

Chapter 2

INDUCTIVE GENETIC PROGRAMMING

Inductive Genetic Programming (IGP) is a specialization of the Genetic Programming (GP) paradigm [Koza, 1992, Koza, 1994, Koza et al., 1999, Koza et al., 2003, Banzhaf et al., 1998, Langdon and Poli, 2002, Rilo and Worzel, 2003] for inductive learning. The reasons for using this specialized term are: 1) *inductive* learning is a search problem and GP is a versatile framework for exploration of large multidimensional search spaces; 2) GP provides *genetic* learning operators for hypothetical model sampling that can be tailored to the data; and 3) GP manipulates *program*-like representations which adaptively satisfy the constraints of the task. An advantage of inductive GP is that it discovers not only the parameters but also the structure and size of the models.

The basic computational mechanisms of a GP system are inspired by those from natural evolution. GP conducts a search with a population of models using mutation, crossover and reproduction operators. Like in nature, these operators have a probabilistic character. The mutation and crossover operators choose at random the model elements that will undergo changes, while the reproduction selects random good models among the population elite. Another characteristic of GP is its flexibility in the sense that it allows us to easily adjust its ingredients for the particular task. It enables us to change the representation, to tune the genetic operators, to synthesize proper fitness functions, and to apply different reproduction schemes.

This chapter demonstrates how to incorporate these mechanisms into a basic IGP framework so as to organize efficient stochastic search of vast hypothesis spaces. A subject of particular interest is the automatic synthesis of tree-structured polynomial networks by IGP. The computational mechanisms of IGP are general to a great degree and similar

to those of traditional GP, so they may be used directly for processing binary tree-like models in other systems for inductive learning. If necessary, the components of the presented framework may be easily modified because of their simplicity. Special attention is given to making efficient mutation and crossover learning operators. Differences between encoded and decoded tree implementations of the PNN models are explained.

2.1 Polynomial Neural Networks (PNN)

Polynomial neural networks are a class of feed-forward neural networks. They are developed with the intention of overcoming the computational limitations of the traditional statistical and numerical optimization tools for polynomial identification, which can only practically identify the coefficients of relatively low-order terms. The adaptive PNN algorithms are able to learn the weights of highly nonlinear models.

A PNN consists of nodes, or neurons, linked by connections associated with numeric weights. Each node has a set of incoming connections from other nodes and one (or more) outgoing connections to other nodes. All nonterminal nodes, including the fringe nodes connected to the inputs, are called hidden nodes. The input vector is propagated forward through the network. During the forward pass it is weighted by the connection strengths and filtered by the activation functions in the nodes, producing an output signal at the root. Thus, the PNN generates a nonlinear real valued mapping $P : \mathcal{R}^d \rightarrow \mathcal{R}$, which taken from the network representation, is a *high-order polynomial model*:

$$P(\mathbf{x}) = a_0 + \sum_{i=1}^L a_i \prod_{j=1}^d x_j^{r_{ji}} \quad (2.1)$$

where a_i are the term coefficients, i ranges up to a preselected maximum number of terms L : $i \leq L$; x_j are the values of the independent variables arranged in an input vector \mathbf{x} , i.e. $j \leq d$ numbers; and $r_{ji} = 0, 1, \dots$ are the powers with which the j -th element x_j participates in the i -th term. It is assumed that r_{ji} is bounded by a maximum polynomial order (degree) s : $\sum_{j=1}^d r_{ji} \leq s$ for every i . The polynomial (2.1) is linear in the coefficients a_i , $1 \leq i \leq L$, and nonlinear in the variables x_j , $1 \leq j \leq d$. It should be noted that equation (2.1) provides a finite format for representing the power series expansion (1.1).

Strictly speaking, a power series contains an infinite number of terms that can represent a function exactly. In practice a finite number of them is used for achieving the predefined sufficient accuracy. The polynomial size is manually fixed by a design decision.

2.1.1 PNN Approaches

There are various approaches to making PNN models: 1) neural network implementations of discrete Volterra models [Wray and Green, 1994]; 2) multilayer perceptron networks with a layer of polynomial activation functions [Marmarelis and Zhao, 1997]; 3) linear networks of polynomial terms [Holden and Rayner, 1992, Liu et al., 1998, Pao, 1989, Rayner and Lynch, 1989]; 4) polynomially modelled multilayer perceptrons [Chen and Manry, 1993]; 5) Pi-Sigma [Gosh and Shin, 1992, Shin and Ghosh, 1995] and Sigma-Pi neural networks [Heywood and Noakes, 1996]; 6) Sigma-Pi neural trees [Zhang and Mühlenbein, 1995, Zhang et al., 1997]; and 6) hierarchical networks of cascaded polynomials [Barron, 1988, Elder and Brown, 2000, Farlow, 1984, Green et al., 1988, Ivakhnenko, 1971, Madala and Ivakhnenko, 1994, Ng and Lippmann, 1991, Pham and Liu, 1995, Müller and Lemke, 2000]. These PNN approaches are attractive due to their universal approximation abilities according to the Stone-Weierstrass theorem [Cotter, 1990], and their generalization power measurable by the Vapnik-Chervonenkis (VC) dimension [Anthony and Holden, 1994].

The differences between the above PNN are in the representational and operational aspects of their search mechanisms for identification of the relevant terms from the power series expansion, including their weights and underlying structure. The main differences concern: 1) what is the polynomial network topology and especially what is its connectivity; 2) which activation polynomials are allocated in the network nodes for expressing the model; are they linear, quadratic, or highly nonlinear mappings in one or several variables; 3) what is the weight learning technique; 4) whether there are designed algorithms that search for the adequate polynomial network structure; and 5) what criteria for evaluation of the data fitting are taken for search control.

An interesting approach to building discrete Volterra models is to assume that the hidden node outputs of a neural network are polynomial expansions [Wray and Green, 1994]. In this case, the weights of these hidden node polynomials can be estimated with a formula derived using the Taylor series expansion of the original node output function. Then the network weights are obtained by backpropagation training. After convergence, the coefficients of the polynomial expansion and the kernels of the Volterra model are calculated.

The multilayer networks with one hidden layer of polynomial activation functions [Marmarelis and Zhao, 1997] are enhancements of the multilayer perceptrons that offer several advantages: they produce more compact models in which the number of hidden units increases more slowly with the increase of the inputs, and they yield more accurate

models. Their common shortcoming is the need to fix in advance the network structure. The idea behind Separable Volterra Networks (SVN) [Marmarelis and Zhao, 1997] is to directly simulate the Kolmogorov-Lorentz theorem for building a continuous function by a two-layer architecture. SVN uses a linear function at the output node and univariate polynomials at the hidden layer nodes. The univariate polynomials in the hidden layer make the higher-order monomials of the model. SVN trains only the weights from the input to the hidden layer.

The linear networks of polynomial terms [Holden and Rayner, 1992, Liu et al., 1998, Pao, 1989, Rayner and Lynch, 1989] increase horizontally a single hidden layer with monomials. The Functional Link Net (FLNet) [Pao, 1989] expands a single hidden layer network with heuristically selected high-order functions. These functions are monomials, called functional links. Unfortunately this algorithm attempts to make power series expansions directly and suffers from a combinatorial explosion of nodes. The Volterra Connectionist Model (VCM) [Holden and Rayner, 1992, Rayner and Lynch, 1989] makes linear networks in a similar way by extending the input variables using non-linear polynomial basis functions. Another similar approach is the Volterra Polynomial Basis Function (VPBF) [Liu et al., 1998] network which adds nodes in a stepwise manner using an orthogonal least squares algorithm.

The output of a MLP network can be expressed as a polynomial function of the inputs [Chen and Manry, 1993]. Such a MLP is trained by backpropagation and a matrix model of the node activation functions is created, assuming that a node output is a finite degree polynomial basis function. The coefficients of these alternative polynomial basis functions, which simulate the hidden node outputs, are estimated by mean square error minimization. This approach yields transparent polynomials however it suffers from the same disadvantages as the MLP, so it may be regarded simply as a model extraction technique.

The Pi-Sigma Networks [Gosh and Shin, 1992] have a hidden layer of linear summing units and a product unit in the output layer. Such a network produces a high-order polynomial whose maximal degree depends on the size of the hidden layer. The efficacy of PSN is that it requires us to train only the weights to the hidden layer as there are no weights from the hidden nodes to the output node. PSN networks have a smaller number of weights compared to the multilayer perceptron networks. The problem is that a single PSN is not a universal approximator. Ridge Polynomial Networks (RPN) [Shin and Ghosh, 1995] offer a remedy to this problem, providing enhanced approximation abilities. The RPN is a linear combination of PSN using ridge polynomials which can represent any multivariate polynomial.

Sigma-Pi neural networks [Heywood and Noakes, 1996] are a kind of MLP networks with summation and multiplicative units that are trained with a backpropagation algorithm for higher-order networks. The Sigma-Pi networks have polynomials as net functions in the summation units which are passed through sigmoids to feed-forward the nodes in the next layer. These networks usually implement only a subset from the possible monomials so as to avoid the curse of dimensionality. Their distinguishing characteristics are the dynamic weight pruning of redundant units while the network undergoes training, and the use of different learning rates for each monomial in the update rule.

Sigma-Pi Neural Trees (SPNT) [Zhang and Mühlenbein, 1995, Zhang et al., 1997] are high-order polynomial networks of summation and multiplication units. The sigma units perform weighted summation of the signals from the lower feeding nodes, and the product units multiply the weighted incoming signals. The neural trees may have an arbitrary but predefined number of incoming connections, and also an arbitrary but predetermined tree depth. SPNT provides the idea to construct irregular polynomial network structures of sigma and product units, which can be reused and maintained in a memory efficient sparse architecture. A disadvantage of this approach is that it searches for the weights by a genetic algorithm which makes its operation slow and inaccurate.

The multilayer GMDH polynomial networks [Barron, 1988, Elder and Brown, 2000, Farlow, 1984, Green et al, 1988, Ivakhnenko, 1971, Madala and Ivakhnenko, 1994, Müller and Lemke, 2000, Ng and Lippmann, 1991, Pham and Liu, 1995] are more suitable than the other networks for intensive evolutionary search. First, taken separately a polynomial is simply a binary tree structure which is easy to manipulate by IGP. Second, GMDH offers the opportunity to learn the weights rapidly by least squares fitting at each node. However, such weights are locally optimal and admit further coordination by additional training with gradient-descent and probabilistic tuning with Bayesian techniques.

2.1.2 Tree-structured PNN

The models evolved by IGP are genetic programs. IGP breeds a population \mathcal{P} of genetic programs $\mathcal{G} \in \mathcal{P}$. The notion of a *genetic program* means that this is a sequence of instructions for computing an input-output mapping. The main approaches to encoding genetic programs are: 1) tree structures [Koza, 1992]; 2) linear arrays [Banzhaf et al., 1998]; and 3) graphs [Teller and Veloso, 1996]. The tree-like genetic programs originate from the expressions in functional programming languages where an expression is arranged as a tree of elementary functions in its nodes and variables in its leaves. The linear genetic programs are

linear arrays of instructions, which can be written in terms of a programming language or written in machine code. The graph-based programs are made as directed graphs with stacks for their processing and memory for the variables. The edges in the graph determine the sequence for execution of the programs. Each node contains the function to be performed and a pointer to the next instruction.

Tree-like genetic programs are suitable for IGP as they offer two advantages: 1) they have parsimonious topology with sparse connectivity between the nodes, and 2) they enable efficient processing with classical algorithms. Subjects of particular interest here are the linear genetic program trees that are genotypic encodings of PNN phenotypes which exhibit certain input-output behaviors.

The Tree-like Representation. A genetic program has a *tree* structure. In it a node is below another node if the other node lies on the path from the root to this node. The nodes below a particular node are a subtree. Every node has a parent above it and children nodes under it. Nodes without children are leaves or terminals. The nodes that have children are nonterminals or functional nodes.

PNN are represented with *binary trees* in which every internal functional node has a left child and a right child. A binary tree with Z functional nodes has $Z + 1$ terminals. The nodes are arranged in multiple levels, also called *layers*. The level of a particular node is one plus the level of its parent, assuming that the root level is zero. The *depth*, or height of a tree, is the maximal level among the levels of its nodes. A tree may be limited by a maximum tree depth or by a maximum tree size, which is the number of all nodes and leaves.

Trees are described here formally to facilitate their understanding. Let \mathcal{V} be a vertex set from *functional nodes* \mathcal{F} and *terminal leaves* \mathcal{T} ($\mathcal{V} = \mathcal{F} \cup \mathcal{T}$). A *genetic program* \mathcal{G} is an ordered tree $s_0 \equiv \mathcal{G}$, in which the sons of each node \mathcal{V} are ordered, with the following properties:

- it has a distinguishing parent $\rho(s_0) = \mathcal{V}_0$, called the *root* node;
- its nodes are labelled $\nu : \mathcal{V} \rightarrow \mathcal{N}$ from left to right and $\nu(\mathcal{V}_i) = i$;
- any functional node has a number of children, called arity $\kappa : \mathcal{V} \rightarrow \mathcal{N}$, and a terminal leaf $\rho(s_i) = \mathcal{T}_i$ has zero arity $\kappa(\mathcal{T}_i) = 0$;
- the children of a node \mathcal{V}_i , with arity $k = \kappa(\mathcal{V}_i)$, are roots of disjoint subtrees $s_{i1}, s_{i2}, \dots, s_{ik}$. A subtree s_i has a root $\rho(s_i) = \mathcal{V}_i$, and subtrees s_{i1}, \dots, s_{ik} at its k children: $s_i = \{(\mathcal{V}_i, s_{i1}, s_{i2}, \dots, s_{ik}) \mid k = \kappa(\mathcal{V}_i)\}$.

This vertex labeling suggests that the subtrees below a node \mathcal{V}_i are ordered from left to right as the leftmost child s_{i1} has smallest label $\nu(s_{i1}) < \nu(s_{i2}) < \dots < \nu(s_{ik})$. This ordering of the nodes is necessary for making efficient tree implementations, as well as for the design of proper genetic learning operators for manipulation of tree structures.

The construction of binary tree-like PNN requires us to instantiate its parameters. The terminal set includes the explanatory input variables $\mathcal{T} = \{x_1, x_2, \dots, x_d\}$, where d is the input dimension. The function set contains the activation polynomials in the tree nodes $\mathcal{F} = \{p_1, p_2, \dots, p_m\}$, where the number m of distinct functional nodes is given in advance. A reasonable choice are the incomplete bivariate polynomials up to second-order that can be derived from the complete one (1.2) assuming that some of its coefficients are zero. The total number of such incomplete polynomials is 25 from all $2^5 - 1$ possible combinations of monomials $w_i h_i(x_i, x_j)$, $1 \leq i \leq 5$, having always the leading constant w_0 , and two different variables. A subset $p_i \in \mathcal{F}$, $1 \leq i \leq 16$ of them is taken after elimination of the symmetric polynomials (Table 2.1).

Table 2.1. Activation polynomials for genetic programming of PNN.

1.	$p_1(x_i, x_j) = w_0 + w_1 x_1 + w_2 x_2 + w_3 x_1 x_2$
2.	$p_2(x_i, x_j) = w_0 + w_1 x_1 + w_2 x_2$
3.	$p_3(x_i, x_j) = w_0 + w_1 x_1 + w_2 x_1 x_2$
4.	$p_4(x_i, x_j) = w_0 + w_1 x_1 + w_2 x_1 x_2 + w_3 x_1^2$
5.	$p_5(x_i, x_j) = w_0 + w_1 x_1 + w_2 x_2^2$
6.	$p_6(x_i, x_j) = w_0 + w_1 x_1 + w_2 x_2 + w_3 x_1^2$
7.	$p_7(x_i, x_j) = w_0 + w_1 x_1 + w_2 x_1^2 + w_3 x_2^2$
8.	$p_8(x_i, x_j) = w_0 + w_1 x_1^2 + w_2 x_2^2$
9.	$p_9(x_i, x_j) = w_0 + w_1 x_1 + w_2 x_2 + w_3 x_1 x_2 + w_4 x_1^2 + w_5 x_2^2$
10.	$p_{10}(x_i, x_j) = w_0 + w_1 x_1 + w_2 x_2 + w_3 x_1 x_2 + w_4 x_1^2$
11.	$p_{11}(x_i, x_j) = w_0 + w_1 x_1 + w_2 x_1 x_2 + w_3 x_1^2 + w_4 x_2^2$
12.	$p_{12}(x_i, x_j) = w_0 + w_1 x_1 x_2 + w_2 x_1^2 + w_3 x_2^2$
13.	$p_{13}(x_i, x_j) = w_0 + w_1 x_1 + w_2 x_1 x_2 + w_3 x_2^2$
14.	$p_{14}(x_i, x_j) = w_0 + w_1 x_1 + w_2 x_2 + w_3 x_1^2 + w_4 x_2^2$
15.	$p_{15}(x_i, x_j) = w_0 + w_1 x_1 x_2$
16.	$p_{16}(x_i, x_j) = w_0 + w_1 x_1 x_2 + w_2 x_1^2$

The notion of *activation polynomials* is considered in the context of PNN instead of transfer polynomials to emphasize that they are used to derive backpropagation network training algorithms (Chapter 6).

The motivations for using all distinctive *complete* and *incomplete* (first-order and second-order) bivariate activation polynomials in the network nodes are: 1) having a set of polynomials enables better identification of the interactions between the input variables; 2) when composed, higher-order polynomials rapidly increase the order of the overall model, which causes overfitting even with small trees; 3) first-order and second-order polynomials are fast to process; and 4) they define a search space of reasonable dimensionality for the GP to explore. The problem of using only the complete second-order bivariate polynomial (1.2) is that the weights of the superfluous terms do not become zero after least squares fitting, which is an obstacle for achieving good generalization.

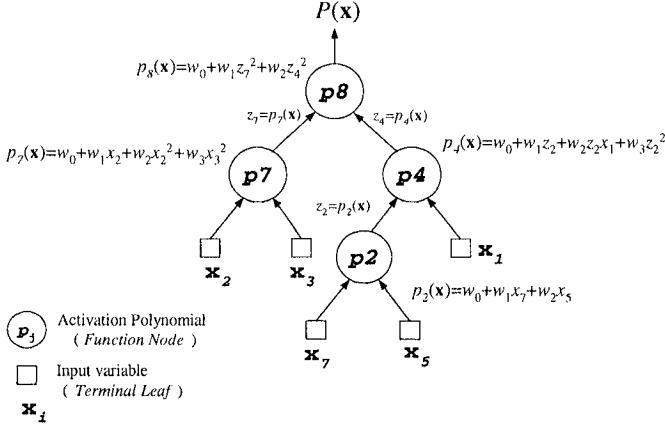


Figure 2.1. Tree-structured representation of a PNN.

Figure 2.2. illustrates a hierarchically composed polynomial extracted from the PNN in Figure 2.1 to demonstrate the transparency and easy interpretability of the obtained model.

Hierarchically Composed Polynomial

$$\begin{aligned}
 &((w_0 + w_1 * z_7^2 + w_2 * z_4^2) \\
 & \quad z_7 = (w_0 + w_1 * x_2 + w_2 * x_2^2 + w_3 * x_3^2) \\
 & \quad \quad x_2 \\
 & \quad \quad x_3) \\
 & \quad z_4 = (w_0 + w_1 * z_2 + w_2 * z_2 * x_1 + w_3 * z_2^2) \\
 & \quad \quad z_2 = (w_0 + w_1 * x_7 + w_2 * x_5) \\
 & \quad \quad \quad x_7 \\
 & \quad \quad \quad x_5) \\
 & \quad x_1)
 \end{aligned}$$

Figure 2.2. Polynomial extracted from the tree-structured PNN in Figure 2.1.

The accommodation of a set of complete and incomplete activation polynomials in the network nodes makes the models versatile for adaptive search, while keeping the neural network architecture relatively compact. Using a set of activation polynomials does not increase the computational demands for performing genetic programming. The benefit of having a set of activation polynomials is of enhancing the expressive power of this kind of PNN representation.

An example of a tree-structured polynomial using some of these activation polynomials is illustrated in Figure 2.1. The computed polynomial $P(\mathbf{x})$ at the output tree root is the multivariate composition: $P(x_1, x_2, x_3, x_5, x_7) = p_8(p_7(x_2, x_3), p_4(p_2(x_7, x_5), x_1))$.

The Search Space of Binary Trees. The number of terminals, instantiated by the input variables, and the number of functional nodes, instantiated by the activation polynomials, is important for the evolutionary learning of PNN because they determine the search space size and, thus, influence the probability for finding good solutions. When conducting evolutionary search by IGP, the size of the search space of tree-structures may be controlled by varying the number of activation polynomials or by the number of input variables.

Let the tree nodes be labelled by \mathcal{F} , which is the number of the hidden and fringe nodes. Let the tree leaves be labelled by \mathcal{T} , which is the number of the input variables that are passed through the leaves. Then, the number of different binary trees up to a predefined depth S is obtained by the following recursive formula [Ebner, 1999]:

$$\begin{aligned} Trees(0) &= \mathcal{T} \\ Trees(S) &= \mathcal{F}.Trees^2(S-1) + \mathcal{T} \end{aligned} \quad (2.2)$$

where $Trees(S)$ is the number of binary trees of depth up to S .

The tree depth impacts the search space size, but it should be noted that it also affects the convergence properties of polynomials. The maximal tree depth may be defined as a logarithmic function of the maximal order so as to restrict the network size: $S - 1 = \log_2(MaxOrder)$. The most probable maximal polynomial order $MaxOrder$ may be determined in advance and considered as a tree depth limit to constrain the IGP search space. The $MaxOrder$ can be found by increasing the order of a randomly generated PNN, and measuring its error on the concrete sample: $\sigma^2 = \sum_{n=1}^N (y_n - P(\mathbf{x}_n))^2 / (N - W - 1)$, where W is the number of the model weights. The denominator makes the error increase if the model becomes larger than the most probable degree.

Linear Implementation of PNN Trees. The design of tree-structured polynomial networks involves three issues: 1) how to represent the tree nodes; 2) how to represent the topological connections between the nodes; and 3) how to perform tree traversal. Taking these issues into consideration is crucial with respect to memory and time efficiency, as they impact the design of IGP systems.

From an implementation point of view the topology of a PNN tree can be stored as: a pointer-based tree, a linear tree in prefix notation, or a linear tree in postfix notation [Keith and Martin, 1994]. Pointer-based trees are such structures in which every node contains pointers to its children or inputs. Such pointer-based trees are easy to develop and manipulate; for example a binary tree can be traversed using double recursion. The problem of pointer-based trees is that the connections between the nodes are directly represented by pointers which incurs a

lot of processing time. The reason is that most of the contemporary programming language implementations make these pointers to address dynamic memory locations, and their reference in run time is time consuming. The operating systems usually arrange the dynamic memory in different data segments which increases the time overhead for fetching data. That is why the speed of manipulating pointer-based tree structures may be several times slower than linearized trees.

Linear trees encapsulate nodes in arrays where the parent nodes precede the children nodes [Banzhaf et al., 1998]. Tree traversal of such linearized trees is made using stacks or recursion. The nodes in a linear tree can be allocated using prefix, infix or postfix notation. Most suitable for GP are the prefix and postfix notations. The prefix notation arranges the nodes as sequences that are traversed in the following order: parent, left subtree, and right subtree. *Linear trees* in prefix notation are preferred for GP because they enable us to make fast genetic learning operators that operate on array representations. Here, *prefix trees* for encoding PNN models are adopted in correspondence to the chosen labeling. The prefix trees are evaluated by recursive preorder tree traversal. The recursive preorder tree traversal visits a node, and then visits recursively its left and right subtrees.

Linearized Tree Genotypes. The tree implementation of PNN influences the development of IGP, and thus impacts the structural search process. This is because the implementation determines which neighboring trees can be sampled, that is it determines the search directions. The linearized tree-structures are genotypic representations of the PNN models, and they do not directly affect their phenotypic characteristics. One may envision that each variable length linear genotype has a corresponding PNN phenotype. The trees are means for representing the structures of PNN and they determine the sampling of polynomial models. However, they also indirectly influence their approximation qualities.

The linearized tree implementation can be described in biological terms in order to explain how the development of computational IGP mechanisms reflects knowledge about natural evolution. IGP relies on notions from biology, but they are only loose associations with the originals because its mechanisms only loosely simulate their biological counterparts in natural organisms.

The genotypic encoding of a genetic program is called a *genome*. The genome is a kind of a linear array of *genes* and has a variable length. In the case of IGP, the genome is a linearly implemented tree. The genes in the genome are labelled by *loci*. The position of each gene within the genome is its *locus*. A locus actually corresponds to the node label $\nu(\mathcal{V}_i)$, $\nu : \mathcal{V} \rightarrow \mathcal{N}$ of the particular tree node \mathcal{V}_i . The value of the

node \mathcal{V}_i , which could be either an activation polynomial function \mathcal{F} or a terminal \mathcal{T} , is called an *allele*. The alleles of the functions are in the range $[1, m]$ when $\mathcal{F} = \{p_1, p_2, \dots, p_m\}$, while the alleles of terminals are in the range $[1, d]$ when $\mathcal{T} = \{x_1, x_2, \dots, x_d\}$. In the case of a functional node, the gene carries the kind of the activation polynomial in its allele and its weights. In the case of a terminal node, the allele is the index for accessing the corresponding input variable from the example database. The tree traversal algorithm examines the tree by visiting its loci and fetches its alleles when necessary to evaluate the fitness.

2.2 IGP Search Mechanisms

IGP performs evolutionary search that explores global as well as local search regions. The power of IGP is due to several characteristics that distinguish it from the traditional search algorithms: 1) it manipulates the encoded implementation of the genetic program-like models rather than the individual models directly; that is the search is conducted in the genotype space; 2) it performs search simultaneously with a population of candidate solutions; and 3) the search is probabilistic and guided by the fitnesses of the individuals. The stochastic nature of IGP is a consequence of the nondeterministic changes of the population and the randomized selection of promising individuals.

IGP is intentionally developed to have characteristics that loosely mimic the principles of natural evolution. The key idea is to make computational mechanisms that operate like the corresponding biological mechanisms in nature. The ultimate goal behind the inspiration from nature is to pursue the powerful learning ability of the evolution process. Simulation of the natural evolutionary processes is organized at three levels: in the *genotype space* of linear tree representations, in the *phenotype space* of tree-structured PNN models, and in the *fitness space* which is the fitness landscape of PNN fitness values. The genetic learning operators act on the linear tree genotypes. Thus, genetic program-like PNN phenotypes are sampled. These PNN phenotypes are distinguished by their properties, e.g. the fitness. The computational IGP system examines the genotype search space with the intention of finding phenotypic solutions with desired fitnesses.

The evolutionary IGP search has two aspects: *navigation*, carried by the genetic sampling and selection operators, and *landscape*, determined by the fitness function and the variable length representation. There are two main genetic sampling operators: recombination, also called crossover, and mutation. They sample polynomials by probabilistically modifying their trees. The selection operator directs the search by randomly choosing and promoting elite individuals having high fitness. The

search navigation moves the population on a landscape surface built of the genetic program fitnesses. The sampling and selection operators should push the population on the fitness landscape. The IGP mechanisms together should have the capacity to guide the population toward very deep landscape basins of good solutions.

2.2.1 Sampling and Control Issues

A critical problem in evolutionary IGP is the enormous dimensionality of the search space. In order to organize an efficient search process, the above two issues should be carefully analyzed.

The first issue is to make such mutation and crossover operators that can potentially visit every landscape region. These are also called learning operators because they sample individuals and thus contribute to finding the model structure. The learning operators for tree structures should be general and should not restrict the representation. These operators should avoid genetic program tree growth, known as bloating phenomenon, which worsens the IGP performance [Koza, 1992, Banzhaf et al., 1998, Langdon and Poli, 2002]. This is difficult to achieve because the tree growth usually implies improvement in fitness. Operators that allow tree bloat are unable to push the population progressively to promising landscape areas and cause search stagnation.

The second issue is that the population flow on the fitness landscape strongly depends on how the landscape has been created. The design of the fitness function can tune the landscape and mitigate the search difficulties. Fitness functions, fitness landscapes and their measures are investigated in separate chapters (Chapters 4 and 5).

A distinguishing feature of IGP is that its search mechanisms are mutually coordinated so as to avoid degenerated search. The size of the genetic programs serves as a common coordinating parameter. Size-dependant crossover, size-dependant mutation, and selection operators (that also depend on the tree size through their fitnesses) are designed. Making a common size biasing of the sampling operators and the fitness helps to achieve continuously improving behavior.

2.2.2 Biological Interpretation

The development of IGP systems follows the principles of natural evolution [Fogel, 1995, Paton, 1997]. Natural evolution acts on individuals having varying hereditary traits causing survival and reproduction of the fittest. Evolution is a term denoting changes of a population of individuals, called chromosomes, during successive generations. A chromosome plays the function of a genome and is a sequence of genes. The chromo-

some have different lengths and shapes as they have different portions of traits. The hereditary traits are carried by the genes. The genes can be separated from the traits that they specify using the notion of a genotype. Genotype means the genes in an individual chromosome structure. The observable traits of an individual are referred to using the notion of a phenotype. Thus, the notions of a genotype and phenotype serve to make a distinction between genes and the traits that they carry. A gene can have different molecular forms that indicate different information about the traits called alleles.

Evolution keeps the most common alleles in the population and discards the less common alleles. The evolution involves updating the allele frequencies through the generations. The alleles in the population undergo modifications by several mechanisms: natural selection, crossover, and mutation. Natural selection reproduces individuals; it exchanges genetic material between the individuals in two populations during successive generations. Natural selection may have different effects on the population: 1) stabilization occurs when it maintains the existing range of alleles in the population, that is when it retains the individuals that contain the most common alleles; 2) shifting happens when it promotes individuals whose range of alleles changes in certain directions; and 3) disruption results when it deliberately favors concrete traits while destroying others. The crossover shuffles individuals, while the mutation changes the alleles.

Individuals in nature compete to survive during the generations. The successful individuals produce increasing numbers of offspring, while the unsuccessful ones produce less or no offspring. Natural selection picks more and more fit individuals, which when reproduced and mutated, lead to even better descendants. The adaptation of the individuals is not perfect. Their continuous evolution is driven probabilistically with respect to their fitness by selection. The population is in permanent movement along the generations.

Individuals are simulated in IGP by the genetic programs. In our case, these are the tree-structured PNNs. Each individual is associated with a fitness measure of its potential to survive in the population. The fitness of an individual accounts for its ability to reproduce, mutate, and so to direct the search. There is a fitness function that maps a genetic program into its fitness value. The most essential property of a genetic program is its *fitness*. The fitness helps to computationally simulate the biological principle of survival of the fittest. The biologically inspired idea is to promote, with higher probability, the fitter PNNs. In this way, the population moves toward promising areas in the search space seeking to locate the globally best solution.

2.3 Genetic Learning Operators

When a tree is modified, neighboring trees are sampled. The two main modification operators for genetic learning have different roles: the mutation performs local search in the vicinity of the parent tree, while the crossover conducts global search of distant search space areas. Both learning operators should be considered in IGP so as to achieve exploration and exploitation of the tree search space. Good solutions can be found only if the genetic learning operators are properly developed so as to sample every structurally possible tree, because every tree is hypothetically likely to be a solution of the task.

The operators for linear trees have to meet several criteria in order to keep the structural consistency of the genetic programs: 1) they should preserve the parent-child relationships among the vertices; and 2) they should not change the prefix ordering $\nu(s_i) < \nu(s_j)$ between the vertices after applying a series of operators $\nu(M(s_i)) < \nu(M(s_j))$.

The operator has to sample only the closest neighbors of the selected node in order to attain high correlation between the parent and the offspring. This requires keeping of the inclusion property between the corresponding subtrees from the parent s_i and the offspring s_j trees: $s_j \subset M(s_i)$ or $s_i \subset M(s_j)$. Tree transformations that satisfy these three topological properties of the trees may be expected to improve the search if properly coordinated with the other IGP mechanisms.

2.3.1 Context-preserving Mutation

Mutation is a genetic learning operator that modifies one parent tree into one offspring. The mutation operator is considered efficient when it causes slight changes of the fitness after transforming a tree. In order to facilitate the evolutionary search process there should be maintained high correlation between the fitness of the parent and that of the offspring. Such a relation is called strong causality [Rosca and Ballard, 1995a, Igel, 1998, Sendhoff et al., 1997]. Having strong causality ensures continuous progress in evolutionary search.

The *context-preserving mutation* (CPM) operator is a means for organizing local search. This mutation edits a tree structure subject to three restrictions: 1) maintaining the approximate topology of the genetic program tree by keeping the representation relationships among the tree vertices; 2) preserving the inclusion property between the subtrees; and 3) affecting only the nearest tree vertices to the chosen mutation point. The genetic program trees shrink and grow slightly, which contributes to the overall improvement of the evolutionary search.

A genetic program with a tree-like structure \mathcal{G} of vertices \mathcal{V}_i , each of which has below a subtree $s_i = \{(\mathcal{V}_i, s_{i1}, s_{i2}, \dots, s_{ik}) \mid k = \kappa(\mathcal{V}_i)\}$, can be transformed by the following three *elementary context-preserving mutations* $M : \mathcal{G} \times \mathcal{V} \times \mathcal{V} \rightarrow \mathcal{G}'$, $M(s, \mathcal{V}_i, \mathcal{V}'_i) = s'$:

- *insert* M_I : adds a subtree $s'_i = \{(\mathcal{V}'_i, s'_{i1}, \mathcal{T}'_{i2}, \dots, \mathcal{T}'_{il}) \mid l = \kappa(\mathcal{V}'_i)\}$, so that the old subtree s_i becomes a leftmost child of the new subtree at the new node \mathcal{V}'_i , i.e. $s'_{i1} = s_i$;

- *delete* M_D : moves up the only subtree $s'_i = \{(\mathcal{F}'_i, s'_{i1}, \dots, s'_{il}) \mid 1 \leq l \leq \kappa(\mathcal{F}'_i)\}$ of s_i iff $\exists \mathcal{F}_{ij} = \rho(s_{ij})$, for some $1 \leq j \leq \kappa(\mathcal{V}_i)$ to become root $\mathcal{F}'_i = \mathcal{F}_{ij}$, and all other leaf children $\forall k, ik \neq j, \rho(s_{ik}) = \mathcal{T}_{ik}$, of the old \mathcal{V}_i are pruned. This deletion is applicable only when the node to be removed has one child subtree, which is promoted up;

- *substitute* M_S : replaces a leaf $\mathcal{T}_i = \rho(s_i)$, by another one \mathcal{T}'_i , or a functional $\mathcal{F}_i = \rho(s_i)$ by \mathcal{F}'_i . If the arity $\kappa(\mathcal{F}'_i) = k$, then $s'_i = \{(\mathcal{F}'_i, s_{i1}, \dots, s_{ik}) \mid k = \kappa(\mathcal{F}'_i)\}$. When $\kappa(\mathcal{F}'_i) = l$ only $l = k \pm 1$ is considered. In case $l = k + 1$ it adds a leaf $s'_i = \{(\mathcal{F}'_i, s_{i1}, \dots, s_{ik}, \mathcal{T}_{il})\}$ else in case $l = k - 1$ it cuts $s'_i = \{(\mathcal{F}'_i, s_{i1}, \dots, s_{il})\}$.

There are various mutation operators, but many of them do not help to achieve progressive search. Many mutation operators augment the tree or trim it by whole subtrees, such as the hierarchical variable length mutation (HVLm) [O'Reilly, 1995]. HVLm inserts a randomly generated subtree before a randomly chosen vertex, deletes a randomly chosen node so that its largest subtree is promoted to replace it, and substitutes a randomly chosen node by another one with the same arity. HVLm generates quite different offspring models from their parents.

A useful idea is to develop a *uniform replacement mutation operator* (URM). The URM operator successively traverses the tree nodes and changes every visited node or leaf with a probabilistically selected corresponding node or leaf. This operator allows us to make large steps in the search space because it makes larger tree changes than CPM. The difference between crossover and URM is that the latter is applied to a single tree and does not transfer material from another parent, rather it randomly updates the parent so as to produce an offspring.

Alternatively to the above transformation there could be made a *replacement mutation* (RM) which substitutes with a predefined probability each allele by a different, randomly chosen allele. While traversing the tree each functional node has the chance to be exchanged with another node, and each terminal leaf has the chance to be exchanged with another leaf. This RM operator may be applied after doing crossover so as to sustain the useful diversity of good alleles in the population.

Figure 2.3 displays an application of the context-preserving mutation operator to a concrete tree.

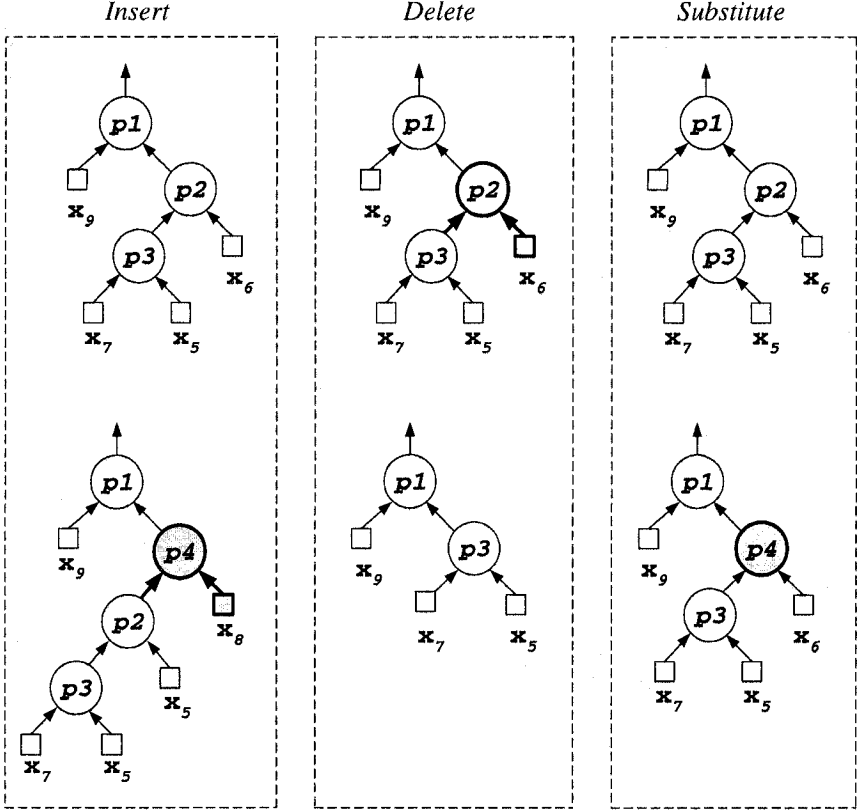


Figure 2.3. Context-preserving mutation (CPM) for tree-structured models.

2.3.2 Crossover Operator

The *crossover* operator should recombine node material by cutting or splicing two parent trees. Material exchange is made by selecting a cut point node in each tree, and swapping the subtrees rooted in the cut point nodes. The offspring trees may become larger than their parents. When a tree is of short size, it is not cut but added as a whole subtree at the crossover point in the other tree. If the two trees are short, they are spliced together. This crossover by cut and splice prevents the trees from rapid shrinking in the case of minimizing fitness functions. The recombination is restricted by a maximum tree size. This is a non-homologous crossover which does not preserve the topological positions of the swapped subtrees. Figure 2.4 illustrates an application of the crossover operator to two arbitrary trees.

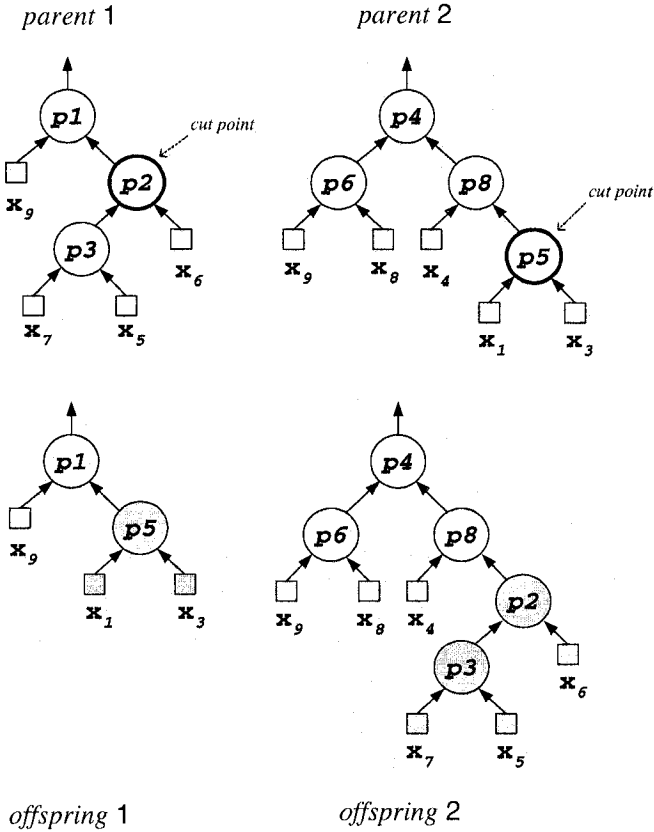


Figure 2.4. Crossover by cut and splice for tree-structured PNN models.

2.3.3 Size-biasing of the Genetic Operators

IGP conducts progressive search when the genetic learning operators climb easily on the fitness landscape. One aspect of the search control is to coordinate the fitness function and the genetic operators with a common bias parameter. The convergence theory of IGP (Section 2.6) suggests to use tree size biasing. The rationale is that small tree-like PNN usually do not fit the data well enough. Larger tree-like PNN exhibit better accuracy but do not necessarily generalize well on future data. While the fitness function only evaluates the genetic programs they need to be properly sampled. Early focusing to very small size or large size genetic programs can be avoided by size dependant navigation operators. The size-biased application of crossover and mutation helps to adequately counteract the drift to short or long programs.

The genetic learning operators should sample trees in such a way that the average tree size adapts without becoming very small or very large. Preferably small trees should be grown and large trees should be pruned. The operators have to prevent the tree bloat phenomenon which disturbs the evolutionary search [Banzhaf et al., 1998, Langdon and Poli, 2002]. Tree bloat occurs when the GP uses a fitness function that accounts only for the degree of fitting the data. When a tree expands, it pushes the search to a particular direction on the landscape which cannot be further avoided if there are no forces to shrink the tree for redirection. A similar disastrous search effect is the uncontrolled tree shrinking phenomenon. A population dominated by shrinking or growing genetic programs may drift to erroneous landscape areas where the evolutionary search stagnates.

The *size-biased mutation* operator for IGP performs context-preserving mutation with probability p_m defined as follows:

$$p_m = \mu \times |g|^2 \quad (2.3)$$

where μ is a free parameter [Goldberg et al., 1989], g is the linear implementation of the genetic program tree \mathcal{G} , and $|g|$ is its size. This operator usually modifies large trees.

The *size-biased crossover* operator for IGP splices or crosses two genetic program trees with probability p_c defined as follows:

$$p_c = \kappa / \sqrt{|g|} \quad (2.4)$$

where κ is a free parameter. More precisely, the probability whether to cut either of the trees is determined independently from the other. The cut points are randomly selected within the parents.

The free parameters serve as knobs with which one may carefully regulate the evolutionary search efficacy. The proper values for μ and κ may be identified with the autocorrelation function (Section 4.3.1).

2.3.4 Tree-to-Tree Distance

The development tools for IGP that manipulate trees should include an algorithm for estimating the distance between the trees. The topological similarity among trees is quantified by the *tree-to-tree distance* metric. Having an algorithm for computing the distance between the trees is useful because it can be applied for analysis and improvement of the GP mechanisms (Section 5.3).

The tree-to-tree distance is defined with respect to the basic concrete tree transformations which should be elaborated in advance. Distance $Dist(\mathcal{G}, \mathcal{G}')$ between two trees \mathcal{G} and \mathcal{G}' is the minimum number \sharp of

elementary mutations $M \in [M_I, M_D, M_S]$ required to convert one of the trees \mathcal{G} into the other \mathcal{G}' :

$$Dist(\mathcal{G}, \mathcal{G}') = \min\{\#\theta(M) | M \in [M_I, M_D, M_S], M(\mathcal{G}) = \mathcal{G}'\} \quad (2.5)$$

where θ is a unit distance function associated with each of the elementary mutations. The unit distance from making a substitution is $\theta(M_S(\mathcal{V}, \mathcal{G}, \mathcal{V}'))$ which can be 1 if $\mathcal{V} \neq \mathcal{V}'$, and otherwise 0. The unit distance from using either of the insertion or deletion suboperators is $\theta(M_I(\mathcal{V}, \mathcal{G})) \equiv \theta(M_D(\mathcal{V}, \mathcal{G})) = 1$ because they always modify the trees on which they have been applied, that is they always change the trees.

This tree-to-tree distance (2.5) is a distance metric as it meets the following mathematical requirements: 1) it is symmetric $Dist(\mathcal{G}, \mathcal{G}') = Dist(\mathcal{G}', \mathcal{G})$; 2) it is nonnegative $Dist(\mathcal{G}, \mathcal{G}') = 0$ only if $\mathcal{G} = \mathcal{G}'$; and, 3) it obeys the triangle inequality.

The algorithm for computing the distance between trees should reflect their implementation and structural relationships between the parent and child nodes. The PNNs manipulated by IGP are implemented as labelled trees using preorder notation. The preorder labelling determines not only the arrangement of the nodes in the tree but it preserves also their structure. Let \mathcal{V}_i be a node at position i in a tree. According to the preorder labeling, the following properties hold: 1) for each pair of labels $i_1 < i_2$, the node \mathcal{V}_{i_1} is an ancestor of node \mathcal{V}_{i_2} ; 2) for labels i which are related by the inequality $i_1 < i \leq i_2$, the nodes \mathcal{V}_i are descendants of \mathcal{V}_{i_1} ; and 3) the parent node \mathcal{V}_{i_3} of node \mathcal{V}_{i_2} is either \mathcal{V}_{i_2-1} or an ancestor of \mathcal{V}_{i_2-1} , and \mathcal{V}_{i_3} is on the path from node \mathcal{V}_{i_1} to node \mathcal{V}_{i_2-1} .

A tree-to-tree distance algorithm for labelled trees in preorder notation using dynamic programming [Tai, 1979] is considered and modified to use the elementary context-preserving mutations $[M_I, M_D, M_S]$ from Section 2.3.1. Given two trees, the similarities between them are detected in three steps. The third step calculates the actual distance, while the first and second steps ensure that only distances achievable through legal context-preserving elementary mutations are counted.

The tree-to-tree distance algorithm using the elementary transformations from the CPM operator (Section 2.3.1) is given in four consecutive tables. Table 2.2b provides the function for computing the similarities between all subtrees from the first tree and all subtrees from the second. It uses the catalog updating function from the previous Table 2.2a. The function for counting the internode distances between arbitrary nodes in labelled preordered trees is shown in Table 2.2c. Finally, the three steps of the tree-to-tree distance algorithm are collected in Table 2.2d. The data arrays are common for all functions and are defined globally.

Table 2.2a. Function for filling the catalog of distances among all subtrees.

Catalog Filling	
<i>Algorithmic sequence</i>	
<pre> UpdateSD(r, u, i, q, z, j, x, y) { if (((r == u) and (u == i)) and ((q == z) and (z == j))) SD[r][u][i][q][z][j] = $\theta(i, j)$; else if ((((r == u) and (u == i)) or ((q < z) and (z == j))) { p2=parent(j, G') SD[r][u][i][q][z][j] = SD[r][u][i][q][p2][j - 1] + $\theta(0, j)$; } else if ((((r < u) and (u == i)) or ((q == z) and (z == j))) { p1=parent(i, G) SD[r][u][i][q][z][j] = SD[r][p1][i - 1][q][z][j] + $\theta(i, 0)$; } else SD[r][u][i][q][z][j] = Min(SD[r][x][i][q][z][j], SD[r][u][i][q][y][j], SD[r][u][x - 1][q][z][y - 1] + SD[x][x][i][y][y][j]); } } </pre>	

Table 2.2b. Function for computing the structural similarity between each subtree from the first tree and each subtree from the second tree.

Distance Cataloging	
<i>Algorithmic sequence</i>	
<pre> CatalogDistances() { for (i = 1; i < treeSize1; i++) for (j = 1; j < treeSize2; j++) { u = i; while (u > 0) { r = u; while (r > 0) { z=j; while (z > 0) { q=z; while (q > 0) { UpdateSD(r, u, i, q, z, j, x, y); q = parent(q, G'); } y=z; z = parent(z, G'); } r = parent(r, G); } x=u; u = parent(u, G); } } } </pre>	

Table 2.2c. Function for counting the distances between arbitrary nodes in labelled preordered trees, necessary for measuring tree-to-tree distance.

Internode Distance Counting

<i>Algorithmic sequence</i>	
<i>CountDistances()</i>	
{	
$ND[1][1] = 0;$	
for ($i = 2; i < \text{treeSize1}; i++$) $ND[i][1] = i;$	
for ($j = 2; j < \text{treeSize2}; j++$) $ND[1][j] = j;$	
for ($i = 2; i < \text{treeSize1}; i++$)	
for ($j = 2; j < \text{treeSize2}; j++$) {	
$ND[i][j] = \text{MAXINT};$	
$r = \text{parent}(i, G);$	
while($r > 0$) {	
$q = \text{parent}(j, G');$	
while($q > 0$) {	
$p1 = \text{parent}(i, G); p2 = \text{parent}(j, G');$	
$ND[i][j] = \text{Min}(ND[i][j],$	
$ND[r][q] + SD[r][p1][i-1][q][p2][j-1] - \theta(r, q));$	
$q = \text{parent}(q, G');$);}	
$r = \text{parent}(r, G);$ }	
$ND[i][j] = ND[i][j] + \theta(i, j);$ }	
}	

Table 2.2d. Algorithm for measuring the distance between labelled preordered trees using the elementary transformations from the CPM operator.

Tree-to-Tree Distance Algorithm

<i>Algorithmic sequence</i>	
1. Initialization	MAXN maximal tree size, MAXD maximal tree depth $SD[\text{MAXN}][\text{MAXN}][\text{MAXD}][\text{MAXD}][\text{MAXD}][\text{MAXD}];$ $D[\text{MAXN}][\text{MAXN}]; ND[\text{MAXN}][\text{MAXN}];$ θ unit distance function, G genetic program tree
2. Perform Cataloging	<i>CatalogDistances()</i> [Tables 2.2.a and 2.2.b]
3. Counting Distances	<i>CountDistances()</i> [Table 2.2.c]
4. Determine the minimal tree-to-tree distance	$D[1][1] = 0;$ for ($i = 2; i < \text{treeSize1}; i++$) $D[i][1] = D[i-1][1] + \theta(i, 0);$ for ($j = 2; j < \text{treeSize2}; j++$) $D[1][j] = D[1][j-1] + \theta(0, j);$ for ($i = 2; i < \text{treeSize1}; i++$) for ($j = 2; j < \text{treeSize2}; j++$) { $D[i][j] = \text{Min}(ND[i][j], D[i-1][j] + \theta(i, 0),$ $D[i][j-1] + \theta(0, j));$ } }

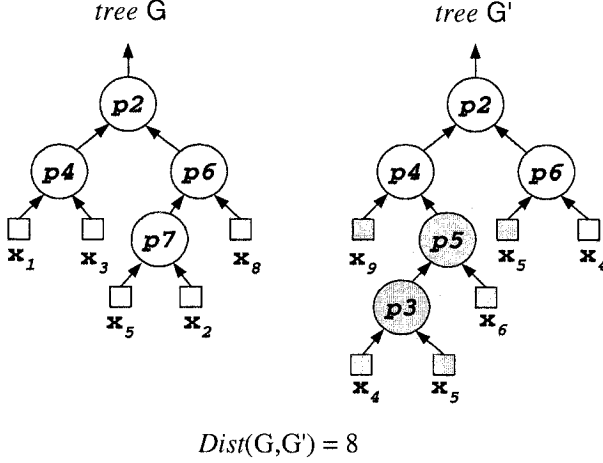


Figure 2.5. Tree-to-tree structural distance between two trees G and G' counted as the number of applications of the elementary operations $M \in [M_I, M_D, M_S]$.

This tree-to-tree distance algorithm is general and it serves for measuring the distance among trees with different branching factors. The algorithm implementation can be instantiated for binary trees in a straightforward manner. It should be noted that this tree-to-tree distance algorithm is computationally relatively expensive having complexity proportional to $\mathcal{O}(LL'S^2S'^2)$, where L and L' are the numbers of the nodes in the corresponding trees, and S and S' are their depths.

Figure 2.5 illustrates an example for the distance between two trees with respect to the mutations $M \in [M_I, M_D, M_S]$.

2.4 Random Tree Generation

The random tree generation algorithm impacts the evolutionary search performance. Its effect is through the ability to sample with equal probability each point from the search space. If the initial population has not been sampled well, the search may not converge to an optimal solution even if the IGP navigation is good. Since apriori information about the optima is usually not available, each search region should be sampled possibly with equal probability.

There are several methods available for random tree generation used in the GP community: naive methods [Koza, 1992], improved naive methods [Koza, 1992, Luke, 2000], and methods for construction of nearly uniformly distributed random trees such as uniform tree sampling [Salustowicz and Schmidhuber, 1997], stratified sampling [Iba, 1994] and compound derivations [Böhm and Geyer-Schulz, 1996].

The traditional method for random tree generation is the *Grow* algorithm [Koza, 1992]. It creates a tree-like genetic program by random generation of functionals and terminals until reaching the predefined depth. If a functional node is made, the algorithm continues recursively with attachment of random children nodes. The next node to install in the tree could be, with equal probability, a functional or a terminal from the node pool. This algorithm, however, does not allow to control the tree shape, therefore nonuniform, skewed distributions of trees result. One remedy to this unbalancing problem is to use the *Full* algorithm [Koza, 1992] which constructs complete trees up to a given depth.

Another improved method is the *ModifiedGrow* [Luke, 2000] which assigns user-defined probabilities to each functional node and terminal leaf so as to control their appearance in the trees. These probabilities are determined offline to speed up the tree construction. If the probability for choosing functionals over terminals is predetermined, random trees could be created around some desired average tree size. This algorithm, as well as the *Full* and *Grow*, does not necessarily produce truly random trees, rather it simply makes trees of different shape.

Better alternatives are provided by the methods that try to generate uniformly distributed random tree structures [Böhm and Geyer-Schulz, 1996, Iba, 1994, Salustowicz and Schmidhuber, 1997]. Such is the uniform sampling algorithm for initializing *probabilistic prototype trees* [Salustowicz and Schmidhuber, 1997]. A probabilistic prototype tree is a complete n -ary tree in which the nodes contain an initial probability threshold for selection of either a function or a terminal filling. A tree is generated by visiting every possible node from the complete tree. When the complete tree is traversed, a random number from the interval $[0; 1)$ is uniformly generated and compared with the threshold. If it exceeds the threshold, a terminal leaf is installed in the tree. Otherwise, a functional node with its children is inserted. The sum of probabilities for selecting the functions is equal to one.

An almost truly random tree generation algorithm can be implemented according to the bijection method [Iba, 1994]. This sophisticated algorithm produces random trees based on uniformly sampled words of symbols from a simple grammar of only two letters. This is a strategy for production of random words with n letters of the one kind and $n - 1$ letters of the other kind. Next, the algorithm makes random permutations of the words, and builds corresponding trees of n nodes using a stack machine when reading the words. The generation of distinct tree structures is restricted by the desired node arities. The number of possible distinct trees and the construction of the solution are proportional to the Catalan number for trees with n nodes.

Exact uniform generation of complete trees can be accomplished using context-free grammars [Böhm and Geyer-Schulz, 1996]. This is a method for uniform sampling of populations of distinct words written in terms of a predefined context-free language. Distinct trees leading to different derivations of the same word are produced with each invocation of this algorithm. This algorithm recursively grows a complete derivation tree by randomly choosing the next context-free grammar rule for expanding the tree. The limit on the number of derivation steps constraints the tree depth. This tree generation algorithm can be applied to IGP after modifying the grammar rules to build functional models.

2.5 Basic IGP Framework

The IGP paradigm can be used for the automatic programming of polynomials. It provides a problem independent framework for discovering the polynomial structure in the sense of shape and size, as well as the weights. The IGP learning cycle involves five substeps: 1) ranking of the individuals according to their fitness; 2) selection of some elite individuals to mate and produce offspring; 3) processing of the chosen parent individuals by the crossover and mutation operators; 4) evaluation of the fitnesses of the offspring; and 5) replacement of predetermined individuals in the population by the newly born offspring. Table 2.3 presents the basic IGP algorithmic framework.

The formalization of the basic framework, which can be used for implementing an IGP system, requires some preliminary definitions. The IGP mechanisms operate at the genotype level; that is, they manipulate linearly implemented genetic program trees g . The basic control loop breeds a population \mathcal{P} of genetic programs g during a number of cycles τ , called generations. Let n denote the size of the population vector; that is, the population includes $g_i, 1 \leq i \leq n$ individuals. Each individual g is restricted by a predefined tree depth S and size L in order to limit the search space to within reasonable bounds. The initial population $\mathcal{P}(0)$ is randomly created (Section 2.4).

The function *Evaluate* estimates the fitness of the genetic programs using the fitness function f to map genotypes $g \in \Gamma$ into real values $f: \Gamma \rightarrow R$. The fitness function f takes a genetic program tree g , decodes a phenotypic PNN model from it, and measures its accuracy with respect to the given data. All the fitnesses of the genetic programs from the population are kept in an array of fitnesses F of size n . The selection mechanism *Select*: $\Gamma^n \rightarrow \Gamma^{n/2}$ operates according to a predefined scheme for randomly picking $n/2$ elite individuals which are going to be transformed by crossover and/or mutation.

The recombination function *CrossTrees*: $\Gamma^{n/4} \times R \rightarrow \Gamma^{n/4}$ takes the half $n/4$ from the selected $n/2$ elite genetic programs, and produces the same number of offspring using size-biased crossover using parameter κ (Section 2.3). The mutation function *MutateTrees*: $\Gamma \times R \rightarrow \Gamma$ processes half $n/4$ from the selected $n/2$ elite genetic programs, using size-biased context-preserving mutation using parameter μ (Section 2.3).

The resulted offspring are evaluated, and replace inferior individuals in the population *Replace*: $\Gamma^{n/2} \times \Gamma^{n/2} \times N \rightarrow \Gamma^n$. The steady-state reproduction scheme is used to replace the genetic programs having the worst fitness with the offspring so as to maintain a proper balance of promising individuals (Section 5.1). Next, all the individuals in the updated population are ordered according to their fitnesses.

Table 2.3. Basic framework for IGP.

Inductive Genetic Programming	
<i>step</i>	<i>Algorithmic sequence</i>
1. Initialization	<p>Let the generation index be $\tau = 0$, and the pop size be n</p> <p>Let the initial population be: $\mathcal{P}(\tau) = [g_1(\tau), g_2(\tau), \dots, g_n(\tau)]$ where g_i, $1 \leq i \leq n$, are genetic programs of depth up to S.</p> <p>Let μ be a mutation parameter, κ be a crossover parameter.</p> <p>Create a random initial population: $\mathcal{P}(\tau) = \text{RandomTrees}(n)$, such that $\forall g, \text{Depth}(g) < S$.</p> <p>Evaluate the fitnesses of the individuals: $F(\tau) = \text{Evaluate}(\mathcal{P}(\tau), \lambda)$ and order the population according to $F(\tau)$.</p>
2. Evolutionary Learning	<p>a) Randomly select $n/2$ elite parents from $\mathcal{P}(\tau)$ $\mathcal{P}'(\tau) = \text{Select}(\mathcal{P}(\tau), F(\tau), n/2)$.</p> <p>b) Perform recombination of $\mathcal{P}'(\tau)$ to produce $n/4$ offspring $\mathcal{P}''(\tau) = \text{CrossTrees}(\mathcal{P}'(\tau), \kappa)$.</p> <p>c) Perform mutation of $\mathcal{P}'(\tau)$ to produce $n/4$ offspring $\mathcal{P}''(\tau) = \text{MutateTrees}(\mathcal{P}'(\tau), \mu)$.</p> <p>d) Compute the offspring fitnesses $F''(\tau) = \text{Evaluate}(\mathcal{P}''(\tau), \lambda)$.</p> <p>e) Exchange the worst $n/2$ from $\mathcal{P}(\tau)$ with offspring $\mathcal{P}''(\tau)$ $\mathcal{P}(\tau + 1) = \text{Replace}(\mathcal{P}(\tau), \mathcal{P}''(\tau), n/2)$.</p> <p>f) Rank the population according to $F(\tau + 1)$ $g_0(\tau + 1) \leq g_1(\tau + 1) \leq \dots \leq g_n(\tau + 1)$.</p> <p>g) Repeat the Evolutionary Learning (<i>step 2</i>) with another cycle $\tau = \tau + 1$ until the termination condition is satisfied.</p>

2.6 IGP Convergence Characteristics

Theoretical studies of the convergence characteristics of GP have been performed in two main directions: to formulate a schema theorem for GP, and to develop a Markov model for GP. The research in these directions examines the evolutionary computation process from different points of view and reveals different factors that affect the search convergence. The theory given in this subsection can help to understand how the system propagates useful subsolutions of the task. A schema theorem valid for IGP is discussed next and an original Markov model for simplified IGP with fitness proportionate selection and context-preserving mutation is proposed for theoretical convergence analysis.

2.6.1 Schema Theorem of IGP

A formal explanation of the GP search performance can be made using the schema convergence theorem [Poli and Langdon, 1998, Rosca and Ballard, 1999, Langdon and Poli, 2002]. The schema theorem establishes a relationship between the individuals in the population and the mechanisms of the GP system. This theorem quantifies the expected behavior of similar individuals that are matched by a common pattern called schemata. The schema theorem describes the rates with which a schemata representing a region on the landscape grows and shrinks, thus allowing us to reason how the search progresses. During evolutionary search, schema of individuals with fitness greater than the average population fitness usually produce more offspring.

Individuals with close structural characteristics are summarized by a schemata. A schemata is a similarity pattern modelling a set of trees that are identical at certain positions; that is, they have coinciding functional and terminal nodes. Imposing a schemata on trees means that it can be taken to generate all similar trees with the property of having the common nodes. The common nodes are fixed positions, while the remaining positions can match any subtree, and are called wildcards.

Most convenient for describing the IGP performance is the rooted-tree schema theorem [Rosca and Ballard, 1999]. This theorem provides an expression for predicting the frequencies of individuals matched by rooted-tree schemata in the population depending on how their fitnesses relate to the average population fitness and depending on the effects of the selection, crossover, and mutation operators. A rooted-tree schemata is a tree fragment of certain order with a common root. The order of the schemata is the number of all its functional and terminal nodes without the wildcard symbols. Due to the fixed root, a schemata matches different trees having a similar shape. The different rooted-trees belong to

disjoint subsets of the search landscape; therefore a rooted-tree schemata covers a concrete search region. When the search proceeds, the schema in the population shrink and grow, thus indicating how the evolutionary search effort is distributed on the search landscape.

Let the population contain schemata \mathcal{H} and let there be $m(\mathcal{H}, \tau)$ instances of it at generation τ . Suppose that the average population fitness is $\bar{f}(\tau)$, and the average fitness of all schemata instances \mathcal{H} at this moment is $\bar{f}_{\mathcal{H}}(\tau)$. The *rooted-tree schema convergence theorem* for traditional GP systems states that [Rosca and Ballard, 1999]:

$$m(\mathcal{H}, \tau + 1) \geq m(\mathcal{H}, \tau) \frac{\bar{f}_{\mathcal{H}}(\tau)}{\bar{f}(\tau)} \left[1 - (p_m + p_c) \frac{\sum_{i \in \mathcal{H}} f_i(\tau)/l_i}{\sum_{i \in \mathcal{H}} f_i(\tau)/O(\mathcal{H})} \right] \quad (2.6)$$

where i enumerates the instances of the schema \mathcal{H} , l_i is the size of instance i , $O(\mathcal{H})$ is the order of the schema, p_m is the mutation probability, and p_c is the crossover probability.

The above theorem shows that the survival of individuals depends not only on their fitness but also on their size. Search strategies can be developed using the tree size as a parameter in the following way: 1) size-biased mutation and crossover operators for tree size adaptation can be made (Section 2.3); and 2) size-biased fitness functions for search guidance and regulation of the tree complexities in the population through selection can be designed (Section 4.1.1). These two strategies have the potential to combat the tree growth problem and contribute to sustaining progressive search.

Although the formalism of the schema theorem gives some clues of how to improve the behavior of the evolutionary system, it has been widely criticized because it can hardly be used for online performance investigation [Vose, 1999]. The schema are difficult to detect and can hardly be identified while the GP system operates in real-time, and so in practice we cannot collect average statistics for the expected number of similar individuals in the population.

2.6.2 Markov Model of IGP

The Markov model of IGP provides a formalization of the probability distribution of chains from populations. The distribution of the populations evolved by the system during the generations allows us to analyze theoretically whether IGP can be expected to converge to a uniform distribution or not. If the population distribution reaches a uniform distribution within a finite number of steps, this means that the search will terminate. Having theoretical results from such analysis may be useful to realize how the genetic operators impact the evolutionary search.

IGP carries a stochastic process of continuous production of successive populations. It can be envisioned that each successive population in this evolutionary process depends only on the last population, and it does not depend on the process history. This is why the evolved populations may be considered a Markov chain. This reasoning allows us to develop a Markov model of the population-based IGP search. For simplicity, it is assumed here that the random effects are caused by selection and mutation operators which act on a population of a fixed size.

The IGP behavior can be studied following the related Markov theory of genetic algorithms [Davis and Principe, 1993], bearing in mind that the genetic programs have variable size. Assume that the IGP genotypes are represented using a language with alphabet \mathcal{A} . A genotype written in terms of this language of size no greater than L is a set of such symbols $\mathcal{G} = \{\mathcal{A} + \varepsilon\}^L$, where ε is the empty symbol. Markov chain analysis helps us to find the distribution of populations from such genotypes. Let the genotypes be specified by $g \in \mathcal{G}$ and $h \in \mathcal{G}$. Suppose that two successive populations are denoted by π and ν , and note that they have equal size $|\pi| = |\nu| = n$. The number of times that a particular genotype g appears in the population may be specified by $\pi(g)$. The population is then $\pi = [\pi(g_1), \pi(g_2), \dots]$ as it contains genotypes g_1, g_2, \dots some of which are repeated and we have $\sum_{g \in \mathcal{G}} \pi(g) = n$. Clearly π is a distribution of n genotypes over $|\{\mathcal{A}\}^L|$ bins, where the symbol $|\cdot|$ denotes the cardinality of a set. The genotype space that the IGP system can explore is a subset of the whole problem solution space $\{\pi\}$, which is the set of all possible populations. Since each population is a distribution of genotypes, it can be assumed that the genetic programming system produces a Markov chain of genotype distributions.

When the IGP manipulates the population using only the selection operator, the *probability for selection* $\text{Pr}_l(g|\pi)$ of an individual g from population π is:

$$\text{Pr}_l(g|\pi) = \frac{\pi(g)f(g)}{\sum_{h \in \mathcal{G}} \pi(h)f(h)} \quad (2.7)$$

where h enumerates all individuals in the population having their particular frequencies $\pi(h)$, $f(g)$ is the fitness value of individual g and f is usually a real number.

The *probability* $\text{Pr}_l(\nu|\pi)$ for producing a population ν from population π after n independent samplings with returning of genotypes is:

$$\text{Pr}_l(\nu|\pi) = \frac{n!}{\prod_{g \in \mathcal{G}} \nu(g)!} \prod_{g \in \mathcal{G}} \text{Pr}_l(g|\pi)^{\nu(g)} \quad (2.8)$$

where n is the population size, and $\nu(g)$ is the number of occurrences of genotype g in population ν (note that $\sum_{g \in \mathcal{G}} \nu(g) = N$).

These two probabilities allow us to obtain a boundary for the convergence of the population due to the forces exerted by the selection operator toward states of uniform populations. The expected number of transitions $E(\nu|\pi)$ from population to population until reaching a uniform population, is given by the inequality [Davis and Principe, 1993]:

$$E(\nu|\pi) \leq \left(\frac{nf_{\max}}{f_{\min}} \right)^{2n} < \infty \quad (2.9)$$

where f_{\max} and f_{\min} are the possible maximal and minimal fitness values.

Consider the case when IGP manipulates the population using two operators: selection and mutation. Since the stochastic effects from the selection operator have been investigated, it remains to study the effects from the context-preserving mutation. The *probability* $\Pr_m(g|h)$ to obtain by mutation individual g from individual h is:

$$\Pr_m(g|h) = \frac{1}{|\mathcal{A}| + 1} \binom{L}{k} p_m^k (1 - p_m)^{L-k} \quad (2.10)$$

where $|\mathcal{A}| + 1$ is the alphabet size plus one empty symbol ϵ for deletion. The alphabet is $\mathcal{A} = F \cup T$, L is the size of genotype h , k is the number of modified alleles by the mutation, and p_m is the mutation probability.

The emergence of a genotype is a two-stage process: selection of a candidate from the population, and mutating it according to the mutation probability. Thus, the *probability* $\Pr_{lm}(g|\pi)$ for obtaining by selection and mutation a genotype g from population π is:

$$\Pr_{lm}(g|\pi) = \sum_{h \in \pi} \Pr_l(h|\pi) \Pr_m(g|h) \quad (2.11)$$

Then, the *probability* $\Pr_{lm}(\nu|\pi)$ for producing a population ν from population π by the combined use of selection and mutation is:

$$\Pr_{lm}(\nu|\pi) = \frac{n!}{\prod_{g \in \mathcal{G}} \nu(g)!} \prod_{g \in \mathcal{G}} \Pr_{lm}(g|\pi)^{\nu(g)} \quad (2.12)$$

where $\nu(g)$ is the frequency of genotype g in the next population.

It is proven that for such combined use of selection and mutation, the chain of population distributions cannot converge to a homogeneous state because the following relationship holds [Davis and Principe, 1993]:

$$\frac{n!}{\prod_{g \in \mathcal{G}} \nu(g)!} \left(\frac{p_m}{(1 - p_m)^2} \right)^{nL} = \Pr_{lm}(\nu|\pi) \quad (2.13)$$

which is an indication for a stationary distribution.

The theoretical results from this Markov chain analysis of the population characteristics generated by an IGP system show that selection and mutation are not sufficient to organize successful evolutionary search; that is, there is a need for crossover as well. From another point of view, the Markov chain analysis demonstrated that the IGP performance depends on the mutation. Although some GP systems do not use it, one can realize that the mutation operator is not only useful but it is a necessary mean to sustain diversity in the population. The maintenance of enough diversity is a condition for achieving sustained exploration of the search space, and thus progressive search.

2.7 Chapter Summary

This chapter presented the basic structure of tree-like PNN models suitable for IGP. The PNN trees were related to other polynomial networks to emphasize the key differences between them, and to strengthen the confidence that tree-structured PNN are more flexible for doing evolutionary search by the mechanisms of IGP. The polynomial network models are allocated in binary tree structures for reducing the dimensionality of the search space and for making the network topology search process more efficient. The trees are implemented as linearized arrays to achieve fast computational speed.

Loosely simulating principles from natural evolution led to the development of the basic mechanism for genetic programming with PNN. A context-preserving mutation operator and a crossover operator were made. They can be applied to transform PNN structures with size-biasing in order to prevent degeneration of the search performance toward very small or very large trees. With the IGP, other mutation and crossover operators can be designed if it is necessary to improve the evolutionary search. The IGP framework provides a basic algorithm which can also be enhanced with novel features.

The presented brief theoretical analysis of the convergence behavior of IGP explained which factors affect the essential behavior of the system and provided clues as to what can be done in order to attain sustained evolution. It has been explained theoretically that: 1) it is beneficial to use size-biasing of the genetic learning operators and the fitness function because they influence the size of the offspring and avoid generating large complex trees which overfit the data and are harder to manage; and 2) it is beneficial to employ a mutation operator, in addition to the traditionally used crossover and selection in GP, because it helps to maintain high population diversity.

Adaptive Learning of Polynomial Networks
Genetic Programming, Backpropagation and Bayesian
Methods

Nikolaev, N.; Iba, H.

2006, XIV, 316 p., Hardcover

ISBN: 978-0-387-31239-2