->    { member(?w, <FUNCTION SET-2>) }, [?w].
  ...
  ...
16n:  op-n         ->    { member(?w, <FUNCTION SET-n>) }, [?w].
```

**Table 6.1: A template for learning S-expressions using the LOGENPRO**

Consider the problem of learning S-expressions such as `( - (* Z X) (+ Y Z))`. Using the terminology of GP, the set of primitive functions for this problem contains arithmetic operators `+`, `-`, and `*`. Each of them takes two arguments as inputs. The terminal set is `{X, Y, Z}`. The terminals can be treated as input arguments of the S-expression being learned.

It is observed that a S-expression is either a terminal or a function invocation. Thus a S-expression can be specified by the grammar rules 11 and 12 of the template in table 6.1. A function call consists of a list of elements enclosed by a pair of parentheses. The first element of the list is the name of the function and the other elements are arguments of the function. These arguments are also S-expressions. Since the primitives of a problem may have different numbers of arguments, there are a variety of function invocations. This fact can be specified by the grammar rules 13a, 13b, ..., 13n, and 14a, 14b, ..., 14n.

Since an S-expression containing only a terminal is usually excluded from consideration as a solution. This fact is declared by the grammar rule 10 which specifies that the target solution must be a function invocation. The non-terminal symbol `term` specifies the terminal set of the problem domain. For the problem studied in this sub-section, the terminal set is represented as:

```
term ->    { member(?w, [X, Y, Z]) }, [?w].
```

where the goal `member(?w, [X, Y, Z])` instantiates the logic variable `?x` to one of the value in the list `[X, Y, Z]`. This grammar rule is obtained from rule 15 in the template by replacing the identifier *<TERMINAL SET>* with `[X, Y, Z]`.

The non-terminal symbols `op-0`, `op-1`, ..., `op-n` in the template specify primitive functions with different numbers of arguments. They represent the primitive function set of the problem domain. For the above problem, all primitives have two arguments, thus only `op-2` will be used. It is represented by the following rule:

```
op-2 ->    { member(?w, [+, -, *]) }, [?w].
```

This rule is obtained from the grammar rule 16c in the template by replacing the identifier *<FUNCTION SET-2>* with `[+, -, *]`. Other non-terminal symbols such as `op-0`, `op-1`, `op-3`, ..., `op-n` will be used if the problem domain requires primitives with the corresponding numbers of arguments. In summary, the logic grammar for the example is:

```
    start            ->   function.
    s-exp            ->   term.
    s-exp            ->   function.
    function         ->   function-2.
    function-2       ->   [(], op-2, s-exp, s-exp, [)].
    term             ->   { member(?w, [X, Y, Z]) }, [?w].
    op-2             ->   { member(?w, [+, -, *]) }, [?w].
```

### 6.1.2. The DOT PRODUCT problem

In this sub-section, we describe how to use LOGENPRO to emulate Koza's GP (Koza, 1992). Koza's GP has a limitation that all the variables, constants, arguments for functions, and values returned from functions must be of the same data type. This limitation leads to difficulties in inducing even some rather simple and straightforward functional programs. For example, one of these programs calculates the dot product of two given numeric vectors of the same size. Let X and Y be the two input vectors, then the dot product is obtained by the following S-expression:

```
(apply (function +) (mapcar (function *) X Y))
```

Let us use this example for illustrative comparisons below. To induce a functional program using LOGENPRO, we have to determine the logic grammar, the fitness cases, the fitness function, and the termination criterion. The logic grammar for learning functional programs is given in table 6.2. In this grammar, we employ the argument of the grammar symbol `s-expr` to designate the data type of the result returned by the S-expression generated from the grammar symbol. For example,

```
(mapcar (function +) X
     (mapcar (function *) X Y))
```

is generated from the grammar symbol `s-expr([list, number, n])` because it returns a numeric vector of size n. Similarly, the symbol `s-expr(number)` can produce `(apply (function *) X)` that returns a number.

```
start                          -> s-expr(number).
s-expr([list, number, ?n])     -> [ (mapcar (function ], op2,
                                  [ ) ] ,
                                  s-expr([list, number, ?n]),
                                  s-expr([list, number, ?n]),[ ) ].
s-expr([list, number, ?n])     -> [ (mapcar (function ], op1,
                                  [ ) ] ,
                                  s-expr([list, number, ?n]),[ ) ].
s-expr([list, number, ?n])     -> term([list, number, ?n]).
s-expr(number)                 -> term(number).
s-expr(number)                 -> [ (apply (function ], op2,
                                  [ ) ] ,
                                  s-expr([list, number, ?n]),[ ) ].
s-expr(number)                 -> [ ( ], op2, s-expr(number),
                                  s-expr(number), [ ) ].
s-expr(number)                 -> [ ( ], op1, s-expr(number), [ ) ].
op2                            -> [ + ].
op2                            -> [ - ].
op2                            -> [ * ].
op2                            -> [ % ].
op1                            -> [ protected-log ].
term( [list, number, n] )      -> [ X ].
term( [list, number, n] )      -> [ Y ].
term( number )                 -> { random(-10, 10, ?a) },   [ ?a ].
```

**Table 6.2: The logic grammar for the DOT PRODUCT problem**

The terminal symbols +, -, and * represent functions for ordinary addition, subtraction, and multiplication respectively. The symbol % represents function that normally returns the quotient. However, if division by zero is attempted, the function returns 1.0. The symbol `protected-log` is a function that calculates the logarithm of the input argument x if x is larger than zero, otherwise it returns 1.0. The logic goal `random(-10, 10, ?a)` generates a random floating point number between -10 and 10 and instantiates ?a to the random number generated

Ten random fitness cases are used for training. Each case is a 3-tuples ‹$\mathbf{X}_i$, $\mathbf{Y}_i$, $Z_i$›, where $1 \leq i \leq 10$, $\mathbf{X}_i$ and $\mathbf{Y}_i$ are vectors of size 3, and $Z_i$ is the corresponding dot product. The fitness function calculates the sum, taken over the ten fitness cases, of the absolute values of the difference between $Z_i$ and the value returned by the S-expression for $\mathbf{X}_i$ and $\mathbf{Y}_i$. A fitness case is said to be covered by a S-expression if the value returned by it is within 0.01 of the desired value. A S-expression that covers all training cases is further evaluated on a testing set containing 1000 random fitness cases. LOGENPRO will stop if the maximum number of generations of 100 is reached or a S-expression that covers all testing fitness cases is found.

For Koza's GP framework, the terminal set **T** is `{X, Y, ℝ}` where ℝ is the ephemeral random floating point constant. ℝ takes on a different random floating point value between -10.0 and 10.0 whenever it appears in an individual program in the initial population. The function set **F** is `{protected+, protected-, protected*, protected%, protected-log, vector+, vector-, vector*, vector%, vector-log, apply+, apply-, apply*, apply%}`, taking 2, 2, 2, 2, 1, 2, 2, 2, 2, 1, 1, 1, 1, and 1 arguments respectively.

The primitive functions `protected+`, `protected-`, and `protected*` respectively perform addition, subtraction, and multiplication if the two input arguments `X` and `Y` are both numbers. Otherwise, they return 0. The function `protected%` returns the quotient. However, if division by zero is attempted or the two arguments are not numbers, `protected%` returns 1.0. The function `protected-log` finds the logarithm of the argument `X` if `X` is a number larger than zero. Otherwise, `protected-log` returns 1.0.

The functions `vector+`, `vector-`, `vector*`, and `vector%` respectively perform vector addition, subtract, multiplication, and division if the two input arguments `X` and `Y` are numeric vectors with the same size, otherwise they return zero. If the input argument `X` is a numeric vector, the primitive function `vector-log` performs the following S-expression:

```
    (mapcar (function protected-log) X),
```

otherwise it returns zero. The functions `apply+`, `apply-`, `apply*`, and `apply%` respectively perform the following S-expressions if the input argument `X` is a numeric vector:

```
(apply (function protected+) X),
(apply (function protected-) X),
(apply (function protected*) X) and
(apply (function protected%) X),
```

otherwise they return zero.

The fitness cases, the fitness function, and the termination criterion are the same as those used by LOGENPRO. Three experiments are performed. The first one evaluates the performance of LOGENPRO using a population of 100 programs. The other two experiments evaluate the performance of Koza's GP using respectively populations of 100 and 1000 programs. In each experiment, over sixty trials are attempted and the results are summarized in figure 6.1. The figure delineates the best standardized fitness values for increasing generations for the three experiments. From the curves in figures 6.1, LOGENPRO has superior performance than that of GP.

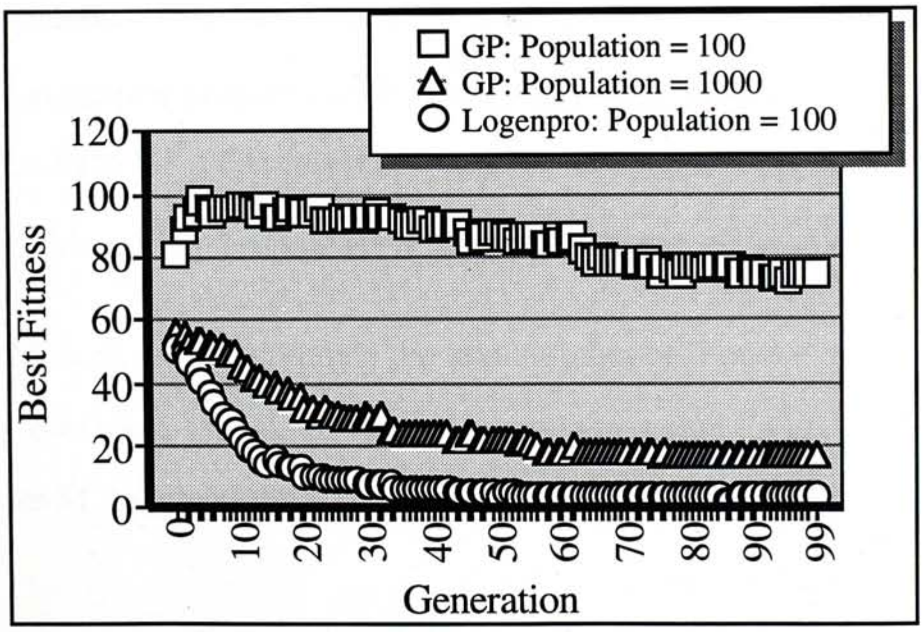


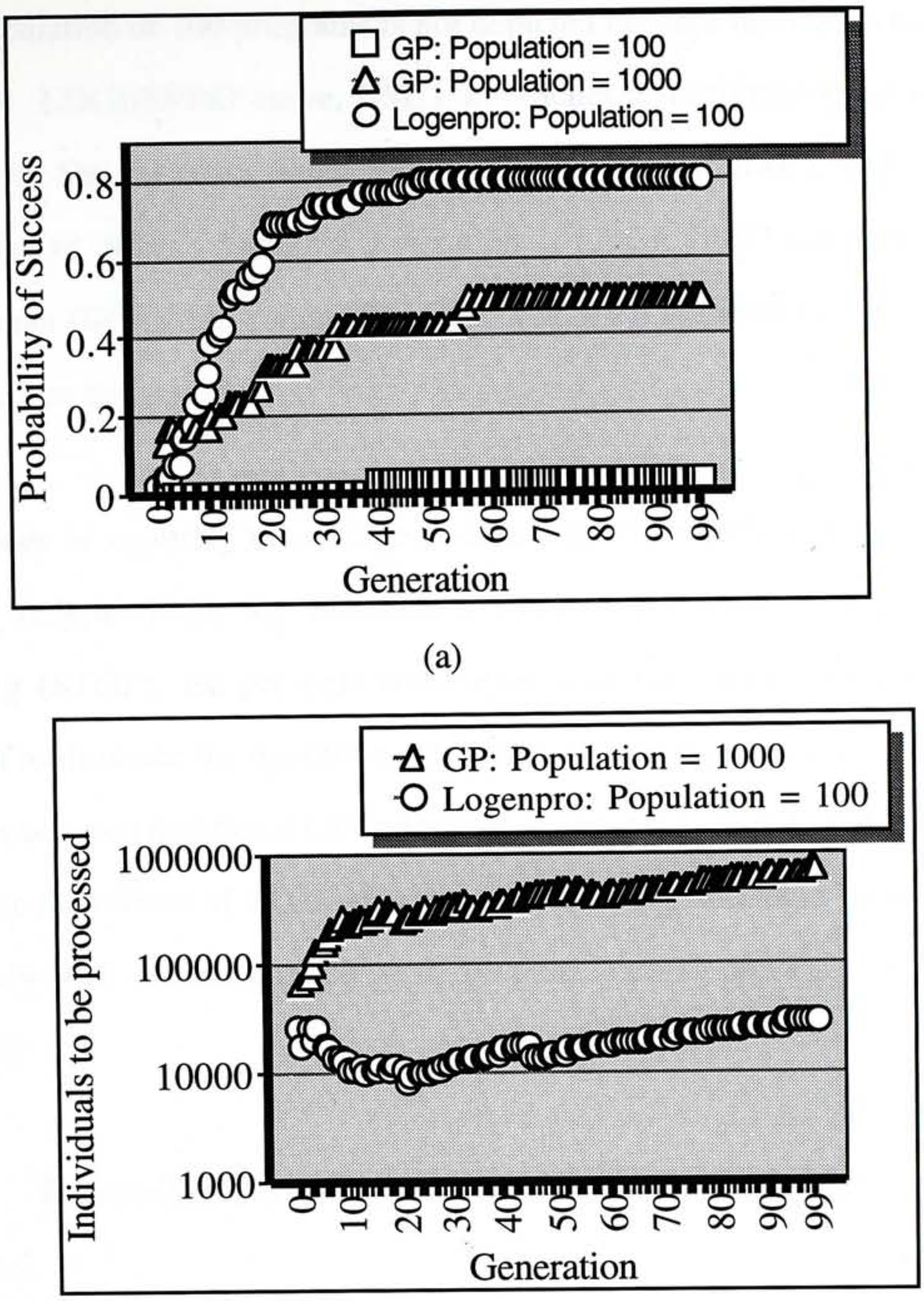
**Figure 6.1: The fitness curves showing the best fitness values for the DOT PRODUCT problem**

Statistical measure are also collected to estimate the computational effort E required to yield satisfactory programs to the problem with a high probability (Koza 1992). We estimate E empirically from a series of runs. Each run is made using a particular fixed population size M and a particular fixed maximum number of generation G.

Since all evolutionary algorithm are non-deterministic, not all runs are successful in producing satisfactory programs by generation G. Consequently, a probabilistic method is used to compute the number of fitness evaluations required. For non-trivial problem, fitness evaluations consume a significant fraction of the computational resources required. Thus, the number of fitness evaluations is a reasonable measure of computational effort consumed.

We can empirically estimate the probability Y(M, i) that a run yields, for the first time, at least one satisfactory program using a population size M on generation i. P(M, i) is then computed to estimate the cumulative probability that a particular run produces satisfactory programs by generation i. Thus, the probability of generating satisfactory programs by generation i at least once in R independent runs is $1-[1-P(M,i)]^{R}$. If we want to find satisfactory programs with a certain specified probability z, then it must be that $z \leq 1-[1-P(M,i)]^{R}$. The minimum number of independent runs R required by generation i with a high probability z is $R = R(M,i,z) \geq \left\lceil \frac{\log(1-z)}{\log(1-P(M,i))} \right\rceil$. After obtaining R(M, i, z), we can compute the total number of fitness evaluations I(M, i, z) that is required to yield satisfactory programs by generation i with probability z for a population size M. In other words, $I(M,i,z) = M*(i+1)*R(M,i,z)$.

The computational effort E required for a particular problem with a pre-specified probability z is the minimal value of I(M, i, z), over all the generations i between 0 and G.



(a)

(b)

**Figure 6.2: The performance curves showing (a) cumulative probability of success P(M, i) and (b) I(M, i, z) for the DOT PRODUCT problem**

The curves in figure 6.2(a) show the experimentally observed cumulative probability of success P(M, i) of solving the problem by generation i using a population of M programs. The curves in figure 6.2(b) show the number of programs I(M, i, z) that must be processed to produce a solution by generation i with a probability z. Throughout this chapter, the probability z is set to 0.99. The curve for GP with a population of 100 programs is not depicted because the values is extremely large. For the LOGENPRO curve, I(M, i, z) reaches a minimum value of 8800 at generation 21. On the other hand, the minimum value of I(M, i, z) for GP with population size of 1000 is 66000 at generation 1. LOGENPRO can find a solution much faster than GP and the computation (i.e. I(M, i, z)) required by LOGENPRO is much smaller than that of GP.

The idea of applying knowledge of data type to accelerate learning has been investigated independently by Montana (1993) in his Strongly Typed Genetic Programming (STGP). He presents three examples involving vector and matrix manipulation to illustrate the operations of STGP. However, he has not compared the performances between traditional GP and STGP. One advantage of LOGENPRO is that it can emulate the effects of STGP effortlessly. Moreover, the logic grammar can be used to specify other domain knowledge to drive the learning process more effectively and efficiently.

### 6.1.2. Learning sub-functions using explicit knowledge

Automatic discovery of problem representation primitives is certainly one of the most challenging research areas in Genetic Programming. Automatically Defined Functions (ADF) is one of the approaches that have been proposed to acquire problem representation primitives automatically (Koza 1992; 1994). In ADF, each program in the population contains an expression, called the result-producing branch, and definitions of one or more sub-functions which may be invoked by the result-producing branch. The result-producing branch is evaluated to produce the fitness of the program. A constrained syntactic structure and some special genetic operators are required for the evolution of the programs. To employ the approach, the user must provide explicit knowledge about the number of automatically defined sub-functions, the number of arguments of each sub-functions, and the allowable terminal and function sets for each sub-function. In this sub-section, we demonstrate how to use LOGENPRO to emulate Koza's ADF approach. Moreover, other knowledge such as argument types can also be applied to speed up the learning task.

In this experiment, LOGENPRO is expected to learn a sub-function that calculates dot product and employ this sub-function in the main program. In other words, it is expected to induce the following S-expression:

```
(progn
   (defun ADF0 (arg0 arg1)
      (apply (function +) (mapcar (function *) arg0 arg1)))
   (+ (ADF0 X Y) (ADF0 Y Z)))
```

The logic grammar for this type of problem is depicted in table 6.3. We employ the argument of the grammar symbol to designate the data type of the result returned by the S-expression generated from the grammar symbol. The terminal symbols +, -, and * represent functions that perform ordinary addition, subtraction, and multiplication respectively. Ten random fitness cases are used for training. Each case is a 4-tuples ‹$\mathbf{X}_i$, $\mathbf{Y}_i$, $\mathbf{Z}_i$, $R_i$›, where $1 \leq i \leq 10$, $\mathbf{X}_i$, $\mathbf{Y}_i$ and $\mathbf{Z}_i$ are vectors of size 3, and $R_i$ is the corresponding desired result. The fitness function calculates the sum, taken over the ten fitness cases, of the absolute values of the difference between $R_i$ and the value returned by the S-expression for $\mathbf{X}_i$, $\mathbf{Y}_i$, and $\mathbf{Z}_i$.. A fitness case is said to be covered by a S-expression if the value returned by it is within 0.01 of the desired value. A S-expression that covers all training cases is further evaluated on a testing set containing 1000 random fitness cases. LOGENPRO will stop if the maximum number of generations of 50 is reached or a S-expression that covers all testing fitness cases is found.

```
start                          -> [(progn  (defun  ADF0  ],
                                  [(arg0  arg1)],
                                  s-expr2(number), [)],
                                  s-expr(number), [)].
s-expr([list, number, ?n])     -> [  (mapcar  (function  ],  op2,
                                  [ ) ] ,
                                  s-expr([list, number, ?n]),
                                  s-expr([list, number, ?n]),[ ) ].
s-expr([list, number, ?n])     -> term([list, number, ?n]).
s-expr(number)                 -> [  (apply  (function  ],  op2,
                                  [ ) ] ,
                                  s-expr([list, number, ?n]),[ ) ].
s-expr(number)                 -> [ ( ], op2, s-expr(number),
                                  s-expr(number), [ ) ].
s-expr(number)                 -> [  (ADF0  ],
                                  s-expr([list, number, ?n]),
                                  s-expr([list, number, ?n]), [ ) ].
term([list, number, n])        -> [ X ].
term([list, number, n])        -> [ Y ].
term([list, number, n])        -> [ Z ].
s-expr2([list, number, ?n])    -> [  (mapcar  (function  ],  op2,
                                  [ ) ] ,
                                  s-expr2([list, number, ?n]),
                                  s-expr2([list, number, ?n]),[ ) ].
s-expr2([list, number, ?n])    -> term2([list, number, ?n]).
s-expr2(number)                -> [  (apply  (function  ],  op2,
                                  [ ) ] ,
                                  s-expr2([list, number, ?n]),[ ) ].
s-expr2(number)                -> [ ( ], op2, s-expr2(number),
                                  s-expr2(number), [ ) ].
term2([list, number, n])       -> [ arg0 ].
term2([list, number, n])       -> [ arg1 ].
op2                            -> [ + ].
op2                            -> [ - ].
op2                            -> [ * ].
```

**Table 6.3: The logic grammar for the sub-function problem**

For Koza's ADF framework, the terminal set $\mathbf{T}_0$ for the automatically defined function (ADF0) is {arg0, arg1} and the function set $\mathbf{F}_0$ is {protected+, protected-, protected*, vector+, vector-, vector*, apply+, apply-, apply*}, taking 2, 2, 2, 2, 2, 2, 1, 1, and 1 arguments respectively. The terminal set $\mathbf{T}_r$ for the result producing branch is {X, Y, Z} and the function set $\mathbf{F}_r$ is {protected+, protected-, protected*, vector+, vector-, vector*, apply+, apply-, apply*, ADF0}, taking 2, 2, 2, 2, 2, 2, 1, 1, 1, and 2 arguments respectively. The primitive functions have already been defined in the previous sub-section. The fitness cases, the fitness function, and the termination criterion are the same as the ones used by LOGENPRO. We evaluate the performances of LOGENPRO and Koza's ADF using populations of 100 and 1000 programs respectively.

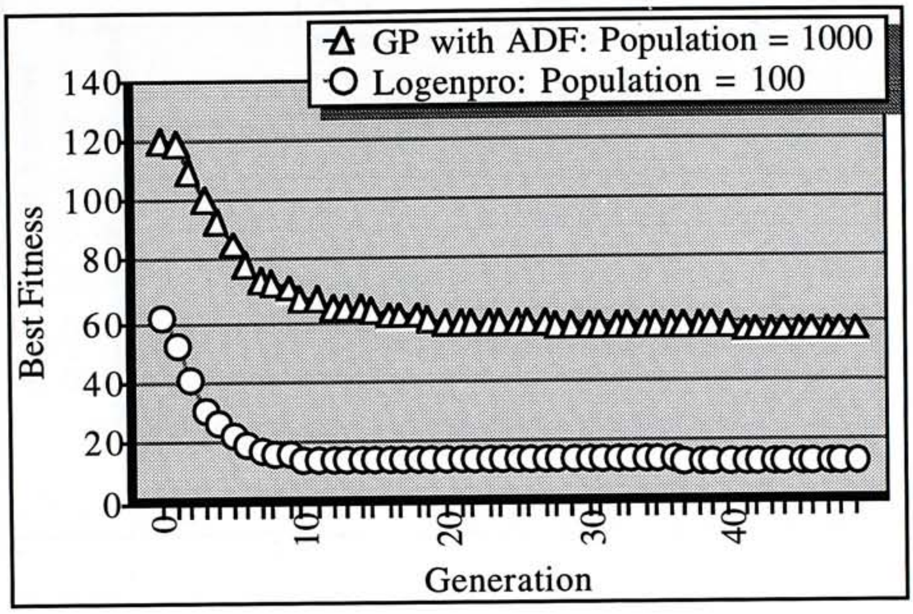
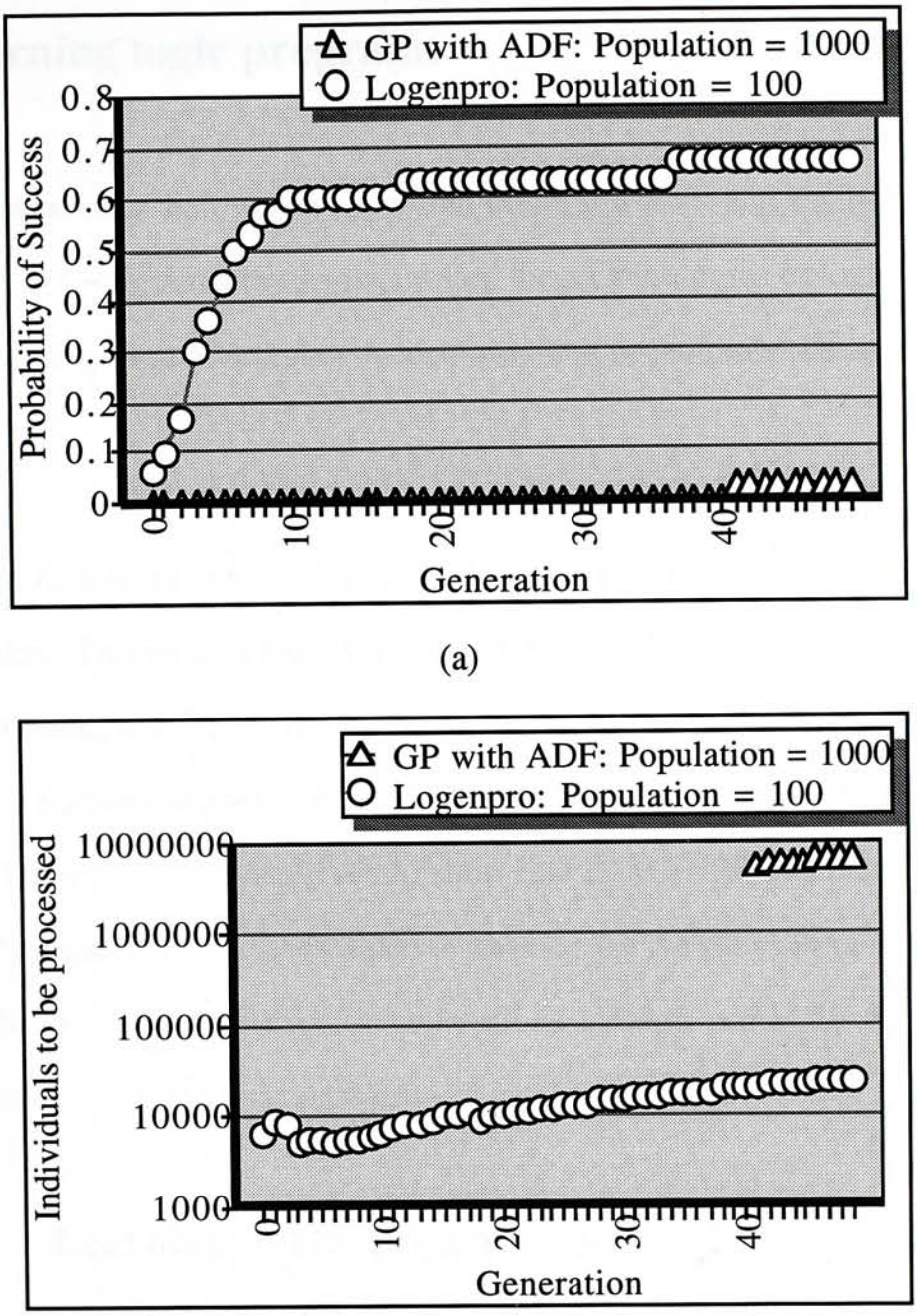


**Figure 6.3: The fitness curves showing the best fitness values for the sub-function problem**

Thirty trials are attempted and the results are summarized in figures 6.3 and 6.4. Figure 6.3 shows, by generation, the fitness of the best program in a population. These curves are found by averaging the results obtained in thirty different runs using various random number seeds and fitness cases. From these curves, LOGENPRO has superior performance than that of ADF. The curves in figure 6.4(a) show the experimentally observed cumulative probability of success P(M, i) of solving the problem by generation i using a population of M programs. The curves in figure 6.4(b) show the number of programs I(M, i, z) that must be processed to produce a solution by generation i with a probability z of 0.99. The curve for LOGENPRO reaches a minimum value of 4900 at generation 6. On the other hand, the minimum value of I(M, i, z) for ADF is 5712000 at generation 41. This experiment clearly shows the advantage of LOGENPRO. By employing various knowledge about the problem being solved, LOGENPRO can find a solution much faster than ADF and the computation (i.e. I(M, i, z)) required by LOGENPRO is much smaller than that of ADF.



(a)

(b)

**Figure 6.4: The performance curves showing (a) cumulative probability of success P(M, i) and (b) I(M, i, z) for the sub-function problem**

This experiment demonstrates that LOGENPRO can emulate Koza's ADF and represent easily the knowledge needed for using ADF. Moreover, LOGENPRO can employ other knowledge such as argument types in a unified framework. It has superior performance than that of Koza's ADF when more domain-dependent knowledge is available. One advantage of LOGENPRO is that it can emulate the effects of STGP and ADF simultaneously and effortlessly.

## 6.2. Learning logic programs

In the first sub-section below, we show that this framework can easily emulate GLPS in learning logic programs. In the discussion, the terminologies of logic programming are used. A logic grammar template is also provided to facilitate the application of the framework.

In the following sub-sections, we describe how to use LOGENPRO to learn logic programs. To induce a logic program using LOGENPRO, we have to determine the logic grammar, the fitness function, the termination criterion, the population size, the maximum number of generations, and the probabilities of applying various genetic operations. Three examples are given to show that LOGENPRO can emulate GLPS in solving the learning problems described in chapter 4. Five runs are performed on each problem. The maximum number of generations of each run is 50 for the first two problems and is 20 for the third problem .

### 6.2.1. Learning logic programs using LOGENPRO

A template for learning logic programs using the framework is shown in table 6.4. To apply the template for a particular problem, various variables, constants, predicate symbols, and function symbols will substitute for the identifiers in italics.

```
20:  start        ->   clauses.
21:  clauses      ->   clauses, clauses.
22:  clauses      ->   clause.
23:  clause       ->   consq, [:-], antes, [.].
24:  clause       ->   consq, [.].
25:  consq        ->   literal.
26:  antes        ->   antes, [,], antes.
27:  antes        ->   ante.
28:  ante         ->   literal.
29:  ante         ->   [not], literal.
30a: literal      ->   literal-0.
30b: literal      ->   literal-1.
     ...
     ...
30n: literal      ->   literal-n.
31a: literal-0    ->   lit-0.
31b: literal-1    ->   lit-1, [(], term, [)].
     ...
     ...
31n: literal-n    ->   lit-n, [(],  term, ..., term,[)].
32:  term         ->   {member(?w, <BASIC ELEMENTS>)}, [?w].
33:  term         ->   function.
34a: function     ->   function-0
34b: function     ->   function-1.
     ...
     ...
34n: function     ->   function-n.
35a: function-0   ->   funct-0.
35b: function-1   ->   funct-1, [(], term, [)].
     ...
     ...
35n: function-n   ->   funct-n, [(],  term, ..., term,[)].
36a: funct-0      ->   {member(?w, <function set-0>)}, [?w].
36b: funct-1      ->   {member(?w, <function set-1>)}, [?w].
     ...
     ...
36n: funct-n      ->   {member(?w, <function set-n>)}, [?w].
37a: lit-0        ->   {member(?w, <predicate set-0>)}, [?w].
37b: lit-1        ->   {member(?w, <predicate set-1>)}, [?w].
     ...
     ...
37n: lit-n        ->   {member(?w, <predicate set-n>)}, [?w].
```

**Table 6.4: A template for learning logic programs using LOGENPRO**

To employ LOGENPRO to induce logic programs, basic elements such as variables and constants must be identified first. These elements are usually domain-dependent. Consider the following logic program:

```
cup(X) :- insulate_heat(X), stable(X), liftable(X).
cup(X) :- paper_cup(X).
```

This logic program determines whether an object X is a cup. There are only one variable X in this program. Thus, for this program, the following grammar rule specifies the basic elements:

```
    term ->    {member(?w, [X])}, [?w].
```

This rule is obtained from the grammar rule 32 in the template by substituting the replacing *<BASIC ELEMENTS>* with [X].

A function is a function symbol followed by a bracketed n-tuple of terms. It is specified by the grammar rules 34a, 34b, ..., 34n and 35a, 35b, ..., 35n. The non-terminal symbols function-0, function-1, ..., function-n in these rules represent functions of various arities. The non-terminal symbols funct-0, funct-1, ..., funct-n in the grammar rules 36a, 36b, ..., 36n represent different sets of function symbols. A function is also a term, this fact is declared by rule 33. For learning the above logic program, it does not use functions. Consequently, the above corresponding grammar rules are not included in the specification of this program.

An atomic formula is a predicate symbol immediately followed by a bracketed n-tuple of terms. It is represented by the grammar rules 30a, 30b, ..., 30n and 31a, 31b, ..., 31n. The non-terminal symbols literal-0, literal-1, ..., literal-n in these grammar rules represent predicates (literals) of various arities. The non-terminal symbols lit-0, lit-1, ..., lit-n in grammar rules 37a, 37b, ..., 37n represent various sets of predicate symbols. Because the above logic program contains only predicates having one argument, the grammar rules 30b and 31b in the template are used. The following rule:

```
    lit-1      ->   {member(?w, [cup, insulate_heat,
                               stable, liftable,
                               paper_cup])},[?w].
```

is obtained from rule 37b by replacing *<predicate set-1>* with the list [cup, insulate_heat, stable, liftable, paper_cup]. This rule is also used in the specification of the program.

A logic program is composed of a number of Horn clauses. This fact is specified by the grammar rules 20, 21, and 22. A clause with an empty body is called a unit clause. It represents facts of the problem domain. Since the above program has not unit clauses, only rule 23 is included in the specification of the program. If unit clauses are allowed in the program being induced, rule 24 should be included. The non-terminal symbols consq and antes in rule 23 represent the head and body of a clause respectively. The grammar rule 25 represents that the head of a clause is a positive literal. The body of a clause consists of a sequence of one or more antecedents. It is represented by rules 26 and 27. Finally rules 28 and 29 specify that an antecedent can be a positive or a negative literal.

### 6.2.2. The Winston's arch problem

The logic grammar for the problem described in sub-section 4.5.1 is depicted in table 6.5. It is derived from the logic grammar template presented in table 6.4. Moreover, some grammar rules are combined to simplify the grammar. The logic goal not-equal(?x, ?y) in the grammar ensures that the logic variables ?x and ?y are not instantiated to the same value. The population size is 1000 and the maximum number of generations is 50. The fitness function and the fitness cases are the same as those used in sub-section 4.5.1. LOGENPRO can find a almost correct program within 2 generations. One of the best programs induced is:

```
arch(A, B, C) :- left-of(C, B), wedge(C).
arch(A, B, C) :- left-of(B, C), supports(B, A).
```

```
start              ->    clauses.
clauses            ->    clauses, clauses.
clauses            ->    clause.
clause             ->    consq, [:-], antes, [.].
consq              ->    [arch(A, B, C].
antes              ->    antes, [,], antes.
antes              ->    ante.
ante               ->    {member(?x,[A, B, C])},
                         {member(?y,[A, B, C])},
                         {not-equal(?x, ?y)},
                         literal(?x, ?y).
ante               ->    {member(?x,[A, B, C])},
                         literal(?x).
literal(?x, ?y)    ->    [supports(?x, ?y)].
literal(?x, ?y)    ->    [left-of(?x, ?y)].
literal(?x, ?y)    ->    [touches(?x, ?y)].
literal(?x)        ->    [brick(?x)].
literal(?x)        ->    [wedge(?x)].
literal(?x)        ->    [parallel-piped(?x)].
```

**Table 6.5: The logic grammar for the Winston's arch problem**

Since the standard solution of this problem uses some negative literals, the correct program cannot be found by employing the grammar in table 6.5. If the modified grammar in table 6.6 is applied. The following correct program can be obtained:

```
arch(?A, ?B, ?C) :- left-of(?B, ?C), supports(?B, ?A),
                    not touches(?B, ?C).
```

This example illustrates that different formulations of a learning problem can be attempted easily using different logic grammars.

```
start             ->    clauses.
clauses           ->    clauses, clauses.
clauses           ->    clause.
clause            ->    consq, [:-], antes, [.].
consq             ->    [arch(A, B, C].
antes             ->    antes, [,], antes.
antes             ->    ante.
ante              ->    {member(?x,[A, B, C])},
                        {member(?y,[A, B, C])}
                        {not-equal(?x, ?y)},
                        literal(?x, ?y).
ante              ->    {member(?x,[A, B, C])},
                        literal(?x).
literal(?x, ?y)   ->    [supports(?x, ?y)].
literal(?x, ?y)   ->    [not supports(?x, ?y)].
literal(?x, ?y)   ->    [left-of(?x, ?y)].
literal(?x, ?y)   ->    [not left-of(?x, ?y)].
literal(?x, ?y)   ->    [touches(?x, ?y)].
literal(?x, ?y)   ->    [not touches(?x, ?y)].
literal(?x)       ->    [brick(?x)].
literal(?x)       ->    [not brick(?x)].
literal(?x)       ->    [wedge(?x)].
literal(?x)       ->    [not wedge(?x)].
literal(?x)       ->    [parallel-piped(?x)].
literal(?x)       ->    [not parallel-piped(?x)].
```

**Table 6.6: The modified logic grammar for the Winston's arch problem**

### 6.2.3. The modified Quinlan's network reachability problem

The logic grammar for solving the problem described in sub-section 4.5.2 is shown in table 6.7. In this experiment, the population size is 1000. The standardized fitness is the total number of misclassified training examples. The maximum number of generations is 50. LOGENPRO can find a perfect program that covers all positive examples while excludes all negative ones within a few generations. One of the correct programs found is:

```
    can-reach(A, B) :- linked-to(A, B).
    can-reach(A, B) :- linked-to(A, C), can-reach(C, B).
```

```
start               ->    clauses.
clauses             ->    clauses, clauses.
clauses             ->    clause.
clause              ->    consq, [:-], antes, [.].
consq               ->    [can-reach(A, B)].
antes               ->    antes, [,], antes.
antes               ->    ante.
ante                ->    {member(?x,[A, B, C])},
                          {member(?y,[A, B, C])}
                          {not-equal(?x, ?y)},
                          literal(?x, ?y).
literal(?x, ?y)     ->    [linked-to(?x, ?y)].
literal(?x, ?y)     ->    [can-reach(?x, ?y)].
```

**Table 6.7: The logic grammar for the modified Quinlan's network reachability problem**

## 6.2.4. The factorial problem

This experiment learns the relation `factorial(X, Y)` where `Y` is the factorial of `X`. The predicate symbols are `factorial`, `plus`, and `multiplication`. The logic grammar of this problem is depicted in table 6.8. The population size is 1000 and the maximum number of generations is 20. The fitness functions, the fitness cases, and the initial incorrect clauses are the same as those presented in sub-section 4.5.3.

```
start                    ->    clauses.
clauses                  ->    clauses, clauses.
clauses                  ->    clause.
clause                   ->    consq, [:-], antes, [.].
consq                    ->    {member(?x, [0, 1, 2, X, Y])},
                               {member(?y, [0, 1, 2, X, Y])},
                               [factorial(?x, ?y)].
antes                    ->    antes, [,], antes.
antes                    ->    ante.
ante                     ->    {member(?x,[0, 1, 2, W, X, Y, Z])},
                               {member(?y,[0, 1, 2, W, X, Y, Z])}
                               {member(?z,[0, 1, 2, W, X, Y, Z])}
                               literal(?x, ?y, ?z).
literal(?x, ?y, ?z)      ->    [plus(?x, ?y, ?z)].
literal(?x, ?y, ?z)      ->    [multiplication(?x, ?y, ?z)].
```

**Table 6.8: The logic grammar for the factorial problem**

During one of the runs, the correct logic program is induced in the twelve generation. It is shown as follows:

```
    factorial(0, 1)        :- plus(1, 1, 2).
    factorial(X, Y)        :- plus(Z, 1, X),
                              factorial(Z, W),
                              multiplication(W, X, Y).
```

### 6.2.5. Discussion

From the above examples, LOGENPRO can be viewed as an automatic programming platform on which formal specifications and program induction can be combined. Logic grammars are formal specifications that describe which programs are valid. LOGENPRO employs deduction to generate the initial population of program from the logic grammar given and uses induction to produce offspring from parental programs.

## 6.3. Learning programs in C

In this section, we employ LOGENPRO to perform symbolic regression. The target program calculates the function value f(X, Y) for the two input arguments and outputs the result. The function f(X, Y) is $((X+Y)^2-Y)$ and the population size used in this experiment is 500. The ten fitness cases are 3-tuples $<X_i, Y_i, f(X_i, Y_i)>$, where $1 \leq i \leq 10$ and $X_i$, $Y_i$ are random integers between 0 and 10. The fitness function calculates the sum, taken over the ten fitness cases, of the absolute values of the difference between $f(X_i, Y_i)$ and the value returned by the generated C program using $X_i$ and $Y_i$ as the inputs. A fitness case is said to be covered by a program if the value returned by it is within 0.01 of the desired value. LOGENPRO terminates if the maximum number of generations, which is 50, is reached or a C program that covers all fitness cases is found. It must be emphasized that the goal of this section is to demonstrate the possibility of learning programs in some imperative languages. Hence, the symbolic regression problem is deliberately constructed as simple as possible so as to illustrate the point clearly.

```
start              ->    preamble, statements, outputs.
statements         ->    statements, statements.
statements         ->    statement.
statement          ->    id, [=], expression, [;].
expression         ->    [(], expression, op, expression, [)].
expression         ->    id.
op                 ->    [+].
op                 ->    [-].
op                 ->    [*].
id                 ->    [X].
id                 ->    [Y].
id                 ->    [Z].
preamble           ->    [#include <stdio.h>],
                         [#include <stdlib.h>],
                         [main(argc, argv)],
                         [int argc; char **argv;],
                         [{ int X Y; float Z;],
                         [   X = atoi(argv[1]);],
                         [   Y = atoi(argv[2]);],
                         [   Z = 0.0;].
outputs            ->    [   printf("\n%f", Z)].
```

**Table 6.9: The logic grammar for learning programs in C**

The logic grammar for this problem is shown in table 6.9. In this grammar, only simple assignment statement can be generated. This restriction is enforced only to limit the size of the search space for the problem so that solutions can be found using the available computational resources. In fact, the search space will be extremely large if the complete grammar for the C programming language is used. In this grammar, the symbol `preamble` produces statements that declare and initialize variables used in the program. On the other hand, the symbol `outputs` creates a statement that prints the final result of the program. In one successful run of LOGENPRO, the following correct C program is found in generation 4:

```
#include <stdio.h>
#include <stdlib.h>
main(argc, argv)
int argc; char **argv;
{ int X, Y; float Z;
  X = atoi(argv[1]);
  Y = atoi(argv[2]);
  Z = 0.0;
  Z = (((X-Z)*X)+((Y*Y)+(((X+X)*Y)-Y)));
  printf("\n%f", Z);}
```

The program is correct because the assignment statement $Z=(((X-Z)*X)+((Y*Y)+(((X+X)*Y)-Y)))$ can be simplified to $Z=X^2+Y^2+2XY-Y$ as the variable Z is initialized to 0.0. The statement can be further simplified to $Z=(X+Y)^2-Y$ which is the desired statement.

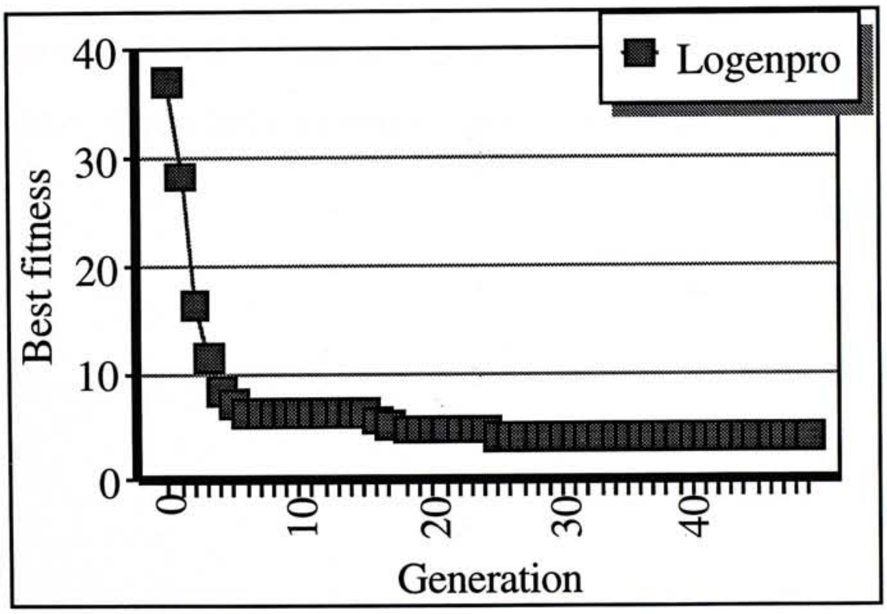


**Figure 6.5: Fitness curve for the problem of inducing a C program**

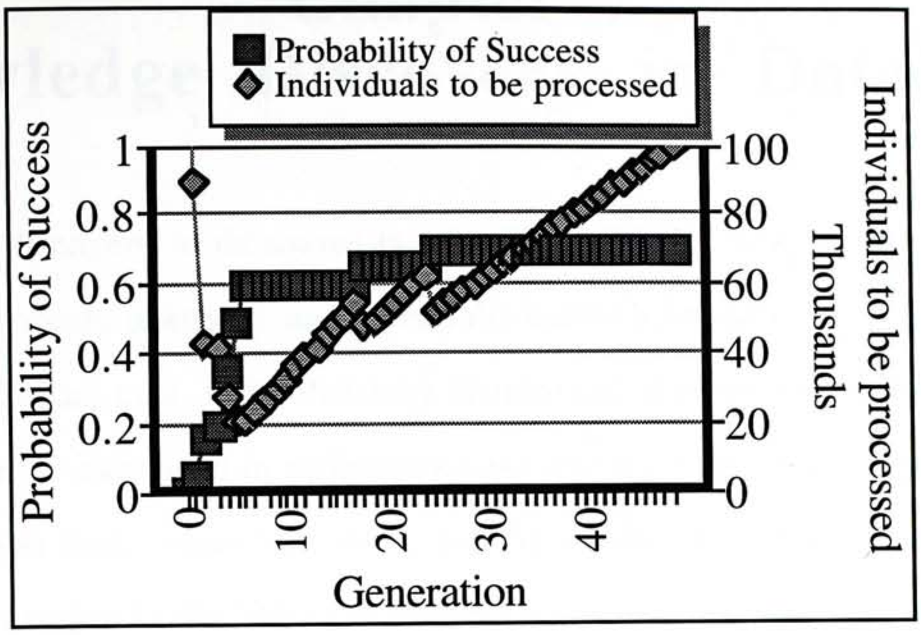


**Figure 6.6: Performance curves for the problem of inducing programs in C**

Twenty trials are attempted using different random number seeds and fitness cases. The results are summarized in figures 6.5 and 6.6. Figure 6.5 shows, by generation, the fitness of the best program in a population. Figure 6.6 shows the performance curves when the population size M is 500 and the probability z is 0.99. The value of I(M, i, z) reaches a minimum value of 21000 at generation 5.

# Chapter 7
# Knowledge Discovery in Databases

Knowledge discovery in databases is concerned with the non-trivial extraction of implicit, previously unknown, and potentially useful information from data stored in databases (Frawley et al. 1991, Piatetsky-Shapiro and Frawley 1991). The knowledge acquired can be expressed in different knowledge representations such as decision trees, decision lists, production rules, and first-order logic programs. In the first section, we employ LOGENPRO to induce knowledge represented in decision trees from a real-world database.

Knowledge discovery systems induce knowledge from datasets which are frequently noisy (incorrect), incomplete, inconsistent, imprecise (fuzzy), and uncertain (Leung and Wong 1991a; 1991b; 1991c). In the second section, we employ LOGENPRO to combine evolutionary algorithms and FOIL (Quinlan, 1990) to induce knowledge represented as logic programs from noisy datasets.

There are very few studies on the issue of inducing knowledge from imprecise and uncertain datasets. Unfortunately, imprecise and uncertain examples are norms rather than exceptions in real world, because many everyday examples are denoted in linguistic terms which are essentially imprecise and uncertain. In the third section, we use LOGENPRO to acquire knowledge from imprecise and uncertain training examples stored in a fuzzy relational database. The induced knowledge is represented as a program in Fuzzy Prolog (Li and Liu 1990).

## 7.1. Inducing decision trees using LOGENPRO

In this section, we illustrate the application of LOGENPRO in inducing decision trees. The first sub-section contains a brief introduction to decision trees. We describe how to represent decision trees as S-expressions in sub-section 7.1.2 . The credit screening problem used in the experiment is explained in the subsequent sub-section. We then present the results of the experiment in sub-section 7.1.4.

### 7.1.1. Decision trees

Decision trees are a means of representing knowledge acquired by a learning system. Quinlan (1986) presented a simple example of the ID3 learning system for inducing a decision tree that classifies whether the weather of a Saturday morning is good or not. ID3 is a hierarchical classification system for learning a decision tree from a finite number of training examples. The training examples and the learned decision tree are depicted in table 7.1 and figure 7.1 respectively.

The set of training examples in table 7.1 contains 14 objects representing characteristics of Saturday mornings. These characteristics are represented by four nominal attributes, namely temperature, humidity, outlook, and windy. An attribute can be classified into three kinds: nominal, linear and structural attributes (Michalski 1983):

- Nominal attribute: The value set consists of independent symbols and no structure is assumed to relate the values in the domain.
- Linear attribute: The value set is an ordered set. Attributes measured on ordinal, interval, ratio, and absolute scales are special cases of linear attribute.
- Structured attribute: The value set has a tree structure that reflects the generalization relation between the values.

The sets of attribute values for temperature, humidity, outlook, and windy are respectively {hot, mild, cool}, {high, normal}, {sunny, overcast, rain}, and {true, false}. An example is positive if it represents that the weather of the Saturday morning is good, otherwise it is negative.

| | **temperature** | **humidity** | **outlook** | **windy** | **class** |
|---|---|---|---|---|---|
| 1 | hot | high | sunny | false | - |
| 2 | hot | high | sunny | true | - |
| 3 | hot | high | overcast | false | + |
| 4 | mild | high | rain | false | + |
| 5 | cool | normal | rain | false | + |
| 6 | cool | normal | rain | true | - |
| 7 | cool | normal | overcast | true | + |
| 8 | mil | high | sunny | false | - |
| 9 | cool | normal | sunny | false | + |
| 10 | mild | normal | rain | false | + |
| 11 | mild | normal | sunny | true | + |
| 12 | mild | high | overcast | true | + |
| 13 | hot | normal | overcast | false | + |
| 14 | mild | high | rain | true | - |

(a)

| **attribute name** | **attribute type** | **attribute values** |
|---|---|---|
| temperature | nominal | {hot, mild, cool} |
| humidity | nominal | {high, normal} |
| outlook | nominal | {sunny, overcast, rain} |
| windy | nominal | {true, false} |
| class | nominal | {positive, negative} |

(b)

**Table 7.1: (a) A set of training examples. (b) The types and the sets of attribute values of the attributes.**

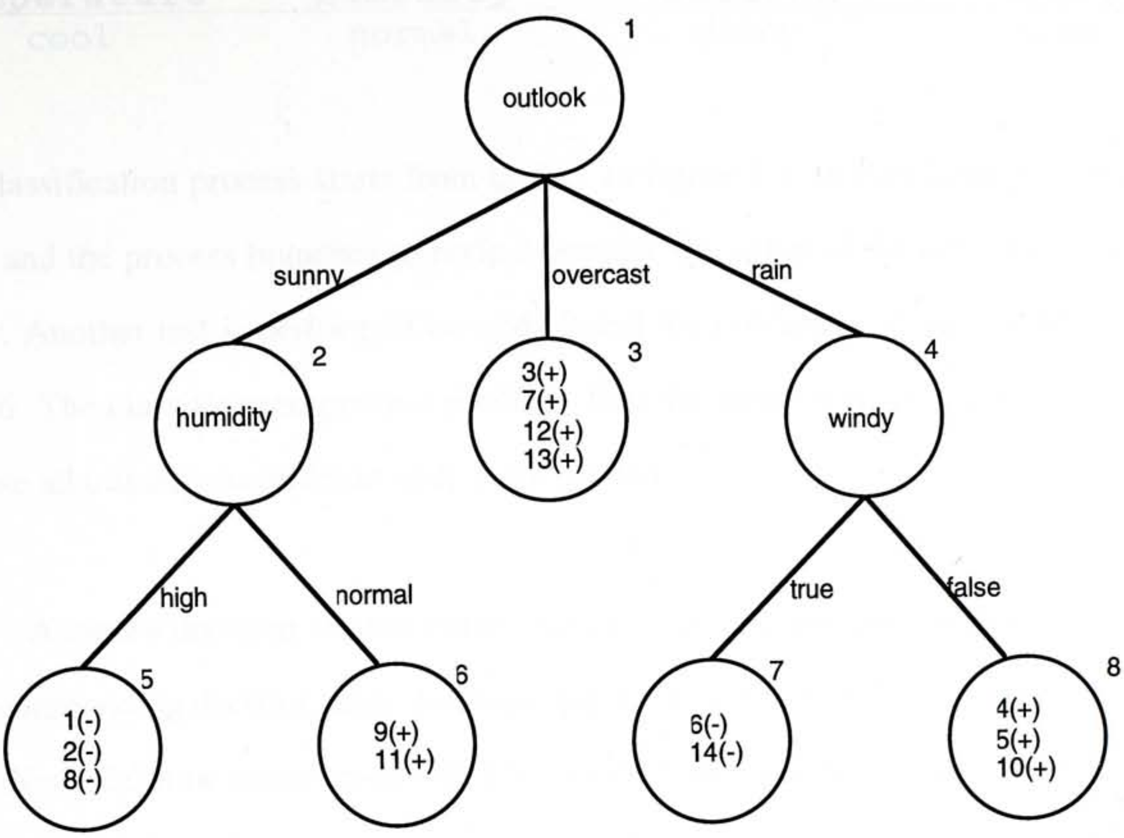


**Figure 7.1: A decision tree**

A decision tree consists of nodes and branches. Each non-terminal (internal) node represents a decision. The starting node is usually referred to as the root node. Depending on the result of the decision of a non-terminal node, the tree will branch to another node. Finally, a terminal (leaf) node is reached, and the classification is decided. For example, node 1 in figure 7.1 is the root node and nodes 2 and 4 are the other non-terminal nodes. There is an attribute name in each non-terminal node. It indicates the attribute on which the decision will be made. Nodes 3, 5, 6, 7, and 8 are terminal nodes. There are numbers in each terminal node, they indicates the training examples that will be classified to the node. The sign, + or −, after each number shows the class of the corresponding training example.

Assume that we want to predict the weather of a Saturday morning using the decision tree in figure 7.1. The characteristics of this morning are summarized as follows:

| **temperature** | **humidity** | **outlook** | **windy** |
|---|---|---|---|
| cool | normal | sunny | true |

The classification process starts from node 1 in figure 7.1. In this node, a decision is made and the process branches to node 2 because the value of the attribute outlook is sunny. Another test is performed on node 2 and the process proceeds to the terminal node 6. The classification process predicts that the weather of this morning is good because all training examples in node 6 are positive.

A binary decision returns either true or false and only two branches can leave the corresponding decision node. For example, node 4 in figure 7.1 is a binary decision node. Node 2 can be transformed into a binary decision node by setting the test to either humidity = high, or humidity = normal. Non-binary decisions are also used. In these cases, more than two branches may leave a non-binary decision node. For example, node 1 in figure 7.1 is a non-binary decision node. A decision performed at a node results in a partition of two or more disjoint sets that cover every possibility, i.e., any new cases must fall into one of the disjoint subsets..

For any decision tree, a path leads to a terminal node corresponding to a decision rule that is a conjunction of the tests along this path. If there are multiple paths for a given class, then these paths represent disjunctions. For example, there are three paths in figure 7.1 for the positive class. Thus, there are three decision rules for this class and they are:

- if outlook is overcast then class is positive.
- if outlook is sunny and humidity is normal then class is positive.
- if outlook is rainy and not windy then class is positive.

All paths in any decision tree are mutually exclusive. Thus, for any new case, one and only one path in the tree will always have to be satisfied.

A decision tree is induced by selecting some starting feature, splitting the training set into disjoint sets according to the selected feature, and then repeating the process for all subsequent nodes. A node becomes terminal and is not splitted further when all members of the training examples in the node belongs to one class. Alternatively, a node becomes terminal when the number of training examples in the remnant group falls below some minimum threshold, and the node is assigned to the class having the greatest frequency at the node. The simplest method for splitting the nodes into disjoint groups is to partition the data by the distinct values of the feature.

However, this splitting method can lead to poor classification. The difficulty arises with linear attributes such as height. Because the set of attribute values of a linear attribute is usually very large or even infinite, it is unreasonable to base predications solely on the values that appear in a small training set. For example, if no one in the training set has height of 68 inches, then a new case with height of 68 inches might not be classified correctly when the attribute height is used in the decision tree. Thus, the values of a linear attribute should be divided into discrete intervals. The optimal sizes of and number of intervals are usually unknown and they are estimated by the learning systems. Arithmetic tests, such as ($A_i > A_{ij}$) or ($A_i < A_{ij}$) where $A_i$ is a linear attribute and $A_{ij}$ is a value within the range of the attribute, can produce intervals that cover more effectively the range of values, and improve the classification performance of the induced decision tree. The learning systems such as AKA-1 and AKA-2 (Leung and Wong 1991a; 1991b; 1991c) can generate this kind of decision tests.

### 7.1.2. Representing decision trees as S-expressions

Koza (1992) presented a method to represent decision trees as S-expressions. For example, the decision tree in figure 7.1 is represented as the S-expression in table 7.2(a).

```
(outlook-test
  (humidity-test 'negative 'positive)
  'positive
  (windy-test 'negative 'positive))
```

(a)

```
(defclass EXAMPLES ()
  ((temperature :accessor temperature)
   ;; The value of the attribute temperature can be either hot, mild,
      or cool.
   (humidity  :accessor humidity)
    ;; The value of the attribute humidity can be either high, or
      normal.
   (outlook   :accessor outlook)
    ;; The value of the attribute outlook can be either sunny,
       overcast, or rain.
   (windy     :accessor windy)))
   ;; The value of the attribute windy can be either true, or false.
```

(b)

```
(defun outlook-test (arg1 arg2 arg3)
  (cond ((equal (outlook X) 'sunny) arg1)
        (equal (outlook X) 'overcast) arg2)
        (t arg3)))
```

(c)

**Table 7.2: (a) An S-expression that represents the decision tree in figure 7.1. (b) The class definition of the training and testing examples. (c) A definition of the primitive function outlook-test.**

In the S-expression, the constants such as `positive` and `negative` representing the class names in this problem. These constants form the set of terminals in GP. On the other hand, the attribute-testing functions such as outlook-test and windy-test are obtained by transforming each of the attributes in the problem into a function. Thus, there are as many attribute-testing functions as there are attributes. These functions form the set of primitive functions in GP.

Consider the attribute outlook, it can assume one of three possible values. Therefore, the function `outlook-test` has three arguments and operates in the following way:

- if the value of the attribute outlook of the current example is sunny, the function returns its first argument as its return value;
- if the value of the attribute outlook of the current example is overcast, the function returns its second argument as its return value;
- if the value of the attribute outlook of the current example is rainy, the function returns its third argument as its return value;

The implementation of the function `outlook-test` is depicted in table 7.2(c). In this implementation, `X` is a global variable that stores the current example being evaluated. Since an example belongs to the class `EXAMPLES` depicted in table 7.2(b), the S-expression `(outlook X)` returns the value of the attribute outlook of the example stored in `X`. The constants `sunny` and `overcast` represent the attribute values of the attribute outlook.

To classify a new example, it is first stored into the global variable `X`. It is then presented to an S-expression representing a decision tree. The function at the root of the tree tests the designated attribute of the example and then executes the particular argument designated by the outcome of the test. If the designated argument is a constant, the function returns the corresponding class names (i.e. `positive` or `negative`). If the designated argument is another function, the above process is repeated until a constant is returned. In summary, the S-expression is a representation of a decision tree that classifies an example into one of the classes.

### 7.1.3. The credit screening problem

The aim of this problem is to induce decision trees or rules for assessing applications for credit cards. This problem has been studied by Quinlan in his ID3 and C4.5 systems (Quinlan 1987; 1992). The original dataset of this problem was provided by Quinlan and stored in the UCI Repository of Machine Learning Databases and Domain Theories. The dataset has been modified in the Statlog project (Michie et al. 1994) so that one of the 15 attributes is removed. The modified dataset has a good mix of attributes of different types. There are 690 instances, 14 attributes and two class names. There are 307 positive instances (44.5%) and 383 negative instances (55.5%).

| Attribute name | Attribute type | Attribute values |
|---|---|---|
| A1 | nominal | {a, b} |
| A2 | linear | 13.75 - 80.25 |
| A3 | linear | 0 - 28 |
| A4 | nominal | {g, p, gg} |
| A5 | nominal | {c, d, cc, i, j, k, m, r, q, w, x, e, aa, ff} |
| A6 | nominal | {v, h, bb, j, n, z, dd, ff, o} |
| A7 | linear | 0 - 28.5 |
| A8 | nominal | {t, f} |
| A9 | nominal | {t, f} |
| A10 | linear | 0 - 67 |
| A11 | nominal | {t, f} |
| A12 | nominal | {g, p, s} |
| A13 | linear | 0 - 2000 |
| A14 | linear | 0 - 100001 |
| class | nominal | {positive, negative} |

**Table 7.3: The attribute names, types, and values attributes of the credit screening problem**

All attribute names, class names, and attribute values have been changed to meaningless symbols to protect confidentiality of the data. Thus, interpretations of the induced decision trees or rules are relatively difficult. This dataset is interesting because there is a good mix of attribute types: linear, nominal with small numbers of values, and nominal with larger numbers of values. The attribute names, types, and values are depicted in table 7.3. There are 37 instances (5%) having one or more missing attribute values. The frequencies of missing values from different attributes are summarized as follows:

| Attribute name | Frequency |
|---|---|
| A1 | 12 |
| A2 | 12 |
| A4 | 6 |
| A5 | 9 |
| A6 | 9 |
| A13 | 13 |

For our purposes, we replaced the missing values by the overall medians or means.

### 7.1.4. The experiment

In this sub-section, we describe how to use LOGENPRO to induce decision trees for the credit screening problem. The representation scheme described in sub-section 7.1.2 is not used directly because it can only express decisions on nominal attributes. To handle linear attributes using the representation, we must first transform these attributes into nominal attributes by assigning disjoint intervals of values to various symbols. Thus, the sizes and the number of intervals must be determined before applying the representation scheme to the credit screening problem.

For example, the range of the values of the attribute A2 is between 13.75 and 80.25. By examining the distribution of the attribute values, the range may be divided into two mutual exclusive intervals: from inclusive 13.75 to exclusive 40; from inclusive 40 to inclusive 80.25. The transformed attribute can be represented as the following attribute-testing function A2-test:

```
(defun A2-test (arg1 arg2)
  (if (>= (A2 X) 40)
   arg2
   arg1))
```

In this function, X is a global variable that stores the current example being evaluated. Since an example belongs to the class EXAMPLES depicted in table 7.4, the S-expression (A2 X) returns the value of the attribute A2 of the example stored in X. The function A2-test has two arguments and operates in the following way:

- if the value of the attribute A2 is greater than or equal to 40, the function returns its second argument as its return value;
- Otherwise, the function returns its first argument as its return value;

```
(defclass EXAMPLES ()
  ((A1  :accessor A1)
   (A2  :accessor A2)
   (A3  :accessor A3)
   (A4  :accessor A4)
   (A5  :accessor A5)
   (A6  :accessor A6)
   (A7  :accessor A7)
   (A8  :accessor A8)
   (A9  :accessor A9)
   (A10 :accessor A10)
   (A11 :accessor A11)
   (A12 :accessor A12)
   (A13 :accessor A13)
   (A14 :accessor A14)))
```

**Table 7.4: The class definition of the training and testing examples.**

The major problem of this representation is that one or more intervals must be determined before performing induction. If the sizes and the number of intervals are inappropriate, they will greatly reduce the performance of the learning system. In order to tackle this problem, we decide that the number of intervals of all linear attributes is fixed to two, and allow the sizes of these intervals to adjust dynamically during the evolution process.

Thus, the following attribute-testing function `A2-test` is used in our representation:

```
(defun A2-test (exp arg1 arg2)
  (if (>= (A2 X) exp)
   arg2
   arg1))
```

This function has three arguments and operates in the following way:

- if the value of the attribute A2 is greater than or equal to the value of the first argument, the function returns its third argument as its return value;
- Otherwise, the function returns its second argument as its return value;

From this function, we can observe that the first argument `exp` must return a numerical value while the other two arguments, `arg1` and `arg2`, must return a class name. In other words, data types must be used to guarantee only appropriate S-expressions can appear as a particular argument of a particular primitive function.

```
start                    ->    node.
node                     ->    [ (A1 ], node, node, [ ) ].
node                     ->    [ (A2 ], exp, node, node, [ ) ].
node                     ->    [ (A3 ], exp, node, node, [ ) ].
node                     ->    [ (A4 ], node, node, node [ ) ].

node                     ->    [ (A5 ], node, node, node, node,
                               node, node, node, node, node,
                               node, node, node, node, node, [ ) ].
node                     ->    [ (A6 ], node, node, node, node,
                               node, node, node, node, node, [ ) ].
node                     ->    [ (A7 ], exp, node, node, [ ) ].
node                     ->    [ (A8 ], node, node, [ ) ].
node                     ->    [ (A9 ], node, node, [ ) ].
node                     ->    [ (A10 ], exp, node, node, [ ) ].
node                     ->    [ (A11 ], node, node, [ ) ].
node                     ->    [ (A12 ], node, node, node, [ ) ].
node                     ->    [ (A13 ], exp, node, node, [ ) ].
node                     ->    [ (A14 ], exp, node, node, [ ) ].
node                     ->    [ positive ].
node                     ->    [ negative ]
exp                      ->    [ ( ], op, exp, exp, [ ) ].
op                       ->    [ + ].
op                       ->    [ - ].
op                       ->    [ * ].
op                       ->    [ % ].
exp                      ->    { random(-10, 10, ?a) },  [ ?a ].
```

**Table 7.5: Logic grammar for the credit screening problem.**

To induce a functional program using LOGENPRO, We have to determine the logic grammar, the fitness cases, the fitness functions, and the termination criterion. The logic grammar for the credit screening problem is given in table 7.5. In this grammar, we employ the grammar symbol `exp` to designate the S-expression that returns a numerical value and the grammar symbol `node` to designate the S-expression that returns a class name.

The terminal symbols `+`, `-`, and `*` represent functions that perform ordinary addition, subtraction, and multiplication respectively. The symbol `%` represents function that normally returns the quotient. However, if division by zero is attempted, the function returns 1.0. The logic goal `random(-10, 10, ?a)` generates a random floating point number between -10 and 10 and instantiates ?a to the random number generated.

A 10-fold cross-validation procedure is employed in this problem. In a general n-fold cross-validation procedure, the examples are randomly divided into n mutually exclusive test partitions of approximately equal size. The examples not found in a particular test partition are used for training, and the resulting decision tree is tested on the corresponding test partition. The above train and test procedure is repeated n times until all test partitions are examined. The average classification accuracy over all n test partitions is the cross-validated classification accuracy. Breiman et al. (1984) have evaluated their CART system extensively with vary numbers of partitions, and 10-fold cross-validation seemed to be adequate and accurate.

Since there are 690 examples in the credit screening dataset, each test partition contains 69 examples and the other 621 examples form the training set. In other words, 10 independent experiments are attempted. In each experiment, LOGENPRO induces a decision tree using 621 examples as the fitness cases and we estimate the classification accuracy of the induced decision tree using the remaining testing examples.

The fitness function measures how well a genetically evolved decision tree classifies the fitness cases. When an evolved decision tree in the population is tested against a particular fitness case, the outcome can be either a true positive, a true negative, a false positive, or a false negative.

The correlation coefficient (Matthews 1975) indicates how much better a particular decision tree is than a random classifier. A correlation coefficient C of 1.0 indicates perfect agreement between the decision tree and the fitness cases; a coefficient of -1.0 indicates total disagreement; a coefficient of 0.0 indicates that the decision tree is not better than a random classifier. For a two-classes classification problem, the correlation coefficient can be computed as:

$$C = \frac{N_{tp}N_{tn} - N_{fp}N_{fn}}{\sqrt{(N_{tn} + N_{fn})(N_{fp} + N_{tn})(N_{tp} + N_{fn})(N_{tp} + N_{fp})}}$$

where $N_{tp}$ is the number of true positives, $N_{tn}$ is the number of true negatives, $N_{fp}$ is the number of false positives, and $N_{fn}$ is the number of false negatives. The coefficient is set to 0 if the denominator is 0.

Since C ranges between -1.0 and 1.0, standardized fitness is defined as $\frac{1-C}{2}$. Thus, a standardized fitness value ranges between 0.0 and 1.0. A standardized fitness value of 0 indicates perfect agreement between the decision tree and the training examples. On the other hand, a value of 1.0 indicates total disagreement. A value of 0.5 shows that the decision tree is not better than a random classifier.

In each of the ten experiment, LOGENPRO induces a decision tree using a population size of 300. LOGENPRO will stop if the maximum number of generations of 50 is reached or a decision tree that has a standardized fitness below 0.01 is found. The decision tree evolved in any generation that has the smallest standardized fitness value is returned as the result of the run. The best decision tree induced by LOGENPRO is further evaluated on the training examples and the testing examples to obtain the corresponding classification accuracy. The results of the ten experiments are summarized in table 7.6.

| Generation | Accuracy (train) | Accuracy (test) |
|---|---|---|
| 0 | 0.857 | 0.870 |
| 14 | 0.850 | 0.928 |
| 26 | 0.873 | 0.754 |
| 32 | 0.862 | 0.884 |
| 45 | 0.860 | 0.870 |
| 2 | 0.849 | 0.928 |
| 25 | 0.868 | 0.797 |
| 4 | 0.858 | 0.826 |
| 28 | 0.852 | 0.913 |
| 22 | 0.863 | 0.812 |
| Average | 0.859 | 0.858 |

**Table 7.6: Results of the decision trees induced by LOGENPRO for the credit screening problem. The first column shows the generation in which the best decision tree is found. The second column contains the classification accuracy of the best decision tree on the training examples. The third column shows the accuracy on the testing examples.**

Michie et al. (1994) has performed a series of experiments in the Statlog project. In these experiments, they compared the performances of different learning systems for the credit screening problem. The results are summarized in table 7.7.

By comparing the results in table 7.6 and those in table 7.7, we find that Cal5, ITrule, Discrim, Logdisc, and DIPOL92 perform better than LOGENPRO. Cal5 and ITrule learns decision trees/rules and their classification accuracy is over 86%. The classification accuracy of Discrim, Logdisc, and DIPOL92 is all 85.9%, The differences in accuracy between them and LOGENPRO are only 0.1%. Since the detailed information of the accuracy of these systems is not available, it cannot be concluded that whether the differences in accuracy are significant.

On the other hand, LOGENPRO performs better than CART, RBF, CASTLE, NaiveBay, IndCART, Back-propagation, C4.5, SMART, Baytree, k-NN, NewID, AC2, LVQ, ALLOC80, CN2, and Quadisc for the credit screening problem. Interestingly, LOGENPRO is better than C4.5 and CN2, two systems that have been reported in the literature (Quinlan 1992, Clark and Niblett 1989) about their outstanding performances in inducing decision trees/rules. The difference is 1.3% for C4.5 and is 6.2% for CN2.

| Algorithm | Accuracy (train) | Accuracy (test) |
|---|---|---|
| Cal5 | 0.868 | 0.869 |
| ITrule | 0.838 | 0.863 |
| Discrim | 0.861 | 0.859 |
| Logdisc | 0.875 | 0.859 |
| DIPOL92 | 0.861 | 0.859 |
| **LOGENPRO** | **0.859** | **0.858** |
| CART | 0.855 | 0.855 |
| RBF | 0.893 | 0.855 |
| CASTLE | 0.856 | 0.852 |
| NaiveBay | 0.864 | 0.849 |
| IndCART | 0.919 | 0.848 |
| Back-propagation | 0.913 | 0.846 |
| C4.5 | 0.901 | 0.845 |
| SMART | 0.910 | 0.842 |
| Baytree | 1.000 | 0.829 |
| k-NN | 1.000 | 0.819 |
| NewID | 1.000 | 0.819 |
| AC2 | 1.000 | 0.819 |
| LVQ | 0.935 | 0.803 |
| ALLOC80 | 0.806 | 0.799 |
| CN2 | 0.999 | 0.796 |
| Quadisc | 0.815 | 0.793 |

**Table 7.7: Results of various learning algorithms for the credit screening problem.**

## 7.2. Learning logic program from imperfect data

In knowledge discovery from databases, we emphasize the need for learning from huge, incomplete, and imperfect datasets (Piatetsky-Shapiro and Frawley 1991). The various kinds of imperfections in data are listed as follows:

- random noise in training examples and background knowledge;
- the number of training examples is too small;
- the distribution of training examples fails to reflect the underlying distribution of instances of the concept being learned;
- an inappropriate example description language is used: some important characteristics of examples are not represented, and/or irrelevant properties of examples are provided;
- an inappropriate concept description language is used: it does not contain an exact description of the target concept; and
- there are missing values in the training examples.

Existing inductive learning systems employ noise-handling mechanisms to cope with the first five kinds of data imperfections. Missing values are usually handled by a separate mechanism. These noise-handling mechanisms are designed to prevent the induced concept from overfitting the imperfect training examples by excluding insignificant patterns (Lavrac and Dzeroski 1994). They include tree pruning in CART (Breiman et al. 1984), rule truncation in AQ15 (Michalski et al. 1986a) and significant test in CN2 (Clark and Niblett 1989). However, these mechanisms may ignore some important patterns because they are statistically insignificant.

Moreover, these learning systems use a limiting attribute-value language for representing the training examples and induced knowledge. This representation limits them to learn only propositional descriptions in which concepts are described in terms of values of a fixed number of attributes. Currently, only a few relation learning systems such as FOIL and mFOIL address the issue of learning from imperfect data.

In this section, we describe the application of LOGENPRO to learn logic programs from noisy and imperfect training examples. Empirical comparisons of LOGENPRO with FOIL (the publicly available version of FOIL, version 6.0 , is used in this experiment) and with mFOIL (Lavrac and Dzeroski 1994) in the domain of learning illegal chess endgame positions from noisy examples are presented.

As described in section 3.3, mFOIL is based on FOIL that has adapted several features from CN2 (Clark and Niblett 1989), such as the use of the Laplace and m-estimate as a search heuristics and the use of significance testing as a stopping criterion. Moreover, mFOIL uses beam search and can apply mode and type information to reduce the search space. The parameters that can be set by a user are listed as follows:

- the beam width,
- the search heuristics,
- the value of m if m-estimate is used as the search heuristics,
- the significance threshold used in the significance test, and
- the concept description language: it determines whether negative literals can appear in the body of a clause.

A number of different instances of mFOIL have been tested on the chess endgame problem. Their parameter values are summarized in table 7.8.

| | **beam width** | **search heuristics** | **m** | **significance threshold** | **Is negative literal used?** |
|---|---|---|---|---|---|
| mFOIL1 | 5 | m-estimate | 0.01 | 0 | yes |
| mFOIL2 | 5 | m-estimate | 0.01 | 0 | no |
| mFOIL3 | 10 | m-estimate | 0.01 | 0 | yes |
| mFOIL4 | 10 | m-estimate | 0.01 | 0 | no |
| mFOIL5 | 5 | m-estimate | 0.01 | 6.35 | no |

**Table 7.8: The parameter values of different instances of mFOIL examined in this section.**

In this section, LOGENPRO employs a variation of FOIL to find the initial population of logic programs. Thus, it uses the same noise-handling mechanism of FOIL. The variation is called BEAM-FOIL because it uses a beam search method rather than the greedy search strategy of FOIL. BEAM-FOIL produces a number of different logic programs when it terminates and the best program among them is the solution of the problem. The logic programs created by BEAM-FOIL are used by LOGENPRO to initialize the first generation. In order to study the effects of the genetic operations performed by LOGENPRO on the initial programs provided by BEAM-FOIL, a comparison between them is also discussed.

The chess endgame problem is presented in the following sub-section. The experimental setup is detailed in sub-section 7.2.2. We compare LOGENPRO with other learning systems in the subsequent sub-sections.

### 7.2.1. The chess endgame problem

In the chess endgame problem, the setup is white king and rook versus black king (Quinlan 1990). The target concept illegal(WKf, WKr, WRf, WRr, BKf, BKr) states whether the positions where the white king at (WKf, WKr), the white rook at (WRf, WRf), and the black king at (BKf, BKr) are not a legal white-to-move position.

The background knowledge is represented by two predicates, adjacent(X, Y) and less_than(W, Z), indicating that rank/file X is adjacent to rank/file Y and rank/file W is less than rank/file Z respectively.

There are 11000 examples in the dataset (3576 positive and 7424 negative examples). Muggleton et al. (1989) used smaller datasets to evaluate the performances of CIGOL and DUCE for the chess endgame problem. There were five small sets of 100 examples each and five large sets of 1000 examples each. In other words, there were 5500 examples in total. Each of the sets was used as a training set. The induced programs obtained from a small training set was tested on the 5000 examples from the large sets, the programs obtained from each large training set was tested on the remaining 4500 examples.

### 7.2.2. The setup of experiments

In each experiment of the ten experiments performed, the training set contains 1000 examples (336 positive and 664 negative examples) and the disjoint testing set has 10000 examples (3240 positive and 6760 negative examples). These training and testing sets are selected from the dataset using different seeds for the random number generator.

Different amounts of noise are introduced into the training examples in order to study the performances of different systems in learning logic programs from noisy environment. To introduce n% of noise into argument X of the training examples, the value of argument X is replaced by a random value of the same type from a uniform distribution, independent to noise in other arguments. For the class variable, n% positive examples are labeled as negative ones while n% negatives examples are labeled as positive ones. Noise in an argument is not necessarily incorrect because it is chosen randomly, it is possible that the correct argument value is selected. In contrast, noise in classification implies that this example is incorrect. Thus, the probability for an example to be incorrect is $1-\{[(1-n\%)+n\%*\frac{1}{8}]^6*(1-n\%)\}$. For each experiment, the percentages of introduced noise are 5%, 10%, 15%, 20%, 30%, and 40%. Thus, the probabilities for an example to be noisy are respectively 27.36%, 48.04%, 63.46%, 74.78%, 88.74% and 95.47%. Background knowledge and testing examples are not corrupted with noise.

A chosen level of noise is first introduced in the training set. Logic programs are then induced from the training set using LOGENPRO, FOIL, different instances of mFOIL, and BEAM-FOIL. Finally, the classification accuracy of the learned logic programs is estimated on the testing set. For BEAM-FOIL, the size of beam is ten and thus ten logic programs are returned. The best one among the programs returned is designated as the solution of BEAM-FOIL.

```
start              ->    clauses.
clauses            ->    clauses, clauses.
clauses            ->    clause.
clause             ->    consq, [:-], antes, [.].
consq              ->    [illegal(WKf, WKr, WRf, WRf, BKf, BKr)].
antes              ->    antes, [,], antes.
antes              ->    ante.
ante               ->    {member(?x,[WKf, WKr, WRf, WRf, BKf, BKr])},
                         {member(?y,[WKf, WKr, WRf, WRf, BKf, BKr])},
                         literal(?x, ?y).
literal(?x, ?y)    ->    [?x = ?y].
literal(?x, ?y)    ->    [ not ?x = ?y].
literal(?x, ?y)    ->    [ adjacent(?x, ?y) ].
literal(?x, ?y)    ->    [ not adjacent(?x, ?y) ].
literal(?x, ?y)    ->    [ less_than(?x, ?y) ].
literal(?x, ?y)    ->    [ not less_than(?x, ?y) ].
```

**Table 7.9: The logic grammar for the chess endgame problem.**

LOGENPRO uses the logic grammar in table 7.9 to solve t problem. In the grammar, `[adjacent(?x, ?y)]` and `[less_than(?x, ?y)]` are terminal symbols. The logic goal `member(?x, [WKf, WKr, WRf, WRr, BKf, BKr])` will instantiate logic variable `?x` of the grammar to either `WKf`, `WKr`, `WRf`, `WRr`, `BKf`, or `BKr`, the logic variables of the target logic program.

The population size for LOGENPRO is 10 and the maximum number of generations is 50. In fact, different population sizes have been tried and the results are still satisfactory even for a very small population. This observation is interesting and it demonstrates the advantage of combining inductive logic programming and evolutionary algorithms using the proposed framework. The fitness function of LOGENPRO evaluates the number of training examples misclassified by each individual in the population. Since LOGENPRO is a probabilistic system, five runs of each experiment are performed and the average of the classification accuracy of these five runs is returned as the classification accuracy of LOGENPRO for the particular experiment. In other words, fifty runs of LOGENPRO have been performed in total. The results of these systems are summarized in table 7.10 . The performances of these systems are compared using the one-tailed paired *t*-test with 0.05% level of significance.

|  | Noise Level |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|
|  | **0.00** | **0.05** | **0.10** | **0.15** | **0.20** | **0.30** | **0.40** |
| **LOGENPRO (Average)** | 0.996 | 0.983 | 0.960 | 0.938 | 0.855 | 0.733 | 0.670 |
| **Variance** | 0.00E+00 | 7.74E-06 | 2.96E-04 | 7.85E-04 | 2.57E-03 | 2.47E-03 | 1.44E-04 |
| **FOIL (Average)** | 0.996 | 0.898 | 0.819 | 0.761 | 0.693 | 0.596 | 0.529 |
| **Variance** | 0.00E+00 | 5.07E-04 | 6.56E-04 | 5.15E-04 | 5.30E-04 | 3.35E-04 | 3.11E-04 |
| **BEAM-FOIL (Average)** | 0.996 | 0.802 | 0.757 | 0.744 | 0.724 | 0.685 | 0.674 |
| **Variance** | 0.00E+00 | 7.07E-04 | 1.62E-04 | 1.88E-04 | 2.00E-04 | 1.40E-04 | 1.04E-04 |
| **mFOIL1 (Average)** | 0.985 | 0.796 | 0.792 | 0.758 | 0.730 | 0.705 | 0.686 |
| **Variance** | 0.00E+00 | 3.67E-04 | 3.30E-04 | 3.60E-04 | 1.29E-04 | 1.83E-04 | 8.94E-05 |
| **mFOIL2 (Average)** | 0.985 | 0.883 | 0.845 | 0.815 | 0.785 | 0.719 | 0.685 |
| **Variance** | 0.00E+00 | 5.15E-05 | 7.29E-05 | 3.12E-04 | 2.15E-04 | 1.39E-04 | 1.30E-04 |
| **mFOIL3 (Average)** | 0.892 | 0.807 | 0.791 | 0.765 | 0.733 | 0.704 | 0.693 |
| **Variance** | 1.97E-16 | 2.46E-04 | 5.15E-04 | 4.02E-04 | 8.10E-05 | 8.72E-05 | 1.33E-04 |
| **mFOIL4 (Average)** | 0.985 | 0.932 | 0.888 | 0.842 | 0.798 | 0.713 | 0.680 |
| **Variance** | 0.00E+00 | 7.47E-05 | 9.16E-05 | 9.26E-04 | 3.09E-04 | 1.41E-04 | 3.05E-04 |
| **mFOIL5 (Average)** | 0.896 | 0.836 | 0.805 | 0.771 | 0.723 | 0.068 | 0.000 |
| **Variance** | 1.97E-16 | 7.83E-04 | 1.05E-04 | 1.89E-04 | 9.81E-04 | 4.69E-02 | 0.00E+00 |

**Table 7.10: The averages and variances of accuracy of LOGENPRO, FOIL, BEAM-FOIL, and different instances of mFOIL at different noise levels.**

### 7.2.3. Comparison of LOGENPRO with FOIL

The classification accuracy of both systems degrades seriously as the noise level increases (figure 7.2). The classification accuracy of LOGENPRO decreases smoothly when the noise level is on or below 0.15. It reduces from 0.996 to 0.938, a 5.8% decrement. There are sudden drops of accuracy when the noise level is between 0.15 and 0.40. It falls from 0.938 to 0.670, a 28.5% reduction. The accuracy of FOIL decreases rapidly when the noise level is on or below 0.20. It drops from 0.996 to 0.693, a 30.4% reduction. The decrease slightly slows down between the noise levels of 0.20 and 0.40. It drops from 0.693 to 0.529, a 23.7% reduction.

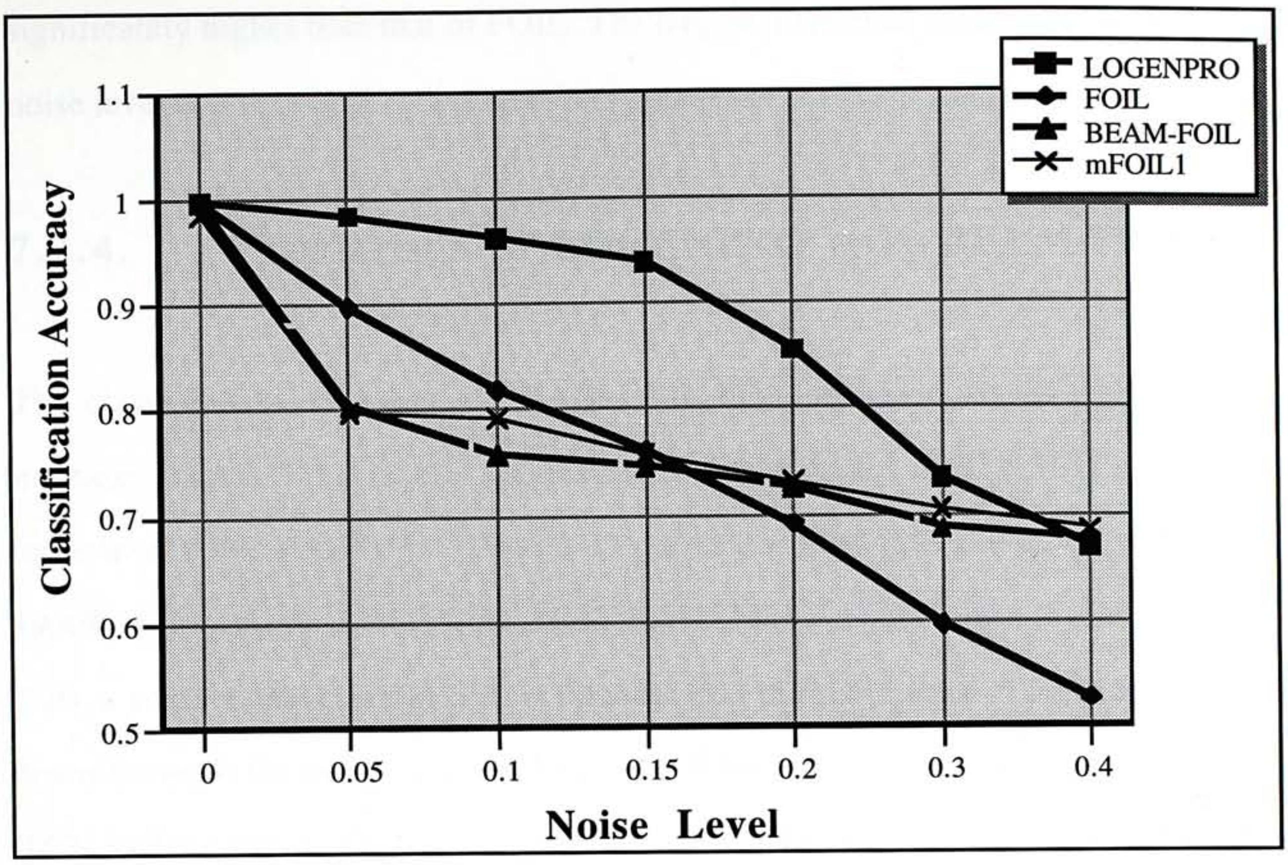


**Figure 7.2: Comparison between LOGENPRO, FOIL, BEAM-FOIL, and mFOIL1**

The results are statistically evaluated using the one-tailed paired *t*-test. For each noise level, the classification accuracy is compared to test the null hypothesis against the alternative hypothesis. The null hypothesis states that the difference in accuracy is zero at the 0.05% level of significance. On the other hand, the alternative hypothesis declares that the difference is greater than zero at the 0.05% level of significance. The t-statistics are listed as follows:

| Noise Level | 0.00 | 0.05 | 0.10 | 0.15 | 0.20 | 0.30 | 0.40 |
|---|---|---|---|---|---|---|---|
| t-statistics | NA | 12.59 | 17.78 | 19.33 | 14.17 | 8.07 | 26.82 |

The t-statistics at the 0.00 noise level is not available because the variances are very small (near zero). The t-statistics at the 0.05 noise level is 12.59 which is greater than the critical value of 4.78. Thus, we can reject the null hypothesis and assert that the classification accuracy of LOGENPRO is higher than that of FOIL. Similarly, the classification accuracy of LOGENPRO at the noise levels between 0.05 and 0.40 is significantly higher than that of FOIL. The largest difference reaches 0.177 at the 0.15 noise level.

### 7.2.4. Comparison of LOGENPRO with BEAM-FOIL

The classification accuracy of BEAM-FOIL degrades seriously as the noise level increases (figure 7.2). There is a significant fall in accuracy of BEAM-FOIL when the noise level is increased from 0.0 to 0.05. It reduces from 0.996 to 0.802, a more than 19.4% of decrement. It falls from 0.802 to 0.757 between the noise levels of 0.05 and 0.10, a smaller reduction (5.6%) is encountered in this interval. The decrease slows down between the noise levels of 0.10 and 0.40. The accuracy drops from 0.757 to 0.674 in this interval. The reduction is about 11%. The results of the one-tailed paired *t*-test are listed as follows:

| Noise Level | 0.00 | 0.05 | 0.10 | 0.15 | 0.20 | 0.30 | 0.40 |
|---|---|---|---|---|---|---|---|
| t-statistics | NA | 22.20 | 33.82 | 21.91 | 9.19 | 3.26 | -0.81 |

The t-statistics at the 0.00 noise level is not available because the variances are very small (near zero). The t-statistics at the 0.05 noise level is 22.20 which is greater than the critical value of 4.78. Thus, we can assert that the classification accuracy of LOGENPRO is higher than that of BEAM-FOIL. Similarly, the classification accuracy of LOGENPRO at the noise levels between 0.10 and 0.20 is significantly higher than that of BEAM-FOIL. At the noise level of 0.30, the accuracy of LOGENPRO is higher than that of BEAM-FOIL, but the difference is not significant because the t-statistics is only 3.26 which is smaller than the critical threshold. On the other hand, the accuracy of BEAM-FOIL at the noise level of 0.40 is higher than that of LOGENPRO, but the difference is insignificant because the absolute value of -0.81 is smaller than the critical value. This comparison indicates that the genetic operations of LOGENPRO can actually improve the logic programs generated by other learning systems such as BEAM-FOIL.

### 7.2.5. Comparison of LOGENPRO with mFOIL1

We compare LOGENPRO with mFOIL1 to mFOIL5 (see section 3.3) one by one in this and the following sub-sections. The parameters of this instance are presented in table 7.8. Lavrac and Dzeroski (1994) compare the performances of mFOIL1 with FOIL2.0, a version of FOIL, for the chess endgame problem using the smaller dataset described in sub-section 7.2.1. They find that mFOIL1 outperforms FOIL2.0 at all noise levels. Our results depicted in figure 7.2 are inconsistent with those obtained by Lavrac and Dzeroski. We find that FOIL outperforms mFOIL1 at the noise levels of 0.05 and 0.1. On the other hand, mFOIL1 has better performance when the noise level is over 0.1. The inconsistency may be explained because we employ an improved version of FOIL, FOIL6.0, and larger sets of training and testing examples.

There is a significant fall in accuracy of mFOIL1 (figure 7.2) when the noise level is changed from 0.0 to 0.05. It reduces from 0.985 to 0.796, a more than 19% of decrement. It falls from 0.796 to 0.792 between the noise levels of 0.05 and 0.10, a very small reduction (0.5%) is encountered in this interval. The drop slows down between the noise levels of 0.10 and 0.40. The accuracy falls from 0.792 to 0.686 in this interval. The reduction is only 13%. The results of the one-tailed paired *t*-test are listed as follows:

| Noise Level | 0.00 | 0.05 | 0.10 | 0.15 | 0.20 | 0.30 | 0.40 |
|---|---|---|---|---|---|---|---|
| t-statistics | 3.03E+08 | 29.35 | 22.78 | 16.17 | 7.52 | 2.27 | -3.92 |

Since t-statistics at the 0.00 noise level is very large, we can assert that the classification accuracy of LOGENPRO is higher than that of mFOIL1. The difference is about 0.01. The t-statistics at the 0.05 noise level is 29.35 which is greater than the critical value of 4.78. Thus, the classification accuracy of LOGENPRO at this noise level is significantly higher than that of mFOIL1, with a difference of 0.19. Similarly, the classification accuracy of LOGENPRO at the noise levels between 0.10 and 0.20 is significantly higher than that of mFOIL1. At the noise level of 0.30, the accuracy of LOGENPRO is higher than that of mFOIL by about 0.03, but the difference is not significant because the t-statistics is only 2.27 which is smaller than the critical threshold. On the other hand, the accuracy of mFOIL1 at the noise level of 0.40 is higher than that of LOGENPRO, the difference is not significant because the absolute value of -3.92 is smaller than the critical value. The difference is about 0.014.

### 7.2.6. Comparison of LOGENPRO with mFOIL2

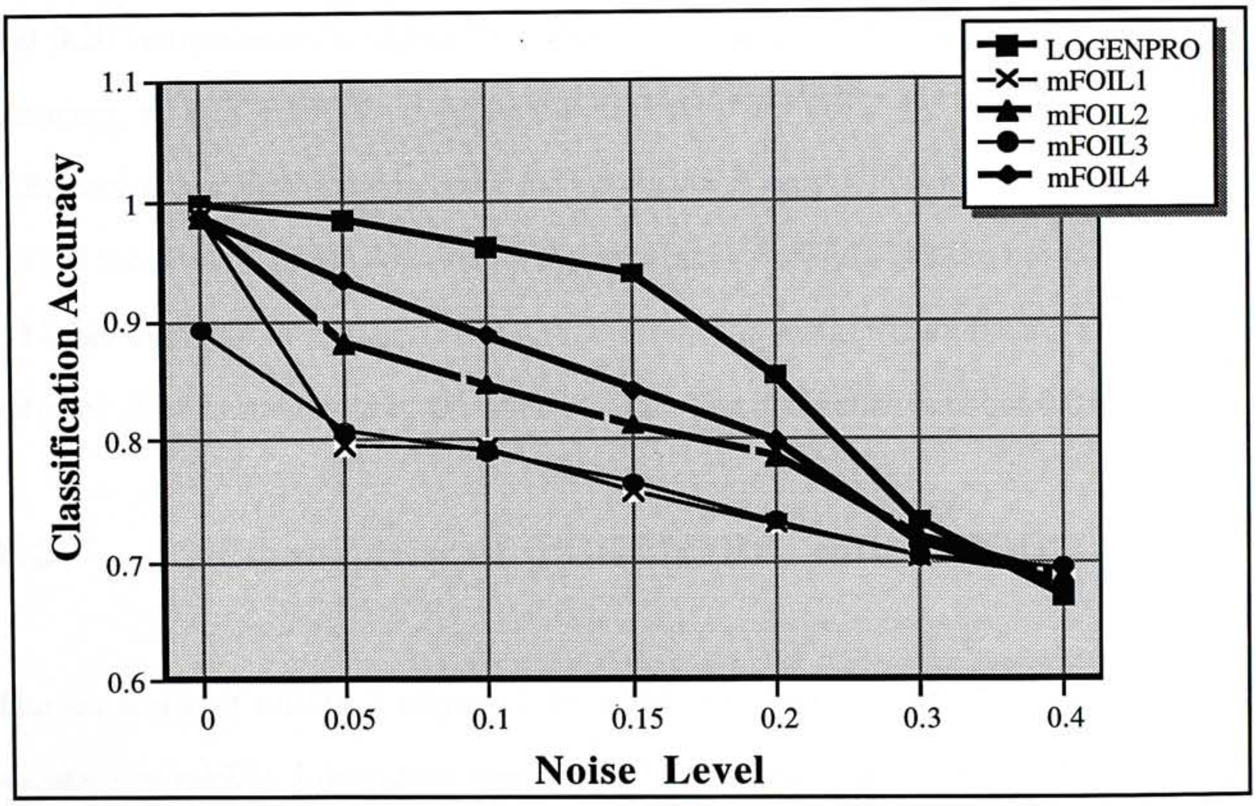


**Figure 7.3: Comparison between LOGENPRO, mFOIL1, mFOIL2, mFOIL3, and mFOIL4**

The accuracy of mFOIL2 (figure 7.3) decreases smoothly between the noise levels 0.0 to 0.20. It drops from 0.985 to 0.785, which is a more than 20% reduction. The decrement slightly slows down between the noise levels of 0.20 and 0.40. It drops from 0.785 to 0.685, with a 12.7% reduction. The results of the one-tailed paired *t*-test are listed as follows:

| Noise Level | 0.00 | 0.05 | 0.10 | 0.15 | 0.20 | 0.30 | 0.40 |
|---|---|---|---|---|---|---|---|
| t-statistics | 3.03E+08 | 35.38 | 17.29 | 14.98 | 5.15 | 1.11 | -3.37 |

Since t-statistics at the 0.00 noise level is very large, we can assert that the classification accuracy of LOGENPRO is significantly higher than that of mFOIL2. The difference is about 0.01. The t-statistics at the 0.05 noise level is 35.38 which is greater than the critical value of 4.78. Thus, the classification accuracy of LOGENPRO at this noise level is significantly higher than that of mFOIL2 with the difference of 0.1. Similarly, the classification accuracy of LOGENPRO at the noise levels between 0.10 and 0.20 is significantly higher than that of mFOIL2. At the noise level of 0.30, the accuracy of LOGENPRO is higher than that of mFOIL2 by about 0.014, but the difference is not significant because the t-statistics is only 1.11 which is smaller than the critical threshold. On the other hand, the accuracy of mFOIL2 at the noise level of 0.40 is higher than that of LOGENPRO, the difference is insignificant because the absolute value of -3.37 is smaller than the critical value. The difference is about 0.015.

### 7.2.7. Comparison of LOGENPRO with mFOIL3

The accuracy of mFOIL3 (figure 7.3) at the noise level of 0.00 is only 0.892. The accuracy of mFOIL5 decreases smoothly when the noise levels between 0.0 and 0.15. It drops from 0.892 to 0.765, with a more than 14.5% reduction. The accuracy reduces slightly between the noise levels of 0.15 and 0.40. It drops from 0.765 to 0.693, with a more than 9% reduction. The results of the one-tailed paired *t*-test are listed as follows:

| Noise Level | 0.00 | 0.05 | 0.10 | 0.15 | 0.20 | 0.30 | 0.40 |
|---|---|---|---|---|---|---|---|
| t-statistics | NA | 32.20 | 20.31 | 17.06 | 8.15 | 2.06 | -4.90 |

The t-statistics at 0.00 noise level is not available because the variances are very small (near zero). The t-statistics at the 0.05 noise level is 32.20 which is greater than the critical value of 4.78. Thus, the classification accuracy of LOGENPRO is significantly higher than that of mFOIL3. The difference is about 0.17. Similarly, the classification accuracy of LOGENPRO at the noise levels between 0.10 and 0.20 is significantly higher than that of mFOIL3. At the noise level of 0.30, the accuracy of LOGENPRO is higher than that of mFOIL3, but the difference is not significant. On the other hand, the accuracy of mFOIL3 at the noise level of 0.40 is significantly higher than that of LOGENPRO. The difference is about 0.02.

## 7.2.8. Comparison of LOGENPRO with mFOIL4

The accuracy of mFOIL4 (figure 7.3) decreases slightly when the noise level increases from 0.0 to 0.15. It drops from 0.985 to 0.842, with a more than 14% reduction. The accuracy reduces smoothly between the noise levels of 0.15 and 0.40. It drops from 0.842 to 0.680, with a more than 19% reduction. The results of the one-tailed paired *t*-test are listed as follows:

| **Noise Level** | **0.00** | **0.05** | **0.10** | **0.15** | **0.20** | **0.30** | **0.40** |
|---|---|---|---|---|---|---|---|
| t-statistics | 3.03E+08 | 21.59 | 13.05 | 9.95 | 4.37 | 1.23 | -1.65 |

The classification accuracy of LOGENPRO at the noise level 0.00 is significantly higher than that of mFOIL4. The difference is about 0.01. The t-statistics at the 0.05 noise level is 21.59 which is greater than the critical value of 4.78. Thus, the classification accuracy of LOGENPRO is significantly higher than that of mFOIL4. The difference is about 0.05. Similarly, the classification accuracy of LOGENPRO at the noise levels between 0.10 and 0.15 is significantly higher than that of mFOIL4. At the noise levels of 0.20 and 0.30, the accuracy of LOGENPRO is higher than that of mFOIL4, but the differences are not significant. On the other hand, the accuracy of mFOIL4 at the noise level of 0.40 is higher than that of LOGENPRO, but the difference is insignificant because the absolute value of -1.65 is smaller than the critical value. The difference is about 0.01.

### 7.2.9. Comparison of LOGENPRO with mFOIL5

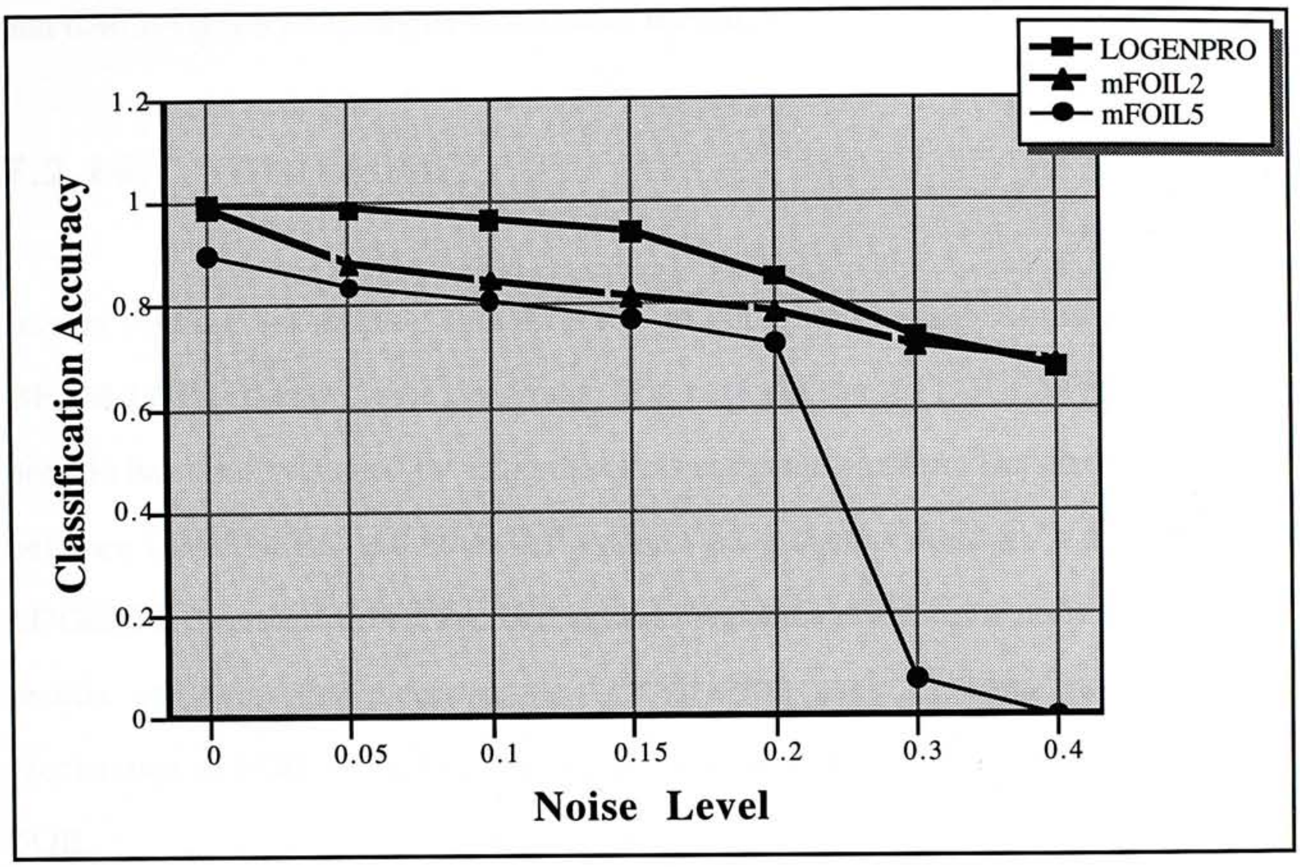


**Figure 7.4: Comparison between LOGENPRO, mFOIL2, and mFOIL5.**

The accuracy of mFOIL5 at the noise levels of 0.00, 0.30, and 0.40 is not acceptable. By comparing mFOIL5 with mFOIL2 (figure 7.4) , we can conclude that the significance threshold for noise-handling affects the performance of mFOIL severely (see table 7.8). The accuracy of mFOIL5 decreases slowly from the noise levels of 0.0 to 0.2. It drops from 0.896 to 0.723, a more than 19% reduction. There is a sudden drop in accuracy from 0.723 at the noise level of 0.20 to 0.0 at the noise level of 0.40. The results of the one-tailed paired *t*-test are listed as follows:

| Noise Level | 0.00 | 0.05 | 0.10 | 0.15 | 0.20 | 0.30 | 0.40 |
|---|---|---|---|---|---|---|---|
| t-statistics | NA | 16.99 | 22.29 | 16.44 | 8.12 | 10.23 | 176.37 |

The t-statistics at the 0.00 noise level is not available because the variances are very small (near zero). The t-statistics at the 0.05 noise level is 16.99 which is greater than the critical value of 4.78. Thus, the classification accuracy of LOGENPRO at this noise level is significantly higher than that of mFOIL5. The difference is about 0.15. Similarly, the classification accuracy of LOGENPRO at the noise levels between 0.10 and 0.40 is significantly higher than that of mFOIL5.

### 7.2.10. Discussion

In this section, we employ LOGENPRO to combine evolutionary algorithms and BEAM-FOIL, to learn logic programs. The performance of LOGENPRO in a noisy domain has been evaluated by using the chess endgame problem. Detailed comparisons between LOGENPRO and other ILP systems have been conducted. It has found that LOGENPRO outperforms these ILP systems significantly at most noise levels. These results are surprising because the LOGENPRO uses the same noise-handling mechanism of FOIL by initializing the population with programs created by BEAM-FOIL.

One possible explanation of the better performance of LOGENPRO is that the Darwinian principle of survival and reproduction of the fittest is a good noise handling method. It avoids overfitting noisy examples, but at the same time, it finds interesting and useful patterns from these noisy examples. This result is very encouraging and we plan to apply LOGENPRO to combine evolutionary algorithms with other learning systems such as GOLEM (Muggletion and Feng 1990), LINUS (Lavrac and Dzeroski 1994), and mFOIL (Lavrac and Dzeroski 1994) for solving problem.

## 7.3. Learning programs in Fuzzy Prolog

The goal of this experiment is to induce a Fuzzy Prolog program that describes the fuzzy relation `can-reach` intensionally. The set of training examples and the background knowledge are stored in a fuzzy relational database. Li and Liu (1990) described the detailed definitions of the syntax and semantics of Fuzzy Prolog and the properties of fuzzy relational databases. To the knowledge of the authors, LOGENPRO is currently the only system that can learn programs in Fuzzy Prolog.

Consider the fuzzy network in figure 7.5, and this network represents the fuzzy relation `linked-to(X, Y)` that denotes node `X` is directly linked to node `Y` with a truth value f, where f ∈ (0, 1]. In the network, the edges represent the instances of the `linked-to` relation and the number on an edge is the truth value of the corresponding instance. For example, the truth value of the instance `linked-to(0, 1)` is 0.9.

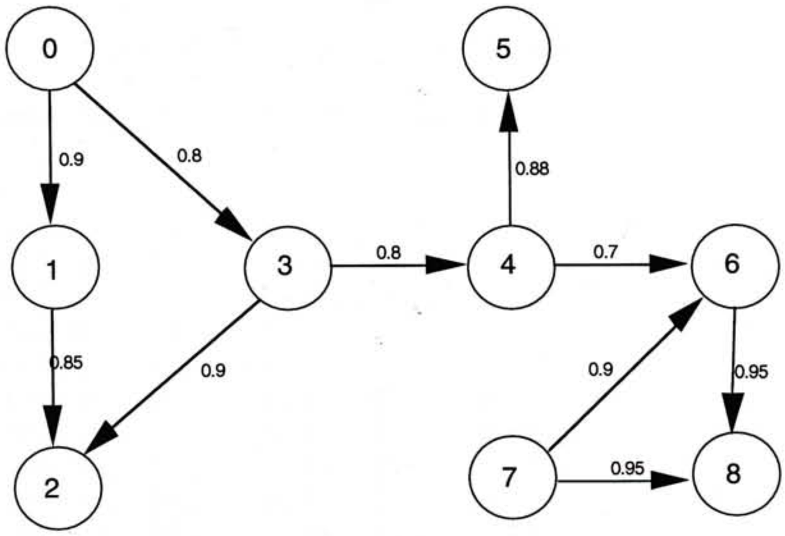


**Figure 7.5: A fuzzy network**

A fuzzy relation is associated with a n-ary fuzzy predicate and can be described extensionally as a set of ordered pairs. The extensional representations of fuzzy relations can be stored in a fuzzy relational database. For example, the fuzzy relation `linked-to(X, Y)` can be stored in a database as:

```
linked-to(X, Y) = {(<0,1>, 0.9), (<0,3>, 0.8), (<1,2>, 0.85),
                   (<3,2>, 0.9),(<3,4>, 0.8), (<4,5>, 0.88),
                   (<4,6>, 0.7), (<6,8>, 0.95), (<7,6>, 0.9),
                   (<7,8>, 0.95) }
```

The first element of an ordered pair is a n-tuple of constants that satisfies the associated fuzzy predicate. The second element of the ordered pair is the corresponding truth value.

Other fuzzy relations can be obtained from the fuzzy network. One of them is `can-reach(X, Y)` which is represented explicitly as:

```
can-reach(X, Y) = {(<0,1>, 0.9), (<0,2>, 0.765), (<0,3>, 0.85),
                   (<0,4>, 0.72), (<0,5>, 0.648), (<0,6>, 0.567),
                   (<0,8>, 0.510), (<1,2>, 0.85), (<3,2>, 0.9)
                   (<3,4>, 0.8), (<3,5>, 0.72), (<3,6>, 0.63),
                   (<3,8>, 0.567), (<4,5>, 0.88), ((4,6>, 0.7)
                   ((4,8>, 0.63), (<6,8>, 0.95), (<7,6>, 0.9),
                   (<7,8>, 0.95) }
```

The negative instances of this relation can be found using the close world assumption (Li and Liu, 1990). Thus, the set of negative instances is:

```
{(<0,0>, 0), (<0,7>, 0), (<1,0>, 0), (<1,1>, 0), (<1,3>, 0),
 (<1,4>, 0), (<1,5>, 0), (<1,6>, 0), (<1,7>, 0), (<1,8>, 0),
 (<2,0>, 0), (<2,1>, 0), (<2,2>, 0), (<2,3>, 0), (<2,4>, 0),
 (<2,5>, 0), (<2,6>, 0), (<2,7>, 0), (<2,8>, 0), (<3,0>, 0),
 (<3,1>, 0), (<3,3>, 0), (<3,7>, 0), (<4,0>, 0), (<4,1>, 0),
 (<4,2>, 0), (<4,3>, 0), (<4,4>, 0), (<4,7>, 0), (<5,0>, 0),
 (<5,1>, 0), (<5,2>, 0), (<5,3>, 0), (<5,4>, 0), (<5,5>, 0),
 (<5,6>, 0), (<5,7>, 0), (<5,8>, 0), (<6,0>, 0), (<6,1>, 0),
 (<6,2>, 0), (<6,3>, 0), (<6,4>, 0), (<6,5>, 0), (<6,6>, 0),
 (<6,7>, 0), (<7,0>, 0), (<7,1>, 0), (<7,2>, 0), (<7,3>, 0),
 (<7,4>, 0), (<7,5>, 0), (<7,7>, 0), (<8,0>, 0), (<8,1>, 0),
 (<8,2>, 0), (<8,3>, 0), (<8,4>, 0), (<8,5>, 0), (<8,6>, 0),
 (<8,7>, 0), (<8,8>, 0)}
```

The 19 positive and 62 negative instances are used as the fitness cases. The fitness function finds the sum, taken over all 81 fitness cases, of the absolute values of the difference between the desired truth value and the truth value returned by the generated program. A fitness case is said to be covered by a program if the truth value returned is within 0.05 of the desired value. LOGENPRO terminates if the maximum number of generations of 25 is reached or a Fuzzy Prolog program that covers all fitness cases is found. The logic grammar for this problem is shown in table 7.11. In this grammar, the background knowledge is represented by the fuzzy relation `linked-to(X, Y)` stored in a fuzzy relational database and the predicate `random(0, 1, ?A)` is a logic goal.

```
start              ->     clauses.
clauses            ->     clauses, clauses.
clauses            ->     clause.
clause             ->     {random(0, 1, ?A)},
                          consq, [:-(?A)], antes, [.].
consq              ->     [can-reach(X, Y)].
antes              ->     antes, [,], antes.
antes              ->     ante.
ante               ->     {member(?A,[W, X, Y, Z])},
                          {member(?B,[W, X, Y, Z])},
                          literal(?A, ?B).
literal(?A, ?B)    ->     [ linked-to(?A, ?B) ].
literal(?A, ?B)    ->     [ can-reach(?A, ?B) ].
```

**Table 7.11: The logic grammar for inducing programs in Fuzzy Prolog**

A number of trials have been performed using a population size of 100. Correct programs can be found in all trials. The following correct and simplified program is found in one trial:

```
can-reach(X, Y)       :- (1)     linked-to(X, Y).
can-reach(X, Y)       :- (0.9)   linked-to(X, Z),
                                 can-reach(Z, Y).
```

# Chapter 8
# An Adaptive Inductive Logic Programming System

In chapter 4, we have described the Genetic Logic Programming System (GLPS) that employs evolutionary algorithms to induce logic programs. In chapters 5, 6, and 7, we have discussed LOGENPRO and demonstrated that LOGENPRO outperforms other ILP systems for learning logic programs in a noisy domain.

However, LOGENPRO and other ILP systems cannot improve themselves automatically. In this chapter, we describe an Adaptive Inductive Logic Programming (Adaptive ILP) system that evolves using evolutionary algorithms. The definition of adaptive inductive logic programming is formulated in the first section. We present a generic top-down ILP algorithm in section 8.2. A meta-level learner that induces search bias is described in section 8.3. Section 8.4 delineates the experimentation and some evaluations of the system followed by a discussion.

## 8.1. Adaptive Inductive Logic Programming

As described chapter 3, an ILP system is a relational concept learner that induces a new relation for the target concept (i.e., the target predicate) from training examples and known relations from the background knowledge **B**. The training examples, the hypothesis space and the background knowledge are represented in first-order Horn clause languages (Muggleton and Feng 1990).

An Adaptive Inductive Learning Programming (Adaptive ILP) system is an ILP system that can improve itself on the learning capability. It maintains various sets of background knowledge and biases. It improves itself by modifying its biases and background knowledge. A hypothesis space for learning is defined through the concept description language, the language bias and the background knowledge. Therefore, by changing the language bias and the background knowledge, the size and structure of the hypothesis space can be modified accordingly. The search strategy and heuristics are changed if the system's search biases are modified. Here, we formulate the task of an Adaptive ILP system in table 8.1.

Given:

- -A set **E** of positive $\mathbf{E}^{+}$ and negative $\mathbf{E}^{-}$ training examples of the target predicate p. Training examples are represented as ground atoms
- -A concept description language **L**
- -A set of learning biases **BIASES**
- -A set of various background knowledge **BKs**

Find:

- -A modified set of learning biases **BIASES'**
- -A modified set of background knowledge **BKs'**
- -A concept definition **H** for the target predicate p expressible in **L** such that **H** is complete and consistent with respect to (w.r.t.) the training examples **E** and a background knowledge **B'** in **BKs'**

**H** is complete if every positive example $e^{+}$ in $\mathbf{E}^{+}$ is covered by **H** w.r.t. the background knowledge **B**. i.e.
$\mathbf{B} \cup \mathbf{H} \models e^{+}$

**H** is consistent if no negative example $e^{-}$ in $\mathbf{E}^{-}$ is covered by **H** w.r.t. the background knowledge **B**. i.e.
$\mathbf{B} \cup \mathbf{H} \not\models e^{-}$

**Table 8.1: The definition of Adaptive ILP**

The logical organization of our adaptive ILP system is depicted in figure 8.1. Its components are introduced as follows:

(1) *External interface*: It provides a user-friendly interface between the system and users. It accepts training examples, a set **BKs** of background knowledge, and a set **BIASES** of biases and transfers them through the learning controller to the *example database*, *BKbase* and *biases base* respectively. The interface also provides commands for

users to query about the results of an adaptive learning task and to directly control the operations of the learning controller.

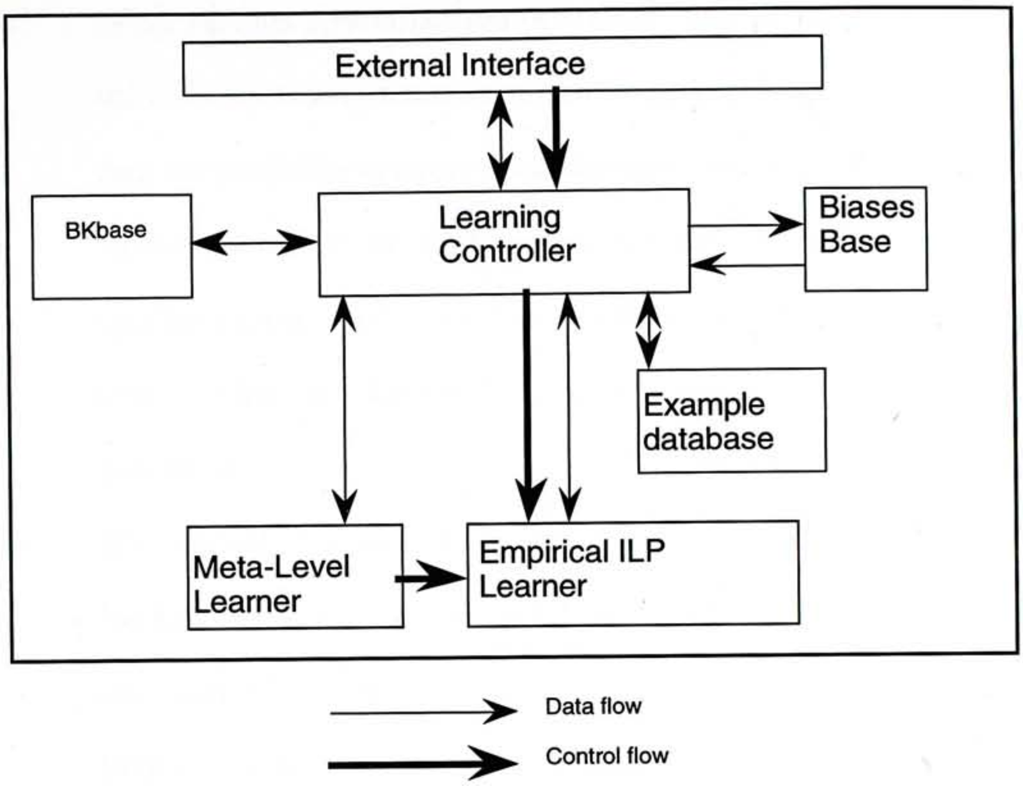


**Figure 8.1: The logical organization of an adaptive ILP system**

(2) *Biases base*: It is a knowledge base that stores all learning biases. Biases can be retrieved, added, deleted, and modified through the interface of this knowledge base.

(3) *BKbase*: It stores various background learning knowledge that can be used in inductive learning. Background knowledge can be retrieved, added, deleted, and modified through the interface of *BKbase*. Since each entity of it is in fact a complex structure representing background knowledge, *BKbase* is implemented using object-oriented techniques.

(4) *Examples database*: It stores the training examples.

(5) *Empirical ILP learner*: It induces a logic program from the training examples, given a concept description language, a specific background knowledge, a search bias and a language bias. A search of the hypothesis space can be performed bottom-up or top-down. Bottom-up

techniques start from the training examples and search the space by employing various generalization operators. Top-down techniques start from the most general concept descriptions, and search the space by using various specialization operators. Top-down techniques are better suited for learning from imperfect examples because a large number of data are available in every specialization step and the system can employ various statistical techniques to decide how to perform the specialization. Moreover, top-down search can easily be guided by the search bias. In section 8.2, a generic top-down ILP algorithm is described.

(6) *Meta-level learner*: It learns search biases, language biases, and background knowledge. Search and language biases can be represented declaratively or procedurally. If biases are expressed in a first-order language, the problem of learning biases can be formulated as an empirical ILP problem and thus the empirical ILP system described in (5) can be used. In section 8.3, we apply LOGENPRO to implement a meta-level learner that induces procedural biases. Background knowledge can be modified by introducing new predicate definitions or adding the definition of the current target predicate. For the former case, if the introduced predicates can facilitate the learning of the current target predicate, the introduced predicates can be viewed as sub-concepts (or sub-functions). In sub-section 6.1.2, we showed that LOGENPRO can effectively induce sub-functions and thus it can be used for this purpose. The induced sub-concepts are remembered in order to improve the learning of similar predicates in the future. For the latter case, the induced definition of the current target predicate is stored to facilitate the learning of higher level predicates. It can also learn other meta-knowledge such as the conditions under which various learning biases and background knowledge can be employed. Since the meta-level

learner performs a variety of learning tasks, it is implemented as a multistrategy learning system.

(7) *Learning controller*: It is a knowledge-based system that controls the empirical ILP learner and the meta-level learner. The knowledge used by the learning controller can be updated by the meta-level learner.

## 8.2. A generic top-down ILP algorithm

This section presents a generic top-down ILP algorithm based on FOIL (Quinlan 1990; 1991). The algorithm is depicted in table 8.2. The algorithm consists of three steps. In the pre-processing step, missing argument values in training examples are handled by assigning default or random values to them. A training example will be removed if it has too many missing values. If there are no or inadequate negative examples in the training set, they can be generated. Different ways of creating negative examples have been proposed (Lavrac and Dzeroski 1994).

The second step performs the construction of a program. This step employs four local variables: $\mathbf{E}_{\mathtt{current}}$ (Current training examples set), $\mathbf{E}'_{\mathtt{current}}$ (Updated training examples set), $\mathbf{P}$ (Current program) and $\mathbf{P}'$ (Modified program). The main component of this step is the covering loop which implements Michalski's covering algorithm (Michalski et al. 1986a). The covering loop construct a program by iteratively executing the following sub-steps:

(a) Construct a clause that covers some positive examples in $\mathbf{E}_{\mathtt{current}}$.

(b) Append the clause to the current program $\mathbf{P}$ and generate a modified program $\mathbf{P}'$.

(c) Remove all positive examples from $\mathbf{E}_{\mathtt{current}}$ which are covered by $\mathbf{P}'$ with respect to the background knowledge $\mathbf{B}$.

```
Input:
   E:                   Training examples
   L:                   The concept description language
   $BIAS_{search}$:     The search bias
   $BIAS_{lang}$:       The language bias
   B:                   Background knowledge
   T:                   The target concept

Output:
   A program P which contains a set of program clauses. Each  clause
   C ∈ L.


Function ILP(E, L, $BIAS_{search}$, $BIAS_{lang}$, B, T)

(1) Pre-processing of the training examples E and producing a
    modified set of examples $E'$: $E'$ := Preprocessing(E).

(2) Let $E_{current}$ := $E'$;
    Let P := {};
    Repeat
       -Let C := T ←;
       -Find a specialization $C'$ of C. This step constructs a
        clause $C'$ from C by calling Clause-Construct(C,
        $E_{current}$, B, L,  $BIAS_{search}$, $BIAS_{lang}$);
       -If a specialization can be found
          -Add $C'$ to P to produce a new program $P'$. i.e.
           $P'$ := P ∪ {$C'$};
          -Remove all positive examples covered by $P'$ from
           $E_{current}$ to get an updated training set $E'$
           $E'_{current}$ := $E_{current}$ - { positive examples in
           $E_{current}$ covered by $P'$ w.r.t. the background
           knowledge B};
          -Let $E_{current}$ := $E'_{current}$;
          -Let P := $P'$
        Else
          -Set the flag *No-More-Improvement* to true;
    Until
      The Covering termination criterion is satisfied. i.e.
      covering-termination(P, *No-More-Improvement*, $E_{current}$, B)
      returns true;

(3) Post-processing the program P and producing $P'$. i.e.
    $P'$ := Post-processing(P);
    Return($P'$);
```

**Table 8.2: A generic top-down ILP algorithm**

The covering loop terminates if the terminating conditions are satisfied. A typical condition is that either all positive examples are covered or no more improvement can be achieved by searching for a new clause. The final step attempts to improve the accuracy of the program induced when classifying unseen examples and to simplify the program.

The covering loop calls the 'Clause-Construct' function which is the core of the generic algorithm. The function constructs a clause $\mathtt{C}_n = \mathtt{T} \leftarrow \mathtt{l}_1, \mathtt{l}_2, \ldots, \mathtt{l}_n$ starting from the most general clause $\mathtt{C}_0 = \mathtt{T} \leftarrow$ with an empty body. A sequence of clauses $\mathtt{C}_0$, $\mathtt{C}_1$, $\mathtt{C}_2$, $\mathtt{C}_3$, ...., $\mathtt{C}_n$ are generated by a number of specialization steps. At each step, the current clause $\mathtt{C}_i = \mathtt{T} \leftarrow \mathtt{l}_1, \mathtt{l}_2, \ldots, \mathtt{l}_i$ is refined by appending a specific literal $\mathtt{l}_j$ to its body. A literal $\mathtt{l}_j$ is constructed from the background knowledge **B** restricted by the concept description language **L** and language bias $\mathtt{BIAS}_{\mathtt{lang}}$. The language may limit $\mathtt{l}_j$ to be function-free while $\mathtt{BIAS}_{\mathtt{lang}}$ may prevent new variable to be introduced in $\mathtt{l}_j$. The aim of the procedure is to find a clause which covers most positive examples while excludes all or most negative examples. In a hill-climbing search, the procedure keeps the current best clause and refines it using the estimated best specialization at each step, until the stopping condition is satisfied. A hill-climbing 'Clause-Construct' algorithm is presented in table 8.3.

The 'Clause-Construct' function calls the 'Find-Extension' function to find the extension $\mathbf{E}_i$ of the current training examples given the partially developed clause $\mathtt{C}_i = \mathtt{T}(\mathtt{X}_1, \mathtt{X}_2, \ldots, \mathtt{X}_n) \leftarrow \mathtt{l}_1, \mathtt{l}_2, \ldots, \mathtt{l}_i$ and the background knowledge **B**. Each training example $<\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n>$ is a *n*-tuple where $\mathtt{x}_i$, $1 \leq i \leq n$, are some constants. To find the extension, the function initializes a clause $\mathtt{C}_0 = \mathtt{T}(\mathtt{X}_1, \mathtt{X}_2, \ldots, \mathtt{X}_n)$, then the literal $\mathtt{l}_1$ is added to the body of $\mathtt{C}_0$ to produce a new clause $\mathtt{C}_1$. The literal $\mathtt{l}_1$ is either of the form $\mathtt{X}_j = \mathtt{X}_k$, $\mathtt{X}_j \neq \mathtt{X}_k$, $\mathtt{p}_m(\mathtt{Y}_1, \mathtt{Y}_2, \ldots, \mathtt{Y}_{s_m})$ or $\mathtt{not}\ \mathtt{p}_m(\mathtt{Y}_1, \mathtt{Y}_2, \ldots, \mathtt{Y}_{s_m})$.

```
Input:
   C:                 An initial clause C = T←
   $\mathbf{E}_{current}$:         The current training examples
   $\mathbf{B}$:                 Background knowledge
   $\mathbf{L}$:                 The concept description language
   $BIAS_{search}$:      The search bias
   $BIAS_{lang}$:        The language bias
Output:
   A clause that covers some positive examples in $\mathbf{E}_{current}$ while
   excludes all or most negatives examples in $\mathbf{E}_{current}$

Function Clause-Construct(C, $\mathbf{E}_{current}$, $\mathbf{B}$, $\mathbf{L}$, $BIAS_{search}$,
                              $BIAS_{lang}$)

There is a scoring function stored in $BIAS_{search}$, save this
function to scoring;
Repeat
   -Set BEST to a bad literal such as X = X where X is a
    variable appearing in the head of the clause;
   -Set Best-score to 0;
   -Find the extension $\mathbf{E}_i$ of $\mathbf{E}_{current}$ using the clause C w.r.t.
    $\mathbf{B}$. i.e. $\mathbf{E}_i$ := Find-Extension(C, $\mathbf{E}_{current}$, $\mathbf{B}$);
   -Let $n_i^+$ be the number of positive tuples in $\mathbf{E}_i$;
   -Let $n_i^-$ be the number of negative tuples in $\mathbf{E}_i$;
   -Current-information := $-\log_2(n_i^+ / (n_i^+ + n_i^-))$;
   -For all literal l from $\mathbf{B}$ that satisfy the constraints
    imposed by the language $\mathbf{L}$ and bias $BIAS_{lang}$
      -Set $C' = C \cup \{l\}$;
      -Find the extension $\mathbf{E}_{i+1}$ of $\mathbf{E}_{current}$ using the clause $C'$
       i.e. $\mathbf{E}_{i+1}$ := Find-Extension($C'$, $\mathbf{E}_{current}$, $\mathbf{B}$);
      -Let $n_{i+1}^+$ be the number of positive tuples in $\mathbf{E}_{i+1}$;
      -Let $n_{i+1}^-$ be the number of negative tuples in $\mathbf{E}_{i+1}$;
      -Let the number of positive tuples in $\mathbf{E}_i$ that have been
       represented by one or more tuples in $\mathbf{E}_{i+1}$ be $n_i^{++}$;
      -Find the score of the literal l by using the scoring
       function i.e. literal-score := scoring ($n_i^{++}$, $n_{i+1}^+$, $n_{i+1}^-$,
       Current-information);
      -If literal-score > Best-score then
         -BEST := l;
         -Best-score := literal-score;
   -If BEST == X=X then
      -No-More-Improvement := true;
    Else
      -Append BEST to the body of C;
Until Clause-Termination(C, No-More-Improvement, $\mathbf{E}_{current}$, $\mathbf{B}$)
      is true;
Post-processing the clause C to find an improvement i.e.
$C'$ := Find-Improvement(C);
If Acceptable($C'$)
   -Return($C'$);
Else
   -Return(No-Specialization-Can-Be-Found);
```

**Table 8.3: A hill-climbing 'Clause-Construct' algorithm**

If the literal contains *k* new variables, the arity of each tuple in the generated training set $\mathbf{E}_1$ increases to $(n + k)$. $\mathbf{E}_1$ can be found by performing a natural join of $\mathbf{E}_{\mathtt{current}}$ with the relation corresponding to literal $\mathtt{l}_1$. The process is repeated for literals $\mathtt{l}_2$, $\mathtt{l}_3$, ..., $\mathtt{l}_i$ until the extension $\mathbf{E}_i$ is found.

The most important component of the hill-climbing 'Clause-Construct' algorithm is the 'scoring' function that estimates the performance of each literal. An accurate estimation directs the search towards the global maxima while a misleading one traps the system into local-maxima. By providing different 'scoring' functions to the generic ILP algorithm, various learning algorithms can be generated. The performances of a good and a bad learners can be significant different as shown in section 8.3.

## 8.3. Inducing procedural search biases

In this section, LOGENPRO is used in the meta-level learner to induce procedural search biases (i.e. the 'scoring' function). In order to employ LOGENPRO, a logic grammar must be defined. It is depicted in table 8.4.

In the grammar, the terminal symbols `n-pos-i-plus-1`, `n-neg-i-plus-1`, and `n-pos-i` represent respectively $n_{i+1}^{+}$, $n_{i+1}^{-}$ and $n_i^{++}$. With reference to the algorithms in tables 8.2 and 8.3, assume that $\mathbf{E}_i$ is the extension of current training examples $\mathbf{E}_{\mathtt{current}}$ by current clause $\mathtt{C}_i$, $n_i^{+}$ and $n_i^{-}$ are respectively the number of positive and negative tuples in $\mathbf{E}_i$. $\mathbf{E}_i$ can be extended by using the literal l to $\mathbf{E}_{i+1}$. $n_{i+1}^{+}$ and $n_{i+1}^{-}$ are respectively the number of positive and negative tuples in $\mathbf{E}_{i+1}$. $n_i^{++}$ is the number of positive tuples in $\mathbf{E}_i$ that have been represented by one or more tuples in $\mathbf{E}_{i+1}$. The terminal symbol `current-information` is defined as $-\log_2(n_i^{+} / (n_i^{+} + n_i^{-}))$.

```
start          ->    function.
s-exp          ->    term.
s-exp          ->    function.

function       ->    [(], op1, s-exp, [)].
function       ->    [(], op2, s-exp, s-exp, [)].

op1            ->    [ protected-log ].
op2            ->    [ + ].
op2            ->    [ - ].
op2            ->    [ * ].
op2            ->    [ % ].
op2            ->    [ info ].

term           ->    [ n-pos-i-plus-1 ].
term           ->    [ n-neg-i-plus-1 ].
term           ->    [ n-pos-i ].
term           ->    [ current-information ].
term           ->    { random(-10, 10, ?a) },   [ ?a ].
```

**Table 8.4: A logic grammar for learning procedural search bias**

The terminal symbols +, -, and * represent functions that perform ordinary addition, subtraction, and multiplication respectively. The symbol % represents function that normally returns the quotient. However, if division by zero is attempted, the function returns 1.0. The symbol protected-log is a function that calculates the logarithm of the input argument x if x is larger than zero, otherwise it returns 1.0. The symbol info represents the basic function that calculates $-\log_2(X / (X+Y))$ given X and Y as inputs. The logic goal random(-10, 10, ?a) generates a random floating point number between -10 and 10 and instantiates ?a to the random number generated

### 8.3.1. The evolution process

The evolution process of the Adaptive ILP system is depicted in figure 8.2. Firstly, the Biases base is initialized with a population of different 'scoring' functions generated randomly using the logic grammar depicted in table 8.4. To estimate the fitness of a specific 'scoring' function, it is combined with the generic top-down ILP learner to produce a specific ILP learner. The performance of this ILP learner is then evaluated by using a fitness function. This measure is assigned as the fitness of the specific 'scoring' function. LOGENPRO employs crossover, selection, mutation, and other genetic operators to generate potentially better functions. The modified functions are stored in the Biases base and the whole evolution process iterates until the best function is found or no computational resource is available. Some induced functions (procedural search biases) are given in appendix A.

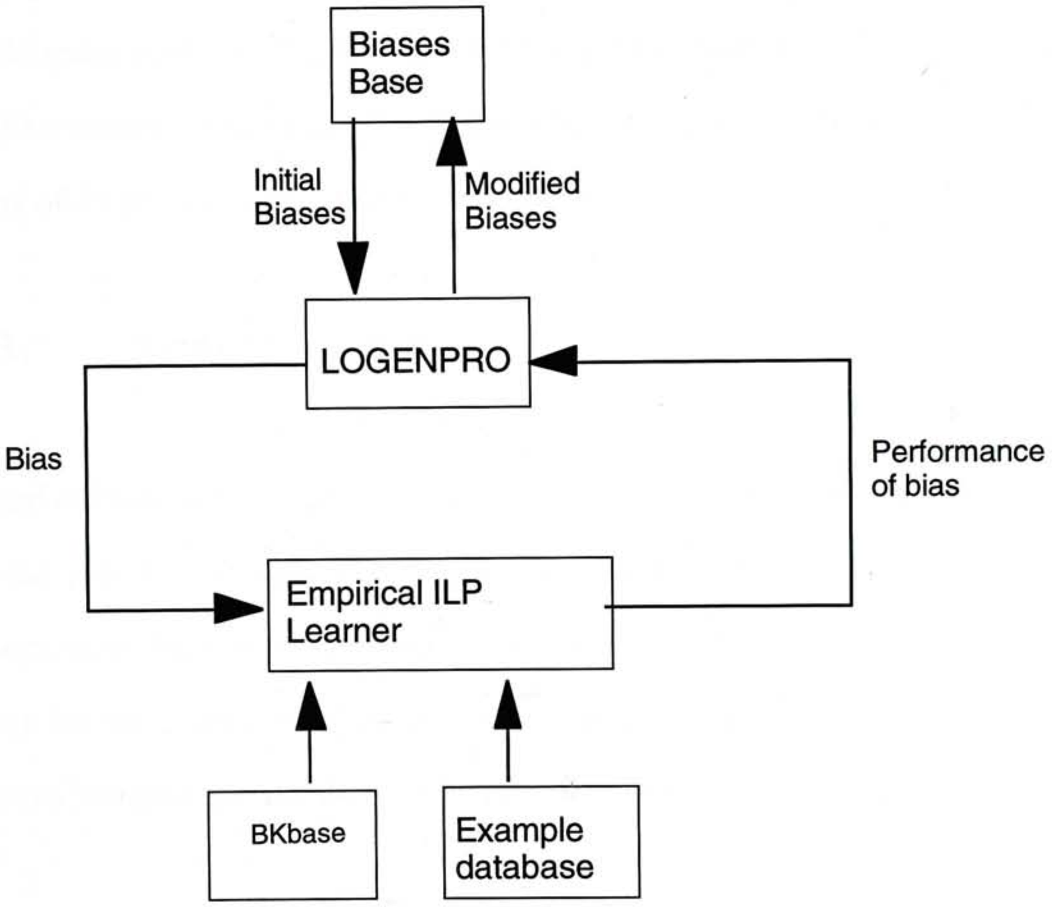


**Figure 8.2: The evolution process of the adaptive ILP system**

### 8.3.2. The experimentation setup

In this chapter, learning curves are used to estimate the performances of various learning systems. The example space is divided randomly into disjoint training and testing sets. The learner is trained on progressively larger portions of the training set and the performance of the induced logic program is estimated on the disjoint testing set. This process of dividing, training, and testing is repeated for 20 trials and the results are averaged to generate a learning curve.

As a running example, we use a traditional problem discussed in the literature (Muggleton and Feng 1990). In the problem of learning the list predicate *member*, the data consist of all lists of lengths 0 to 3 defined over three constants. The background knowledge **B** contains definitions of list construction predicates: *null* which holds for an empty list and *component* which decomposes a list into its head and tail. The example space contains 75 positive and 45 negative examples. The training sets contain 20 to 52 examples, one-half of each training set is positive examples. The testing set consists of 45 positive and 15 negative examples.

### 8.3.3. Fitness calculation

Adjusted and normalized fitness values are used as in Koza (1992). They are calculated from the raw fitness which is estimated by the fitness function. Various fitness functions have been tried and two of them are described here. The impact of fitness function on the generality of the evolved function is also demonstrated. The problem domain of learning the *member* predicate is used here.

For the first fitness function, a random set of 24 positive and 21 negative examples is used. A specific 'scoring' function is combined with the generic top-down ILP learner to produce a specific ILP learner called Adapted-ILP hereafter. Adapted-ILP induces a logic program using the random example set. The quality of the induced logic program is evaluated by counting the total number of misclassified examples from the same training set. This measure is used as the raw fitness of the specific 'scoring' function. Using this fitness function, only poor 'scoring' functions have been evolved. The learning curve of a poor learner is depicted in figure 8.3.

For the second fitness function, the raw fitness is developed in several steps. At the beginning of each generation, four instances of the learning task are created randomly from the member domain. Each learning task has a training and a disjoint testing data. The training set contains 20 positive and 20 negative examples. For each learning task, a specific Adapted-ILP induces a logic program from the training set and the logic program is evaluated by counting the number of misclassified examples from the testing set. The performance of the Adapted-ILP is the sum of numbers of misclassified examples for all learning tasks. This measure is then used as the raw fitness of the corresponding 'scoring' function. This fitness function can force the evolution of good 'scoring' functions. The learning curve of a good learner is shown in figure 8.3.

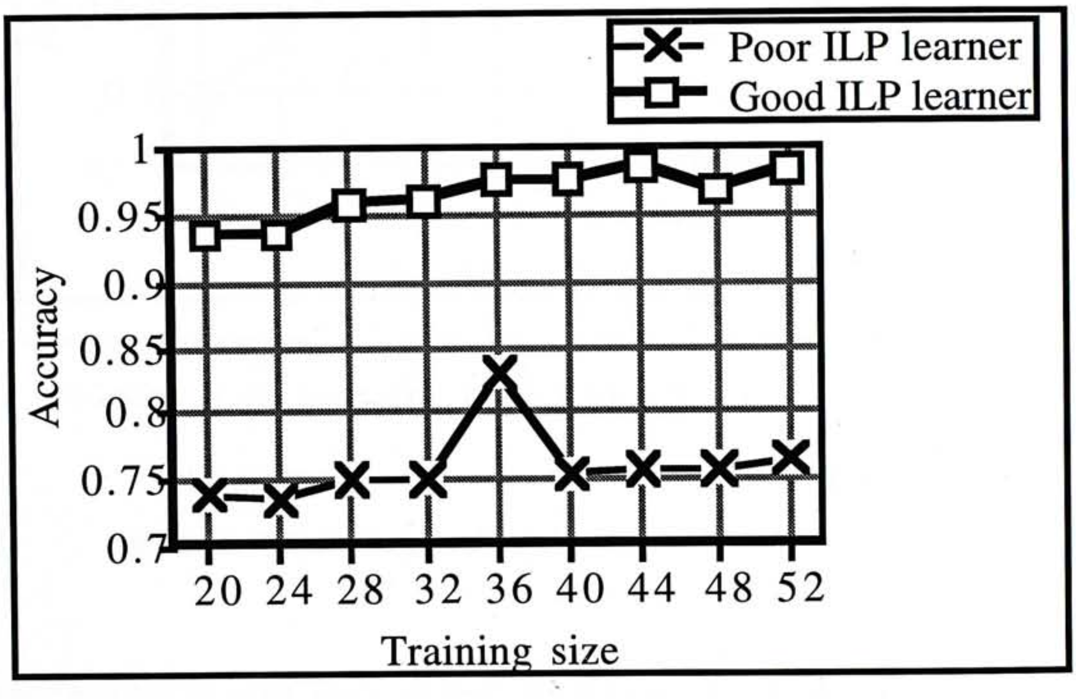


**Figure 8.3: The learning curves of good and poor ILP learner**

## 8.4. Experimentation and evaluations

This section compares the performance of the adaptive ILP system with that of FOIL which is a famous ILP system (Quinlan 1990). Standard learning tasks in the literature are used in these experiments (Quinlan 1990, Muggleton and Feng 1990).

### 8.4.1. The member predicate

The learning curves for this problem are depicted in figure 8.4. It is interested to find that the adaptive ILP system has higher accuracy than FOIL. The difference is significant at 5% level of significance when the training size is less than 36.

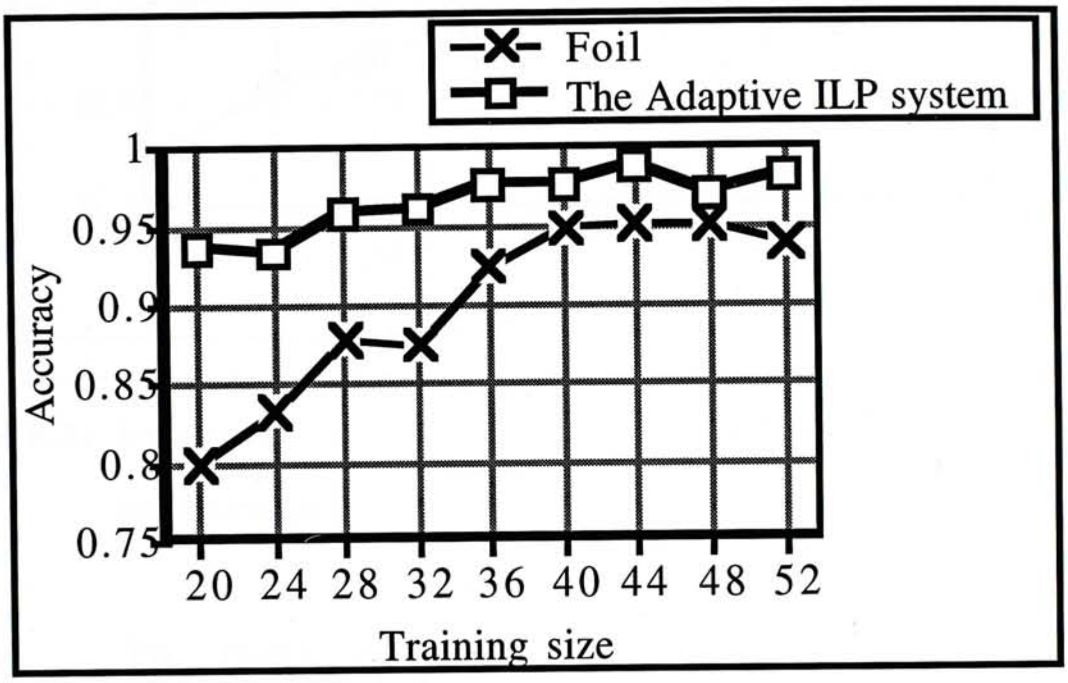


**Figure 8.4: Learning curves for the member problem**

### 8.4.2. The member predicate in a noisy environment

Difference amount of noise is introduced into the training examples in order to study the performances of both systems in learning programs in noisy environment. To introduce *n%* of noise into the examples, *n%* positive examples are labeled as negative ones while *n%* negative examples are labeled as positive ones. In this experiment, the percentages of introduced noise are 10% (0.1) and 40% (0.4). Their learning curves are summarized in figure 8.5. The adaptive ILP system performs better than FOIL at all noise level.

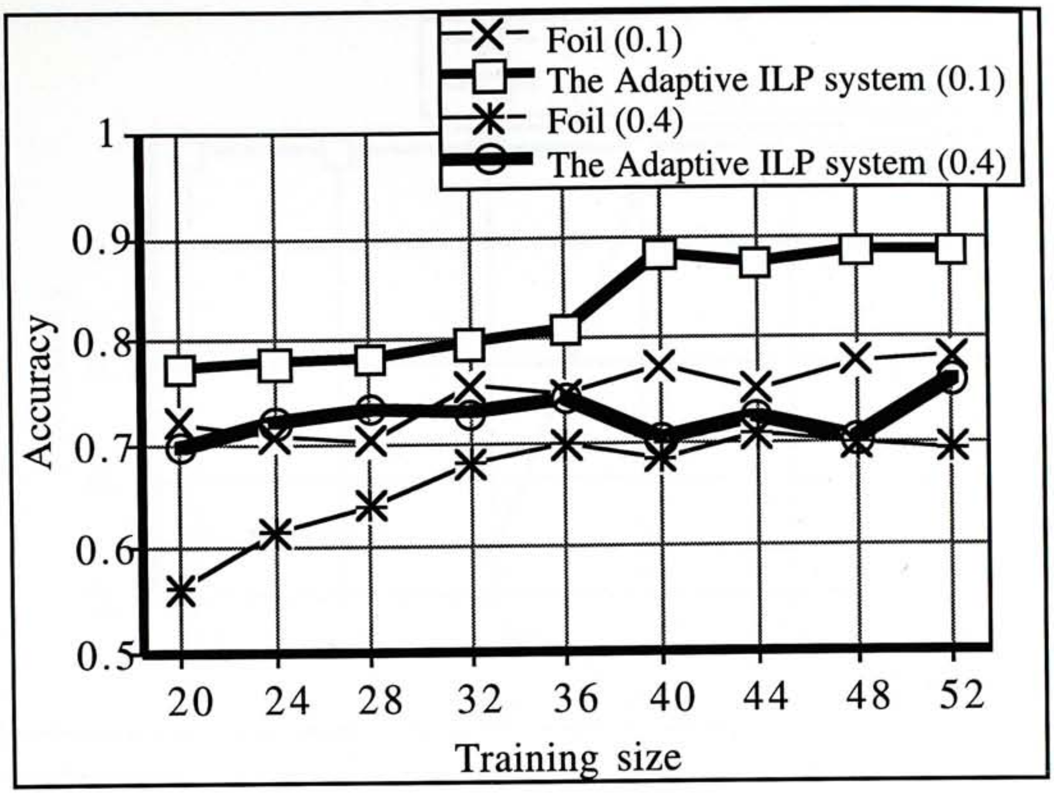


**Figure 8.5: Learning curves for the member problem in a noisy environment**

### 8.4.3. The multiply predicate

In the problem of learning the arithmetic predicate *multiply* (Muggletion and Feng 1990), the data contain integers in the range from zero to ten. The background knowledge is composed of definitions for arithmetic predicates *plus*, *decrement*, *zero*, and *one*. The example space has 73 positive and 1258 negative examples respectively. The training sets consist of 400 to 500 examples, one-tenth of each training set is positive and the remainder is negative. The learning curves for multiply are presented in figure 8.6. The Adaptive ILP system performs better than FOIL when the size of training set is less than 460. The difference is significant at 5% level of significance.

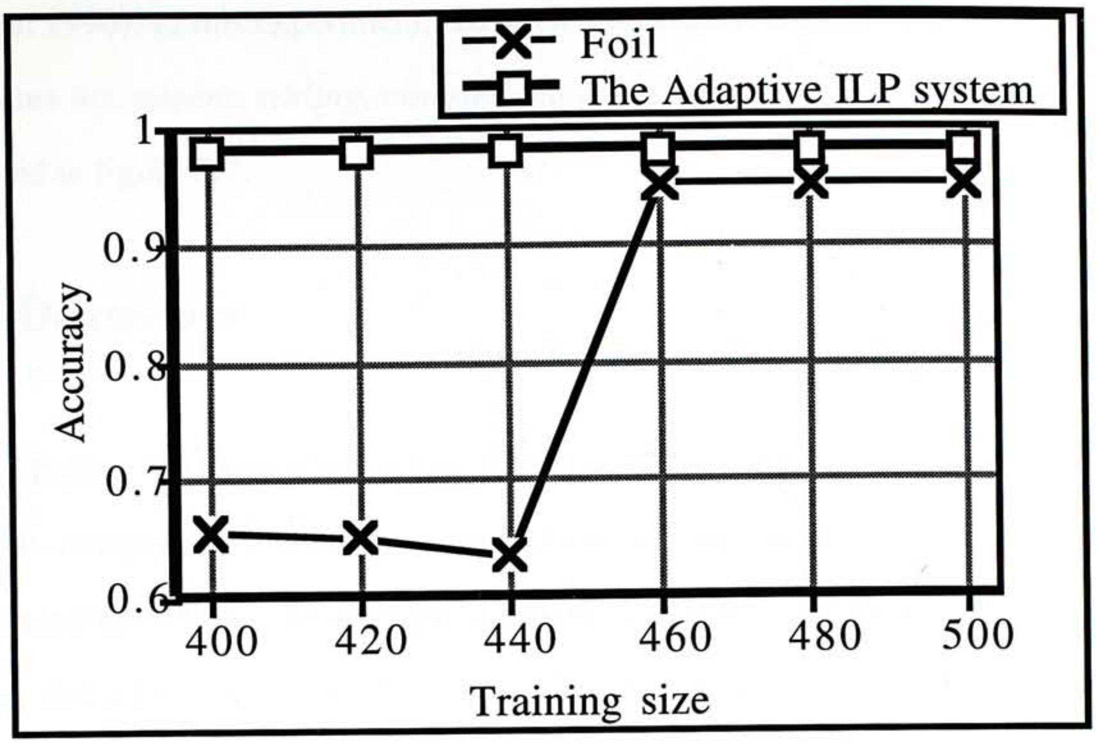


**Figure 8.6: Learning curves for the multiply problem**

### 8.4.4. The uncle predicate

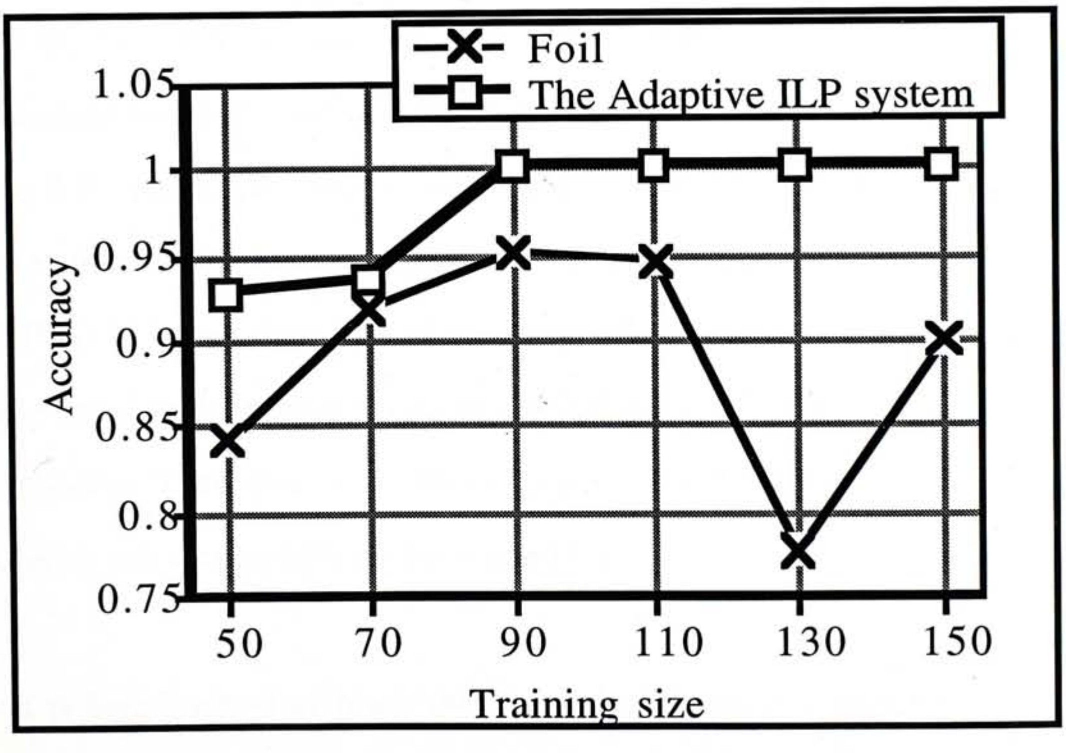


**Figure 8.7: Learning curves for the uncle problem**

Another traditional testbed for relational learners is the domain of family relationships (Quinlan 1990). In this experiment, the *uncle* predicate is induced and the background predicates are *parent*, *sibling*, *married*, *male*, and *female*. The learning curves are presented in figure 8.7.

## 8.5. Discussion

In this chapter, we have proposed an Adaptive Inductive Logic Programming system which is composed of an external interface, a biases base, a knowledge base of background knowledge, an example database, an empirical ILP learner, a meta-level learner, and a learning controller. In our implementation of the Adaptive Inductive Logic Programming system, the empirical ILP learner performs top-down search in the hypothesis space defined by the concept description language, the language bias and the background knowledge. The search is directed by search biases which can be induced and refined by LOGENPRO.

It has been demonstrated that the induced bias is better than that of FOIL on many standard learning tasks. From these experiments, it can be concluded that the Adaptive ILP system has superior learning ability compared to FOIL. Since they are different in their search biases only, the result implies that the search bias induced by LOGENPRO is better than that of FOIL for the learning problems. This result is surprising because the search biases of the Adaptive ILP system are initialized by a random process. These biases are normally poor, but the process of natural selection and evolution can successfully evolve a good bias.

It is important to mention that the induced search biases are rather general because they have reasonable performances on many traditional learning problems. For future work, in order to find a general, efficient, and effective bias, a large number of learning tasks of different kinds, such as the *member*, *append*, *quick sort*, *ackermann*, *uncle*, and *grandfather* problems, of various characteristics should be used. This adaptive learning approach, though computationally intensive, is rather exciting, as it opens up many opportunities for creating or improving learning algorithms.

# Chapter 9
# Conclusion and Future Work

## 9.1. Conclusion

The goal of program induction is to create computers that can learn to solve problems without being explicitly programmed. A means to achieve this goal is to allow computers to generate computer programs in different programming languages from specifications. The advantage of inducing computer programs rather than other high-level abstractions such as binary chromosomes, formal grammars, and semantic networks is that computer programs are flexible and executable. Computer programs are flexible because their sizes, shapes, and structural complexities are not restricted in advance. In contrast, these properties of programs are emerged during the learning process as a result of the demands of the problem. In order to learn computer programs, a program induction system should search the solutions in the space of all possible programs. However, the space is extremely large and the traditional weak search methods clearly cannot solve the problem. Thus, some adaptive and intelligent search methods are required.

An intelligent search method starts with one or more search points (structures) in the search space, evaluates the performances of the current structures for solving the problem at hand, and then employs the information about the performances to determine how to proceed the search in the space. As described in chapter 2, evolutionary algorithms have these properties and thus they are intelligent and effective. Nevertheless, they are weak methods without domain-specific knowledge hard-wired in the methods.

In chapter 3, we have introduced some ILP systems which can be classified into top-down and bottom-up systems. Most existing ILP systems applies strong search methods obtained by combining some greedy search strategies and heuristics. Various systems differ mainly in the strong methods used to guide the search for the desired programs. The problem is that these strong methods are not always applicable because they may trap the systems in local maxima. Moreover, other learning paradigms such as reinforcement learning (Sutton 1988; 1992, Tesauro 1992, Lin 1992, Kaelbling 1993) and strategy learning cannot be achieved by ILP systems.

Since evolutionary algorithms are effective weak methods, we have proposed the idea of combining the effective search power of evolutionary algorithms and the knowledge representation power of first-order logic. In chapter 4, we have described a novel system called the Genetic Logic Programming System (GLPS) that realizes the idea. GLPS can learn function free first-order logic programs with constants. It takes the advantages of existing ILP and GP systems while avoids the disadvantages of them. We have devised a new method so that a logic program can be represented as a forest of AND-OR trees. This representation method facilitates the generation of the initial population of logic programs and the operations of various genetic operators such as crossover and reproduction. A number of applications of GLPS have been successfully implemented. They are the Winston's arch problem, the modified Quinlan's network reachability problem, and the factorial problem. These applications have demonstrated that GLPS is a promising alternative to other ILP systems. Since GLPS uses the same representation of other ILP systems, it is possible to combine GLPS with them.

Although GLPS can induce logic programs, it cannot accept domain-specific knowledge in order to perform knowledge-intensive and evolutionary search. Moreover, existing program induction systems are limited in the programming languages in which the induced programs are expressed. For example, GP systems can only induce programs represented as S-expressions in Lisp. ILP systems can only produce logic programs. Since the formalisms of the above two kinds of systems are so different, they cannot be integrated easily although their properties and goals are similar. If they can be combined in a common framework, then their techniques and theories can be shared and their problem solving power can be enhanced.

In chapter 5, we have proposed a novel, flexible, and general framework that combines GP and ILP. Moreover, the search space can be specified declaratively under this framework. This framework is based on a formalism of logic grammars and a system called LOGENPRO (The LOgic grammar based GENetic PROgramming system) has been developed. The formalism is powerful enough to represent context-sensitive information and domain-dependent knowledge. The knowledge can be used to accelerate the learning of programs. The formalism is also very flexible and programs in various programming languages such as Lisp, Prolog, Fuzzy Prolog, and C can be induced. We have demonstrated that programs in different programming languages can be expressed as derivation trees. This representation method facilitates the generation of the initial population of programs and the operations of various genetic operators such as crossover and mutation. The novel and effective methods of performing the crossover and mutation operations have been described. They guarantee that only valid offspring will be generated.

We have demonstrated that LOGENPRO can be used easily in Chapter 6. Furthermore, it has been shown that programs in Lisp, Prolog, and C can be induced. Firstly, a logic grammar template is provided to facilitate the application of LOGENPRO to emulate GP and two experiments have been performed. In the first experiment, it has been shown that knowledge of data type can be represented easily in LOGENPRO. We have illustrated that LOGENPRO alleviates the problem in traditional GP that all the variables, constants, arguments for functions, and values returned from functions must be of the same data type. The experiment has proven that LOGENPRO can find a solution much faster than GP and the computation required by LOGENPRO is much smaller than that of GP. Thus, LOGENPRO can emulate the effects of Strongly Type Genetic Programming (STGP) effortlessly (Montana 1993).

In the second experiment, we have illustrated how to apply LOGENPRO to emulate Automatically Defined Functions (ADF) proposed by Koza (1992; 1994). Automatic discovery of problem representation primitives is one of the most important research areas in Genetic Programming. ADF is one of the approaches that have been proposed to acquire problem representation primitives automatically. We have found that LOGENPRO can learn a program much faster than ADF and the computation required by LOGENPRO is much smaller than that of ADF. This experiment has shown that LOGENPRO can emulate the effects of STGP and ADF simultaneously and effortlessly. It has also been proven that our framework can transform evolutionary weak methods to strong methods by incorporating various knowledge about the problem being solved.

Secondly, we have shown that LOGENPRO can easily emulate GLPS in learning logic programs. A logic grammar template has been provided to facilitate the application of LOGENPRO. We have performed three experiments to show that LOGENPRO can emulate our GLPS. The experiment described in sub-section 6.2.2 has also demonstrated the advantage of LOGENPRO. Since different formulations of a learning problem can be experimented easily with different logic grammars to find the most appropriate one, LOGENPRO can handle some learning problems, such as the Winston's arch problem, that cannot be solved completely by GLPS.

Thirdly, we have employed LOGENPRO to perform symbolic regression to illustrate that LOGENPRO can induce programs in the C programming language. We have demonstrated the possibility of learning programs in some imperative languages.

Knowledge discovery in databases is an important and promising research field in computer science and artificial intelligence (Frawley et al. 1991, Piatetsky-Shapiro and Frawley 1991). We have presented three applications of LOGENPRO in acquiring knowledge from databases in chapter 7. These applications have demonstrated the advantages of LOGENPRO over other learning systems. In the first application, we have employed LOGENPRO to induce knowledge represented in decision trees from a real-world database and compared the results obtained by Michie et al. (1994) for the same problem. We have found that Cal5, ITrule, Discrim, Logdisc and DIPOL92 perform better than LOGENPRO marginally. Since the detailed information about the accuracy of the former systems is not available, it cannot be concluded that whether the differences in accuracy are significant. On the other hand, LOGENPRO performs better than CART, RBF, CASTLE, NaiveBay, IndCART, Back-propagation, C4.5, SMART, Baytree, k-NN, NewID, AC2, LVQ, ALLOC80, CN2, and Quadisc for the problem. Interestingly, LOGENPRO is better than C4.5 and CN2, two systems that have been reported in the literature (Quinlan 1992, Clark and Niblett 1989) about their outstanding performances in inducing decision trees or rules.

In the second application, we have employed LOGENPRO to combine evolutionary search methods and a variation of FOIL, BEAM-FOIL, in learning logic programs. Since noise handling mechanisms are very important research topics in knowledge discovery in databases, we have evaluated the performance of LOGENPRO in inducing knowledge from noisy datasets using the chess endgame problem. Detailed comparisons between LOGENPRO and other ILP systems have been conducted. It has been found that LOGENPRO outperforms these ILP systems significantly at most noise levels. These results are surprising because the LOGENPRO uses the same noise-handling mechanism of FOIL by initializing the population with programs created by BEAM-FOIL. One possible explanation of the better performance of LOGENPRO is that the Darwinian principle of survival and reproduction of the fittest is a good noise handling method. It avoids overfitting noisy examples, but at the same time, it can find interesting and useful patterns from these noisy examples.

Imprecise and uncertain examples are frequent in real world environment, because many everyday examples are denoted in linguistic terms which are essentially imprecise and uncertain. However, there are very few studies on the issue of inducing knowledge from imprecise and uncertain datasets. In the third application, we have successfully used LOGENPRO to acquire knowledge from imprecise and uncertain training examples stored in a fuzzy relational database. The induced knowledge is represented as a program in Fuzzy Prolog (Li and Liu 1990). To the knowledge of the authors, LOGENPRO is currently the only system that can learn programs in Fuzzy Prolog.

Existing ILP systems cannot improve themselves automatically. In chapter 8, we have proposed an adaptive ILP system that can improve itself during the learning process. The adaptive ILP system is composed of an external interface, a biases base, a knowledge base of background knowledge, an example database, an empirical ILP learner, a meta-level learner, and a learning controller. An implementation of the adaptive ILP system has been completed. In this implementation, the empirical ILP learner performs top-down search in the hypothesis space defined by the concept description language, the language bias, and the background knowledge. The search is directed by search biases which can be induced and refined by LOGENPRO.

It has been demonstrated that the adaptive ILP system performs better than FOIL in inducing logic programs from perfect or noisy training examples. The experimentation has illustrated the benefit of an adaptive ILP system over existing ILP systems because the former can improve itself automatically. The result implies that the search bias induced by LOGENPRO is better than that of FOIL, which is designed by a top researcher in the field. Consequently, LOGENPRO  is a promising technique for implementing a meta-level learning system. The result is very encouraging as it suggests that the process of natural selection and evolution can successfully evolve a high performance ILP system. This adaptive learning approach, though computationally intensive, is rather exciting, as it opens up many opportunities for creating or improving learning algorithms.

The field of program induction investigates the problem of inducing computer programs in different programming languages from specifications. Different ways have been proposed to present specifications. They are natural language, special-purpose languages, very high-level languages, formal specification languages, and examples. They have their own pros and cons, and it is beneficial to combine their advantages while preventing their disadvantages.

Our LOGENPRO can be viewed as a system that accepts specifications in different ways: A logic grammar is a partial specification in a formal specification language that describes which programs are valid; A fitness function represents another partial specification using examples (i.e. fitness cases) and/or very high-level languages, and it evaluates different programs allowed by the logic grammar. Moreover, LOGENPRO employs deduction to generate the initial population of program from the logic grammar given and uses induction to produce offspring from parental programs. The inductive methods have been implemented in the form of genetic operators such as crossover and mutation. Thus, LOGENPRO employs both deduction and induction to find appropriate programs from the extremely large search space. The effectiveness and efficiency of LOGENPRO can be attributed to this insightful combination of various ways of specifications and different inference mechanisms.

## 9.2. Future work

The future work can be classified into four categories: applying LOGENPRO to discover knowledge from databases; learning recursive programs; applying LOGENPRO in engineering design; and exploiting parallelism of evolutionary algorithms. These categories are detailed in the following sub-sections.

### 9.2.1. Applying LOGENPRO to discover knowledge from databases

In section 7.2, we have shown that LOGENPRO can successfully induce knowledge represented as logic programs from noisy datasets. We have also found that the noise handling ability of LOGENPRO is better than many existing ILP systems. Since training examples stored in everyday databases are usually imperfect, a very important research area in knowledge discovery in databases investigates how to improve the noise handling mechanisms of learning algorithms.

One can use LOGENPRO on extracting knowledge from other datasets of the field. One can also combine LOGENPRO with other learning systems such as GOLEM (Muggletion and Feng 1990), LINUS (Lavrac and Dzeroski 1994), and mFOIL (Lavrac and Dzeroski 1994) to explore the possibility of further improvement on its learning ability.

We have demonstrated in section 7.3 that LOGENPRO can acquire imprecise and uncertain knowledge represented as programs in Fuzzy Prolog from fuzzy relational databases. Although the result is very promising, it seems that the example shown in that section is rather simple. It is believed that LOGENPRO can be applied to acquire knowledge represented in Fuzzy Prolog from real-world databases. We have applied the Automatic Knowledge Acquisition and Refinement System (AKARS) to induce a complicated real-life knowledge base incorporated with fuzzy concepts from medical training examples (Leung and Wong 1991a; 1991b). The induced knowledge is used in a medical expert system which deals with the problem of rupture of membranes. The limitation of AKARS is that only propositional production rules extended with fuzzy concepts can be acquired. Thus, one could try to apply LOGENPRO to learn knowledge represented in a more expressive language (i.e. Fuzzy Prolog) using the medical training examples.

### 9.2.2. Learning recursive programs

One of the most important and challenging areas of research in evolutionary algorithms is to investigate ways to successfully apply evolutionary algorithms to larger and more complicated problems. As discussed in sub-section 6.1.2, one approach to make a given problem more tractable is to discover problem representations automatically. Koza (1994) uses the even-n-parity problem to demonstrate extensively that his approach of Automatic Function Definition (ADF) can facilitate the solution of the problem.

The boolean even-n-parity function of n boolean arguments return T (True) if an even number of its arguments are T, otherwise it returns NIL (False). Since there are n boolean arguments $D_1$, $D_2$, ..., $D_n$ involved in the problem, they form the terminal set. The function set {AND, OR, NAND, NOR} contains four two-argument primitive boolean functions.

Koza shows that an even-7-parity problem can be solved using ADF. He finds that about 1440000 individuals, I(M, i, z), should be evaluated to obtain at least one solution with 99% probability (see sub-section 6.1.1 to find out how to obtain this number). Unfortunately, the solutions found by ADF can only solved the even-n-parity for a particular value of n. If a different value of n is used, ADF must be used again to find other programs that can solve the new even-n-parity problem.

Clearly, the solution found is not general enough to solve all even-n-parity problem for n greater than or equal to zero. A better solution should be recursive such as the following one:

```
(defun parity (L)
  (if (null L)
      T
     (NAND
       (OR (first  L)
           (NAND (parity (rest  L)) T))
       (NAND (first  L)
             (NAND
               (AND  T
                (parity (rest  L)))
               T)))))
```

In this recursive program, the argument `L` is a list of boolean values. Any number of boolean values can exist in the list `L`. In fact, this program can solve all even-n-parity problem for n greater than or equal to zero. To evolve this function, the terminal set must contain the argument `L`, the truth value `T` and the truth value `NIL`. The function set **F** is:

**F** = {if, null, first, rest, parity, AND, OR, NAND, NOR}.

Moreover, the above program can be simplified to:

```
(defun parity (L)
  (if (null L)
      T
  (xor (first L) (parity (rest L)))))

(defun xor (a b)
  (AND (OR a b) (NAND a b)))
```

Since the simplified program invokes a sub-function `xor` which is not available in the function set, it must be learned simultaneously with the main program. It seems that this problem can be solved because we have already shown in sub-section 6.1.2 that LOGENPRO can emulate ADF. Thus, one should investigate how to apply

LOGENPRO to learn recursive programs with different difficulties and properties. There are many inductive learning systems such as THESYS (Summers 1977) and ADATE (Olsson 1995) that can induce recursive functional programs efficiently. Therefore, one could try to implement these techniques on LOGENPRO.

### 9.2.3. Applying LOGENPRO in engineering design

The field of engineering design methodologies is one of the most active fields of research in mechanical engineering (Roston 1994). Engineering design is the systemic, intelligent generation and evaluation of specifications for artifacts whose form and function achieve stated objectives and satisfy specified constraints (Dym 1992). It is observed that this definition fits well with that of automatic programming.

Pahl and Beitz (1984) introduced a systematic approach to engineering design. This approach decomposes the design process into four phases: clarification of the task, conceptual design, embodiment design, and detailed design. It is believed that LOGENPRO can be applied in the last three phases to assist the designer. One of the fundamental problems is how to represent different designs generated in various phases. It is important that these representations can be translated into other representations, including a final physical instantiation of the artifact being designed.

There are numerous representation methodologies. The function logic method of value analysis is one of the most general representation methods (Sturges et al. 1992). In this method, objects and classes of objects are represented by a hierarchy of noun-verb pairs. The disadvantages of this method are that it lacks some of the formality of other methods and it is difficult to be implemented in a computer. Cagan and Agogino (1987) proposed to represent the concept of designing from the basic and underlying principles. However, it is difficult to generalize this method for more complicated designs.

Another means of representation is formal grammars. Stiny (1980) develops the concept of shape grammars which are used to describe planar shapes. It has been shown that graph grammars are equivalent to other types of formal grammars (Gips and Stiny 1980). Mullins and Rinderle (1991) presented the reasons for employing formal grammars for engineering design.

Tanaka presented a method to understand the functions of electronic circuits (1993). A circuit is viewed as a sentence and its elements as words. Circuit structures are defined by rules written in a logic grammar called Definite Clause Set Grammar (DCSG). The advantage of this approach is that circuit designs can be analyzed automatically.

Reddy and Cogan (1994) used shape grammars and simulated annealing to solve a variety of design program. They showed that a grammatical representation of an artifact and a means of intelligent search can be used to generate optimal designs. Roston (1994) extended their work using an evolutionary algorithm, Strongly Typed Genetic Programming (Montana 1993), to search for optimal designs represented in a context-free grammar.

We believe that LOGENPRO is a better approach for engineering design because context-sensitive information and domain-specific knowledge can be represented to accelerate the intelligent search process. This property has been established in section 6.1. Moreover, LOGENPRO is a flexible enough to induce programs in various special-purpose languages that represent the designs of artifacts.

### 9.2.4. Exploiting parallelism of evolutionary algorithms

For almost all practical applications of LOGENPRO, most computation time is consumed in evaluating the fitness of each program in the population since the genetic operators of LOGENPRO can be performed efficiently. The fitness evaluation process is time-consuming for the following reasons:

- It is required to interpret or compile each program in the population.
- It is necessary to compute fitness over several different fitness cases in order to obtain an accurate estimate of the fitness of a program. For example, consider the problem of learning search biases for an adaptive ILP system. Many different problems of learning logic programs should be used to estimate the fitness of a search bias. In other words, these problems of learning logic programs, such as the *member* problem and the *uncle* problem, are the fitness cases for the problem of learning search bias.
- It is required to perform complicate and time-consuming computation to get a fitness value for a single fitness case. Consider the above example again, the search bias to be evaluated and a set of training examples are provided first. Then, LOGENPRO invokes a top-down or bottom-up algorithm to solve the problem of inducing a logic program from the set of training examples. Usually, several minutes are required to find a satisfactory logic program. After a logic program is induced, the logic program must be evaluated using another set of testing examples. In normal situation, this process takes about a few minutes because the testing set usually contains hundreds or even thousands of examples. The accuracy of the learned logic program on the testing set forms the fitness value of the search bias for a single fitness case. Thus, it is clear that several minutes or even hours are needed to find the fitness value.

Memory availability is another important problem of LOGENPRO because the population usually has a large number of programs. Moreover, since programs are represented as derivation trees of varying sizes, shapes and structures. This representation method requires more memory to store programs than that used in GP.

There is a relation between the difficulty of the problem to be solved and the size of the population. In order to solve substantial and real-world problems, a population size of thousands and a longer evolution process are usually required. A larger population and a longer evolution process imply a more number of fitness evaluation must be conducted and more memory are required. In other words, a lot of computational resources are required to solve substantial and practical problems. Usually, this requirement cannot be fulfilled by normal workstations.

Fortunately, these time-consuming fitness evaluations can be performed independently for each program in the population and programs in the population can be distributed among multiple computers. Thus, we plan to develop a parallel version of LOGENPRO.

Evolutionary algorithms have a high degree of inherent parallelism which is one of the motivation of studies in this field. In natural populations, thousands or even millions of individuals exist in parallel and these individuals operates independently with a little cooperation and/or competition among them. This suggests a degree of parallelism that is directly proportional to the population size used in evolutionary algorithms. There are different ways of exploiting parallelisms in evolutionary algorithms. We plan to study the possibility of parallelizing LOGENPRO using four different approaches. They are master-slave models, improved-slave models, massively parallel evolutionary algorithms, and island models.

The most direct way to implement a parallel evolutionary algorithm is to implement a global population in the master processor. The master sends each individual to a slave processor and let the slave to find the fitness value of the individual. After the fitness values of all individuals are obtained, the master processor selects some individuals from the population using some selection method, performs some genetic operations, and then creates a new population of offspring. The master sends each individual in the new population to a slave again and the above process is iterated until the termination criterion is satisfied.

Master-slave models can be improved easily using the tournament selection. Another direct way to implement a parallel evolutionary algorithm is to implement a global population and use the tournament selection. As described in sub-section 2.2.1.1, the tournament selection approximates the behavior of ranking. Assume that the population size N is even and there are more than N/2 processors. N/2 slave processors are selected and are numbered from 1 to N/2. A processor selected from the remaining processors maintains the global population and implements an algorithm that controls the overall evolution process and the other N/2 slave processors. Each slave processor performs two independent m-ary tournaments. In each tournament, m individuals are sampled randomly form the global population. These m individuals are evaluated in the slave processor and the winner is kept. Since there are two tournaments, the two winners produced can be crossed in the slave processor to generate two offspring. The slave processor may perform further modifications to the offspring. The offspring are then sent back to the global population and the master processor proceeds to the next generation if all offspring are received from the N/2 slave processors.

Massively parallel evolutionary algorithms explore the computing power of massively parallel computers such as the Maspar. To explore the power of this kind of computers, one can assign one individual to each processor, and allow each individual to seek a mate close to it. A global random mating scheme is inappropriate because of the limitation of the communication abilities of these computers. Each processor can select probabilistically an individual in its neighborhood to mate with. The selection is based on the fitness proportionate selection, the ranking, the tournament selection, or other selection methods proposed in the literature. Only one offspring is produced and becomes the new resident at that processor. The common property of different massively parallel evolutionary algorithms is that selections and mating are typically restricted to a local neighborhood.

Island models can fully explore the computing power of course grain parallel computers such as the Sparc 2000 and distributed workstations. Assume that we have 20 high performance processors, such as the ultrasparc processors, and have a population of 4000 individuals. We can divide the total population down into 20 subpopulations (islands or demes) of 200 individuals each. Each processor can then execute a normal evolutionary algorithm such as LOGENPRO on one of these subpopulations. Occasionally, the subpopulations would swap a few individuals. The migration allows subpopulations to share genetic material (Whitley and Starkweather 1990, Gorges-Schlenter 1991, Tanese 1989, Starkweather et al. 1991).

Since there are 20 independent evolutionary searches occur concurrently, these searches will be different to a certain extent because the initial subpopulations will impose a certain sampling bias. Moreover, genetic drift will tend to drive these subpopulations in different directions. By employing migration, island models are able to exploit differences in the various subpopulations. These differences maintain genetic diversity of the whole population and thus can prevent the problem of premature convergence. We plan to exploit a number of variations of island models. These variations investigate the effects of subpopulations with different sizes or even dynamic sizes, asynchronous migration, dynamic number of migrating individuals, subpopulations with different fitness functions, adaptive migration methods, and cooperative/competitive co-evolution.

# Reference

Abramson, H. and Dahl, V. (1989). *Logic Grammars*. Berlin: Springer-Verlag.

Aho, A. V. and Ullman, J. D. (1977). *Principles of Compiler Design*. Reading MA: Addison-Wesley.

Angeline, P. (1993). *Evolutionary Algorithms and Emergent Intelligence*. Ph.D. Dissertation. The Ohio State University.

Angeline, P. (1994). Genetic Programming and Emergent Intelligent. In K. E. Kinnear, Jr. (ed.), *Advances in Genetic Programming*, pp. 75-97. Cambridge MA: MIT Press.

Angeline, P. and Pollack, J. (1992). The evolutionary induction of subroutines. In *Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society*, pp. 236-241. Hillsdale NJ: Lawrence Erlbaum.

Angeline, P. and Pollack, J. (1993). Competitive Environments evolve better Solutions for Complex Tasks. In S. Forrest (eds.), *Proceeding of the Fifth International Conference on Genetic Algorithms*, pp. 264-270. San Mateo CA: Morgan Kaufmann.

Bäck, T., Hoffmeister, F., and Schwefel, H. P. (1991). A Survey of Evolution Strategies. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pp. 2-9. San Mateo CA: Morgan Kaufmann.

Baker, J. (1985). Adaptive Selection Methods for Genetic Algorithms. In J. Grefenstette (ed.), *Proceedings of an International Conference on Genetic Algorithms and Their Applications*, pp. 101-111. Hillsdale NJ: Lawrence Erlbaum.

Baker, J. (1987). Reducing Bias and Inefficiency in the Selection Algorithm. In *Proceedings of the Second International Conference on Genetic Algorithms and their Applications*. Hillsdale NJ: Lawrence Erlbaum.

Bergadano, F., Giordana, A., and Saitta, L. (1991). *Machine Learning: An Integrated Framework and its Applications*. London: Ellis Horwood.

Biermann, A. W. (1972). On the Inference of Turing Machines from Sample Computations. *Artificial Intelligence*, **3**, pp. 181-198.

Biermann, A. W. and Smith, D. R. (1979). A Production Rule Mechanism for Generating LISP Code. *IEEE Trans. on Systems, Man, and Cybernetics*, **9**, pp. 260-276.

Bratko, I. and King, R. (1994). Applications of Inductive Logic Programming. *SIGART Bulletin*, **5** (1), pp. 43-49.

Bridges, C. and Goldberg, D. (1987). An Analysis of Reproduction and Crossover in a Binary-coded Genetic Algorithm. In J. J. Grefenstette (ed.), *Proceedings of the Second International Conference on Genetic Algorithms and Their Applications*. Hillsdale NJ: Lawrence Erlbaum.

Breiman, L., Friedman, J. H., Olshen, R. A. and Stone, C. J. (1984). *Classification and Regression Trees*. Belmont: Wadsworth.

Buchanan, B. G. and Shortliffe, E. H. Editors. (1984). *Rule-based Expert Systems The MYCIN Experiments of the Stanford Heuristic Programming Project. Reading.* MA: Addison-Wesley.

Cagan, J. and Agogino, A. M. (1987). Innovative Design of Mechanical Structures from First Principles. *AI EDAM*, **1**, pp. 169-189.

Cameron-Jones, R. and Quinlan, J. (1993). Avoiding Pitfalls when Learning Recursive Theories. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*. San Mateo, CA: Morgan Kaufmann.

Cameron-Jones, R. and Quinlan, J. (1994). Efficient Top-down Induction of Logic Programs. *SIGART Bulletin*, **5** (1), pp. 33-42.

Carbonell, J. G., editor (1990). *Machine Learning: Paradigms for Machine Learning.* Cambridge MA: MIT Press.

Cestnik, B. (1990). Estimating Probabilities: A Crucial Task in Machine Learning. In *Proceedings of the Ninth European Conference on Artificial Intelligence*, pp. 147-149. London: Pitman.

Cestnik, B. and Bratko, I. (1991). On Estimating Probabilities in Tree Pruning. In Y. Kodratoff (ed.), *Proceedings of the Fifth European Working Session on Learning*, pp. 151-163. Berlin: Springer Verlag.

Cestnik, B., Kononenko, J. and Bratko, I. (1987). ASSISTANT 86: A knowledge elicitation tool for sophisticated users. In I. Bratko and N. Lavrac (Ed.), *Progress in Machine Learning*, pp. 31-45. Wilmslow: Sigma Press.

Clark, K. (1978). Negation as Failure. In H. Gallaire and J. Minker (eds.), *Logic and Databases*, pp. 293-322. NY: Plenum Press.

Clark, P. and Boswell, R. (1991). Rule Induction with CN2: Some Recent Improvements. In Y. Kodratoff (ed.), *Proceedings of the Fifth European Working Session on Learning*, pp. 151-163. Berlin: Springer-Verlag.

Clark, P. and Niblett, T. (1989). The CN2 induction algorithm. *Machine Learning*, **3**, pp. 261-283.

Cohen, W. W. (1993). Pac-learning a Restricted Class of Recursive Logic Programs. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pp. 86-92. Cambridge, MA: MIT Press.

Colmerauer, A. (1978). Metamorphosis Grammars. In L. Bolc (Ed.), *Natural Language Communication with Computers*. Berlin: Springer-Verlag.

Cormen, T. H., Leiserson, C. E., and Rivest, R. L. (1990). *Introduction to Algorithms*. Cambridge, MA: MIT Press.

Davidor, Y. (1991). A generic Algorithm Applied to Robot Trajectory Generation. In L. Davis (ed.), *Handbook of Genetic Algorithms*, pp. 144-165. Van Nostrand Reinhold.

Davis, L. D. (1991). *Handbook of Genetic Algorithms*. Van Nostrand Reinhold.

DeJong, G. F., editor (1993). *Investigating Explanation-Based Learning*. Boston: Kluwer Academic Publishers.

DeJong, G. F. and Mooney, R. (1986). Explanation-Based Learning: An Alternative View. *Machine Learning*, **1**, pp. 145-176.

DeJong, K. A. and Spears, W. M. (1990). An Analysis of the Interacting Roles of Population Size and Crossover in Genetic Algorithms. In *Proceedings of the First Workshop on Parallel Problem Solving from Nature*, pp. 38-47. Berlin: Springer-Verlag.

De Jong, K. A., Spears, W. M. and Gordon, D. F. (1993). Using Genetic Algorithms for Concept Learning. *Machine Learning*, **13**, pp. 161-188.

De Raedt, L. (1992). *Interactive Theory Revision: An Inductive Logic Programming Approach.* London: Academic Press.

De Raedt, L. and Bruynooghe, M. (1989). Towards friendly Concept-learners. In *Proceeding of the Eleventh International Joint Conference on Artificial Intelligence*, pp. 849-854. San Mateo, CA: Morgan Kaufmann.

De Raedt, L. and Bruynooghe, M. (1992). Interactive Concept Learning and Constructive Induction by Analogy. *Machine Learning*, **8**, pp. 251-269.

Dietterich, T. G. (1986). Learning at the Knowledge Level. *Machine Learning*, **1**, pp. 287-316.

Dym, C. L. (1992). Representation and Problem-Solving: the Foundations of Engineering Design. *Environment and Planning B*, **19**, pp. 97-105.

Dzeroski, S. and Lavrac, N. (1993). Inductive Learning in Deductive Databases. *IEEE Transactions on Knowledge and Data Engineering*, **5**, pp. 939-949.

Ellman, T. (1989). Explanation-Based Learning: A Survey of Programs and Perspectives. *ACM Computing Surveys*, **21**, 163-222.

Eshelman, L. J., Caruna, R., and Schaffer, J. D. (1989). Biases in the Crossover Landscape. In J. D. Schaffer (ed.), *Proceedings of the Third International Conference on Genetic Algorithms*, pp. 10-19. San Mateo CA: Morgan Kaufmann.

Fogel, D. B. (1992). A Brief History of Simulated Evolution. *In Proceedings of the First Annual Conference on Evolutionary Programming*. La Jolla CA.

Fogel, D. B. (1994). An Introduction to Simulated Evolutionary Optimization. *IEEE Trans. on Neural Network.*, **5**, pp. 3-14

Fogel, L., Owens, A., and Walsh, M. (1966). *Artificial Intelligence through Simulated Evolution*. New York: John Wiley and Sons.

Forrest, S. (1990). *A Study of Parallelism in the Classifier System and its Application to Classification in KL-ONE Semantic Networks*. London: Pitmann.

Frawley, W., Piatetsky-Shapiro, G., and Matheus, C. (1991). Knowledge Discovery in Databases: an Overview. In G. Piatetsky-Shapiro and W. Frawley (eds.), *Knowledge Discovery in Databases*, pp. 1-27. Menlo Park, CA: AAAI Press.

Furnkranz, J. (1994). FOSSIL: A Robust Relational Learner. In F. Bergadano and L. De Raedt (Eds.), *Proceedings of the European Conference on Machine Learning 1994*, pp. 122-137. Berlin: Springer-Verlag.

Gips, J. and Stiny, G. (1980). Production Systems and Grammars. *Environment and Planning B*, **7**, pp. 399-408.

Goldberg, A. T. (1986). Knowledge-based Programming: A Survey of Program Design and Construction Techniques. *IEEE Trans. on Software Engineering*, **12**, pp. 752-768.

Goldberg, D. (1987). Simple Genetic Algorithms and the Minimal, Deceptive Problem. In L. Davis (ed.), *Genetic Algorithms and Simulated Annealing*. London: Pitman.

Goldberg, D. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading MA: Addison-Wesley.

Goldberg, D. and Bridges, C. L. (1990). An Analysis of a Reordering Operator on a GA-hard Problem. *Biological Cybernetics*, **62**, pp. 397-405.

Goldberg, D. and Deb, K. (1991). A Comparative Analysis of Selective Schemes Used in Genetic Algorithms. In G. Rawlins (ed.), *Foundations of Genetic Algorithms*, pp. 69-93. San Mateo CA: Morgan Kaufmann.

Gorges-Schleuter, M. (1991). Explicit Parallelism of Genetic Algorithms through Population Structures. *Parallel Problem Solving from Nature*, pp. 150-159. Berlin: Springer-Verlag.

Greene, D. P. and Smith, S. F. (1993). Competition-Based Induction of Decision Models from Examples. *Machine Learning*, **13**, pp. 229-257.

Grefenstette, J. J. (1986). Optimization of Control Parameters for Genetic Algorithms. *IEEE Trans. Systems, Man, and Cybernetics*, **16**, pp. 122-128.

Holland, J. (1987). Genetic Algorithms and Classifier systems: Foundations and Future Directions. In J. J. Grefenstette (ed.), *Proceedings of the Second International Conference on Genetic Algorithms and Their Applications*, pp. 82-89. Hillsdale NJ: Lawrence Erlbaum.

Holland, J. (1992). *Adaptation in Natural and Artificial Systems*. Cambridge MA: MIT Press.

Hopcroft, J. E. and Ullman, J. D. (1979). *Introduction to automata theory, languages, and computation*. MA: Addison-Wesley.

Janikow, C. Z. (1993). A Knowledge-Intensive Genetic Algorithm for Supervised Learning. *Machine Learning*, **13**, pp. 189-228.

Kaelbling, L. P. (1993). *Learning in embedded systems*. Cambridge, MA: MIT Press.

Kalbfleish, J. (1979). *Probability and Statistical Inference, volume II*. NY: Springer-Verlag.

Kijsirikul, B., Numao, M., and Shimura, M. (1992a). Efficient Learning of Logic Programs with Non-Determinate, Non-Discriminating Literals. In S. Muggleton (ed.), *Inductive Logic Programming*, pp. 361-372. London: Academic Press.

Kijsirikul, B., Numao, M., and Shimura, M. (1992b). Discrimination-Based Constructive Induction of Logic Programs. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pp. 44-49. San Jose, CA.

Kinnear, K. E. Jr. (1994a). Alternatives in Automatic Function Definition: A Comparison of Performance. In K. E. Kinnear, Jr. (ed.), *Advances in Genetic Programming*, pp. 119- 141. Cambridge MA: MIT Press.

Kinnear, K. E. Jr., editor (1994b). *Advances in Genetic Programming*. Cambridge MA: MIT Press.

Kodratoff, Y. and Michalski, R., editors (1990). *Machine Learning: An Artificial Intelligence Approach, Volume III*. San Mateo CA: Morgan Kaufmann.

Kowalski, R. A. (1979). *Logic For Problem Solving*. Amsterdam: North-Holland.

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

Koza, J. R. (1994). *Genetic Programming II: Automatic Discovery of Reusable Programs*. Cambridge MA: MIT Press.

Lavrac, N. and Dzeroski, S. (1994). *Inductive Logic Programming: Techniques and Applications*. London: Ellis Horword.

Leung, K. S. and Wong, M. H. (1990). An Expert-system Shell using Structured Knowledge. *IEEE Computer*, **23** (3), pp. 38-47.

Leung, K. S. and Wong, M. L. (1991a). Inducing and refining rule-based knowledge from inexact examples. *Knowledge Acquisition*, **3**, pp. 291-315.

Leung, K. S. and Wong, M. L. (1991b). Automatic refinement of Knowledge Bases with Fuzzy Rules. *Knowledge-Based Systems*, **4**, pp. 231-246.

Leung, K. S. and Wong, M. L. (1991c). AKARS-1: an Automatic Knowledge Acquisition and Refinement System. In H. Motada, R. Mizoguchi, J. Boose and B. Gaines (Eds.), *Knowledge Acquisition for Knowledge-Based Systems*. Amsterdam: IOS Press.

Levenick, J. (1991). Inserting Introns Improves Genetic Algorithm Success Rate: Taking a Cue from Biology. In R. K. Belew and L. B. Booker (eds.), *Proceeding of the Fourth International Conference on Genetic Algorithms*, pp. 123-127. San Mateo CA: Morgan Kaufmann.

Lewis, H. R. and Rapadimitrion, C. H. (1981). *Elements of the theory of computation*. NJ: Prentice Hall.

Li, D. and Liu, D (1990). *A Fuzzy Prolog Database system*. Great Britain: Research Studies Press Ltd.

Lin, L. J. (1992). Self-Improving Reactive Agents Based on Reinforcement Learning, Planning and Teaching. *Machine Learning*, **8**, pp. 293-321.

Lloyd, J. (1987). *Foundation of Logic Programming*. 2nd edition. Berlin: Springer Verlag.

Louis, S. J. and Rawlins, G. J. E. (1991). Designer Genetic Algorithms: Genetic Algorithms in Structure Design. In R. K. Belew and L. B. Booker (eds.), *Proceeding of the Fourth International Conference on Genetic Algorithms*, pp. 53-60. San Mateo CA: Morgan Kaufmann.

Matthews, B. W. (1975). Comparison of the Predicted and Observed Secondary Structure of T4 Phase Lysozyme. *Biochemica et Biophysical Acta*, **405**, pp. 442-451.

Michalewicz, Z. (1994). *Genetic Algorithms + Data Structures = Evolutionary Programs*. 2nd Edition. New York: Springer-Verlag.

Michalski, R. J. (1983). A Theory and Methodology of Inductive Learning. In R. Michalski, J. G. Carbonell and T. M. Mitchell (eds.), *Machine Learning: An Artificial Intelligence Approach, Volume I*, pp. 83-134. San Mateo CA: Morgan Kaufmann.

Michalski, R. J., Carbonell, J. G., and Mitchell, T. M., editors (1983). *Machine Learning: An Artificial Intelligence Approach, Volume I*. San Mateo CA: Morgan Kaufmann.

Michalski, R. S., Mozetic, I., Hong, J. and Lavrac, N. (1986a). The multi-purpose incremental learning system AQ15 and its testing application on tree medical domains. In *Proceedings of the National Conference on Artificial Intelligence*, pp. 1041-1045. San Mateo, MA: Morgan Kaufmann.

Michalski, R. J., Carbonell, J. G., and Mitchell, T. M., editors (1986b). *Machine Learning: An Artificial Intelligence Approach, Volume II*. San Mateo CA: Morgan Kaufmann.

Michalski, R. and Tecuci, G., editors (1994). *Machine Learning: A Multistrategy Approach, Volume IV*. San Francisco CA: Morgan Kaufmann.

Michie, D. Spiegelhalter, D. J., and Taylor, C. C. editors (1994). *Machine Learning, Neural and Statistical Classification*. London: Ellis Horwood.

Minton, S (1989). *Learning Search Control Knowledge: An Explanation-Based Approach*. Boston: Kluwer Academic.

Minsky, M. (1963). Steps towards Artificial Intelligence. In E. Feigenbaum and I. Feldman (eds.), *Computer and Thought*. Reading MA: Addison Wesley.

Mitchell, T. M. (1982). Generalization as Search. *Artificial Intelligence*, **18**, pp. 203-226.

Mitchell, T. M., Keller, R. M. and Kedar-Cabelli, S. T. (1986). Explanation-Based Generalization: A Unifying View. *Machine Learning*, **1**, pp. 47-80.

Montana, D. J. (1993). Strongly Typed Genetic Programming. Technical report 7866, Bolt, Beranek, and Newman .

Mooney, R. J. (1989). *A General Explanation-Based Learning Mechanism and its Application to Narrative Understanding*. London: Pitman.

Morik, K. Wrobel, S. Kietz, J., and Emde, W. (1993). *Knowledge Acquisition and Machine Learning: Theory, Methods, and Applications*. London: Academic Press.

Muggletion, S. (1992). Inductive Logic Programming. In S. Muggletion (ed.), *Inductive Logic Programming*, pp. 3-27. London: Academic Press.

Muggletion, S. (1994). Inductive Logic Programming. *SIGART Bulletin*, **5** (1), pp. 5-11.

Muggleton, S. and Buntine, W. (1988). Machine Invention of First-order Predicates by Inverting Resolution. In *Proceedings of the Fifth International Conference on Machine Learning*, pp. 339-352. San Mateo, CA: Morgan Kaufmann.

Muggletion, S., Bain, M., Hayes-Michie, J., and Michie, D. (1989). An Experimental Comparison of Human and Machine Learning Formalisms. In *Proceedings of the Sixth International Workshop on Machine Learning*, pp. 113-118. San Mateo, CA: Morgan Kaufmann.

Muggletion, S. and Feng, C. (1990). Efficient Induction of Logic Programs. In *Proceedings of the FIrst Conference on Algorithmic Learning Theory*, pp. 368-381. Tokyo: Ohmsha.

Muggleton, S. and De Raedt, L. (1994). Inductive Logic Programming: Theory and Methods. *J. Logic Programming*, **19-20**, pp. 629-679.

Muhlenbein, H. (1991). Evolution in Time and Space - The Parallel Genetic Algorithm. In G. Rawlins (ed.), *Foundations of Genetic Algorithms*, pp. 316-337. San Mateo CA: Morgan Kaufmann.

Muhlenbein, H. (1992). How genetic Algorithms Really Work: I. Mutation and Hillclimbing. In R. Manner and B. Manderick (eds.), *Parallel Solving from Nature 2*. North Holland.

Mullins, S. and Rinderle, J. R. (1991). Grammatical Approaches to Engineering Design, Part 1: An Introduction and Commentary. *Research in Engineering Design*, **2**, pp. 121-135.

Newell, A. and Simon, H. A. (1972). *Human Problem Solving*. Englewood Cliffs NJ: Prentice Hall.

Nilson, N. J. (1980). *Principles of Artificial Intelligence*. Palo Alto CA: Tioga.

Nix, A. and Vose, M. (1992). Modeling Genetic Algorithms with Markov Chains. *Annals of Mathematics and Artificial Intelligence*, **5**, pp. 79-88.

Olsson, R. (1995). Inductive Functional Programming using Incremental Program Transformation. *Artificial Intelligence*, **74**, pp. 55-81.

Pahl, G. and Beitz, W. (1984). *Engineering Design*. Berlin: Springer-Verlag.

Paterson, M. S. and Wegman, M. N. (1978). Linear Unification. *Journal of Computer and System Sciences*, **16**, pp. 158-167.

Pazzani, M., Brunk, C. A. and Silverstein, G. (1991). A Knowledge-Intensive Approach to Learning Relational Concepts. In *Proceedings of the Eighth International Workshop on Machine Learning*, pp. 432-436, CA: Morgan Kaufmann.

Pazzani, M. and Kibler, D. (1992). The utility of knowledge in Inductive learning. *Machine Learning*, **9**, pp. 57-94.

Pearl, J. (1984). *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Reading MA: Addison Wesley.

Pereira, F. C. N. and Warren, D. H. D. (1980). Definite Clause Grammars for Language Analysis - A Survey of the Formalism and a Comparison with Augmented Transition Networks. *Artificial Intelligence*, **13**, pp. 231-278.

Pereira, F. C. N. and Shieber, S. M. (1987). *Prolog and Natural-Language Analysis.* CA: CSLI.

Piatetsky-Shapiro, G. and Frawley, W. J. (1991). *Knowledge Discovery in Databases.* Menlo Park, CA: AAAI Press.

Plotkin, G. D. (1970). A Note on Inductive Generalization. In B. Meltzer and D. Michie (eds.), *Machine Intelligence: Volume 5*, pp. 153-163. New York: Elsevier North-Holland.

Quinlan, J. R. (1986). Induction of Decision Trees. *Machine Learning*, **1**, pp. 81-106.

Quinlan, J. R. (1987). Simplifying Decision Trees. *Int. J. Man-Machine Studies*, **27**, pp. 221-234.

Quinlan, J. R. (1990). Learning Logical Definitions from Relations. *Machine Learning*, **5**, pp. 239-266.

Quinlan, J. R. (1991). Knowledge Acquisition from Structured Data - using Determinate Literals to Assist Search. *IEEE Expert*, **6**, pp. 32-37.

Quinlan, J. R. (1992). *C4.5: Programs for Machine Learning*. San Mateo, CA: Morgan Kaufmann.

Rechenberg, I. (1973). *Evolutionsstrategie: Optimienrung Technischer Systeme nach Prinzipien der Biologischen Evolution*. Stuttgart: Frommann-Holzboog Verlag.

Reddy, G. and Cagan, J. (1994). An Improved Grape Annealing Method for Truss Topology Generation. Accepted in *ASME Journal of Mechanical Design.*

Rich, C. and Waters, R. C. (1988). Automatic Programming: Myths and Prospects. *IEEE Computer*, **21**(8), pp. 40-51.

Rich, C. and Waters, R. C. (1990). *The Programmer's Apprentice*. New York: Addison-Wesley..

Roston, G. D. (1994). *A Genetic Methodology for Configuration Design*. Ph.D. Dissertation, Department of Mechanical Engineering, Carnegie Mellon University.

Rouveirol, C. (1991). Completeness for Inductive Procedures. In A. B. Lawrence and G. C. Collins (eds.), *Proceedings of the Eight International Workshop on Machine Learning*, pp. 452-456. San Mateo, CA: Morgan Kaufmann.

Rouveirol, C. (1992). Extensions of Inversion of Resolution Applied to Theory Completion. In S. Muggletion (ed.), *Inductive Logic Programming*, pp. 63-92. London: Academic Press.

Sammut, C. and Baneji, R. (1986). Learning Concepts by Asking Questions. In R. Michalski, J. G. Carbonell and T. M. Mitchell (eds.), *Machine Learning: An Artificial Intelligence Approach, Volume II*, pp. 167-191. San Mateo CA: Morgan Kaufmann.

Schaffer, J. D. (1987). Some Effects of Selection Procedures on Hyperplane Sampling by Genetic Algorithms. In L. Davis (ed.), *Genetic Algorithms and Simulated Annealing*. London: Pitman.

Schaffer, J. D. and Morishma, A. (1987). An Adaptive Crossover Distribution Mechanism for Genetic Algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms*, pp. 36-40. San Mateo CA: Morgan Kaufmann.

Schewefel, H. P. (1981). *Numerical Optimization of Computer Models*. John Wiley and sons.

Shapiro, E. (1983). *Algorithmic Program Debugging*. Cambridge, MA: MIT Press.

Shavlik, J. W. and Dietterich, T. G., editors (1990). *Readings in Machine Learning*. San Mateo CA: Morgan Kaufmann.

Starkweather, T., McDaniel, S., Mathias, K., Whitley, D., and Whitley, C. (1991). A Comparison of Genetic Sequencing Operators. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pp. 69-76. San Mateo CA: Morgan Kaufmann.

Sterling, L. and Shapiro, E. (1986). *The Art of Prolog*. MA: MIT Press.

Stiny, G. (1980). Introduction to Shape and Shape Grammars. *Environment and Planning B*, **7**, pp. 343-351.

Sturges, R. H. O'Shaughnessy, K. and Reed, R. G. (1992). A Systematic Approach to Conceptual Design Based on Function Logic. *International Journal of Concurrent Engineering*, **1**, pp. 93-106.

Summers, P. D. (1977). A Methodology for LISP Program Construction from Examples. *JACM*, **24**, pp. 161-175.

Sutton, R. S. (1988). Learning to predict by the method of temporal differences. *Machine Learning*, **3**, pp. 9-44.

Sutton, R. S. (1992). Introduction: The Challenge of Reinforcement Learning. *Machine Learning*, **8**, pp. 225-227.

Syswerda, G. (1989). Uniform Crossover in Genetic Algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms*, pp. 2-9. San Mateo CA: Morgan Kaufmann.

Syswerda, G. (1991a). A Study of Reproduction in Generational and Steady-State Genetic Algorithms. In G. Rawlins (ed.), *Foundations of Genetic Algorithms*, pp. 94-101. San Mateo CA: Morgan Kaufmann.

Syswerda, G. (1991b). Schedule Optimization Using Genetic Algorithms. In L. Davis (ed.), *Handbook of Genetic Algorithms*, pp. 332-349. Van Nostrand Reinhold.

Tanaka, T. (1993). Parsing Electronic Circuits in a Logic Grammar. *IEEE Trans. on Knowledge and Data Engineering*, **5**, pp. 225-239.

Tanese, R. (1989). Distributed Genetic Algorithms. In J. D. Schaffer (ed.), *Proceedings of the Third International Conference on Genetic Algorithms*, pp. 434-439. San Mateo CA: Morgan Kaufmann.

Tangkitvanich, S. and Shimura, M. (1992). Refining a Relational Theory with Multiple Faults in the Concept and Subconcepts. In *Proceedings of the Ninth International Conference on Machine Learning*, pp. 436-444. San Mateo, CA: Morgan Kaufmann.

Tesauro, G. (1992). Practical Issues in Temporal Difference Learning. *Machine Learning*, **8**, pp. 257-277.

Vose, M. (1993). Modeling Simple Genetic Algorithms. In D. Whitley (ed.), *Foundations of Genetic Algorithms 2*, pp. 63-73. San Mateo CA: Morgan Kaufmann.

Vose, M. and Liepins, G. (1991). Punctuated Equilibria in Genetic Search. *Complex Systems*, **5**, pp. 31-44.

Whitley, D. (1989). The GENITOR Algorithm and Selective Pressure. In *Proceedings of the Third International Conference on Genetic Algorithms*, pp. 116-121. San Mateo CA: Morgan Kaufmann.

Whitley, D. (1993). An Executable Model of a Simple Genetic Algorithm. In D. Whitley (ed.), *Foundations of Genetic Algorithms 2*, pp. 45-62. San Mateo CA: Morgan Kaufmann.

Whitley, D., Das, R., and Crabb, C. (1992). Tracking Primary Hyperplane Competitors During Genetic Search. *Annals of Mathematics and Artificial Intelligence*, **6**, pp. 367-388.

Whitley, D., Starkweather, T. (1990). Genitor II: a Distributed Genetic Algorithm. *Journal of Experimental and Theoretical Artificial Intelligence*, **2**, pp. 189-214.

Winston, P. H. (1975). Learning structural descriptions from examples. In P. H. Winston (ed.), *The psychology of computer vision.* New York: McGraw-Hill.

Wirth, R. (1989). Completing Logic Programs by Inverse Resolution. In *Proceedings of the Fourth European Working Session on Learning*, pp. 239-250. London: Pitman.

Wong, M. L. and Leung, K. S. (1994a). Inductive Logic Programming Using Genetic Algorithms. In J. W. Brahan and G. E. Lasker (Eds.), *Advances in Artificial Intelligence - Theory and Application II*, 119-124. I.I.A.S., Ontario.

Wong, M. L. and Leung, K. S. (1994b). Learning First-order Relations from Noisy Databases using Genetic Algorithms. In *Proceedings of the Second Singapore International Conference on Intelligent Systems*, B159-164.

Wong, M. L. and Leung, K. S. (1995a). An adaptive Inductive Logic Programming system using Genetic Programming. In *Proceedings of the Fourth Annual Conference on Evolutionary Programming*. MA: MIT Press.

Wong, M. L. and Leung, K. S. (1995b). Genetic Logic Programming and Applications. *IEEE Expert.*

Zelle, J. M., Mooney, R. J. and Konvisser, J. B. (1994). Combining Top-down and Bottom-up Techniques in Inductive Logic Programming. Technical Report, Department of Computer Science, University of Texas.

# Appendix A
# Some procedural search biases induced by the Adaptive ILP system

## 1. Biases (with `current-information` and `Info`)

```
(% (+ (% $n^-_{i+1}$ CURRENT-INFORMATION)
      (- CURRENT-INFORMATION 1.9389733))
   (* (- (+ $n^-_{i+1}$ $n^+_{i+1}$) (* $n^-_{i+1}$ -1.2297009)) $n^-_{i+1}$))
```

```
(INFO
 (+
  (PROTECTED-LOG
   (+ CURRENT-INFORMATION
      (INFO
       (*
        (% (PROTECTED-LOG CURRENT-INFORMATION)
         (% CURRENT-INFORMATION CURRENT-INFORMATION))
        (- (+ CURRENT-INFORMATION CURRENT-INFORMATION)
           (+ -2.415241 CURRENT-INFORMATION)))
       $n_{i+1}^{-}$)))
  $n_{i+1}^{-}$)
 (+
  (+
   (+
    (INFO
     (+
      (PROTECTED-LOG
       (+ CURRENT-INFORMATION
          (+ (PROTECTED-LOG CURRENT-INFORMATION)
             CURRENT-INFORMATION)))
      (% $n_{i+1}^{-}$ $n_{i+1}^{-}$))
     (+
      (+ (PROTECTED-LOG $n_{i+1}^{-}$)
         (INFO
          (%
           (INFO $n_{i+1}^{-}$
            (- (INFO CURRENT-INFORMATION -0.17352863)
               CURRENT-INFORMATION))
           2.477609)
          3.603527))
      (PROTECTED-LOG
       (+ (+ (+ $n_{i+1}^{-}$ -1.8040327) (INFO $n_{i+1}^{+}$ $n_{i+1}^{-}$))
          $n_{i}^{++}$))))
     $n_{i}^{++}$)
    (INFO (% -3.51194 2.477609) 3.603527))
   $n_{i+1}^{-}$))
```

## 2. Biases (without `current-information` and `Info`)

```
(-
 (PROTECTED-LOG
  (+
   (* (+ $n_i^+$ (- (% $n_{i+1}^+$ $n_{i+1}^-$) $n_i^{++}$)) 3.1308892)
   $n_i^+$))
 (%
  (* $n_{i+1}^-$
     (PROTECTED-LOG
      (* (+ (% $n_{i+1}^-$ $n_i^+$) $n_i^{++}$)
         (+ (% $n_{i+1}^-$ $n_i^-$) $n_i^-$))))
  (* $n_{i+1}^+$ $n_i^{++}$)))
```