Chapter 5
Genetic Algorithms and Genetic Programming

5.1 Introduction

In this chapter we introduce powerful optimization techniques based on evolutionary computation. The techniques mimic natural selection and the way genetics works. Genetic algorithms were first proposed by J. Holland in the 1960s. Today, they are mainly used as a search technique to find approximate solutions to different kinds of problems. In intelligent control (IC) they are mostly used as an optimization technique to find minimums or maximums of complex equations, or quasi-optimal solutions in short periods of time.

N. Cramer later proposed genetic programming in 1985, which is another kind of evolutionary computation algorithm with string bases in genetic algorithms (GA). The difference basically is that in GA strings of bits representing chromosomes are evolved, whereas in genetic programming the whole structure of a computer program is evolved by the algorithm. Due to this structure, genetic programming can manage problems that are harder to manipulate by GAs. Genetic programming has being used in IC optimize the sets of rules on fuzzy and neuro-fuzzy controllers.

5.1.1 Evolutionary Computation

Evolutionary computation represents a powerful search and optimization paradigm. The metaphor underlying evolutionary computation is a biological one, that of natural selection and genetics. A large variety of evolutionary computational models have been proposed and studied. These models are usually referred to as evolutionary algorithms. Their main characteristic is the intensive use of randomness and genetic-inspired operations to evolve a set of solutions.

Evolutionary algorithms involve selection, recombination, random variation and competition of the individuals in a population of adequately represented potential solutions. These candidate solutions to a certain problem are referred to as chromosomes or individuals. Several kinds of representations exist such as bit string,

© Springer 2010

real-component vectors, pairs of real-component vectors, matrices, trees, tree-like hierarchies, parse trees, general graphs, and permutations.

In the 1950s and 1960s several computer scientists started to study evolutionary systems with the idea that evolution could be applied to solve engineering problems. The idea in all the systems was to evolve a population of candidates to solve problems, using operators inspired by natural genetic variations and natural selection.

In the 1960s, I. Rechenberg introduced evolution strategies that he used to optimize real-valued parameters for several devices. This idea was further developed by H.P. Schwefel in the 1970s. L. Fogel, A. Owens and M. Walsh in 1966 developed *evolutionary programming*, a technique in which the functions to be optimized are represented as a finite-state machine, which are evolved by randomly mutating their state-transition diagrams and selecting the fittest. Evolutionary programming, evolution strategies and GAs form the backbone of the field of evolutionary computation.

GAs were invented by J. Holland in the 1960s at the University of Michigan. His original intention was to understand the principles of adaptive systems. The goal was not to design algorithms to solve specific problems, but rather to formally study the phenomenon of adaptation as it occurs in nature and to develop ways in which the mechanisms of natural adaptation might be ported to computer systems. In 1975 he presented GAs as an abstraction of biological evolution in the book *Adaptation in Natural and Artificial Systems*.

Simple biological models based on the notion of survival of the best or fittest were considered to design robust adaptive systems. Holland's method evolves a population of candidate solutions. The chromosomes are binary strings and the search operations are typically crossover, mutation, and (very seldom) inversion. Chromosomes are evaluated by using a fitness function.

In recent years there has been an increase in interaction among researchers studying different methods and the boundaries between them have broken down to some extent. Today the term GA may be very far from Holland's original concept.

5.2 Industrial Applications

GAs have been used to optimize several industrial processes and applications. F. Wang and others designed and optimized the power stage of an industrial motor drive using GAs at the Virginia Polytechnic Institute and State University at Virginia in 2006 [1]. They analyzed the major blocks of the power electronics that drive an industrial motor and created an optimization program that uses a GA engine. This can be used as verification and practicing tools for engineers.

D.-H. Cho presented a paper in 1999 [2] that used a niching GA to design an induction motor for electric vehicles. Sometimes a motor created to be of the highest efficiency will perform at a lower level because there are several factors that were not considered when it was designed, like ease of manufacture, maintenance, reliability, among others. Cho managed to find an alternative method to optimize the design of induction motors.

GAs have also been used to create schedules in semiconductor manufacturing systems. S. Cavalieri and others [3] proposed a method to increase the efficiency of dispatching, which is incredibly complex. This technique was applied to a semiconductor manufacture plant. The algorithm guarantees that the solution is obtained in a time that is compatible with on-line scheduling. They claim to have increased the efficiency by 70%.

More recently V. Colla and his team presented a paper [4] where they compare traditional approaches, and GAs are used to optimize the parameters of the models. These models are often designed from theoretical consideration and later adapted to fit experimental data collected from the real application. From the results presented, the GA clearly outperforms the other optimization methods and fits better with the complexity of the model. Moreover, it provides more flexibility, as it does not require the computation of many quantities of the model.

5.3 Biological Terminology

All living organisms consist of cells that contain the same set of one or more *chromosomes* serving as a blueprint. Chromosomes can be divided into *genes*, which are functional blocks of DNA. The different options for genes are *alleles*. Each gene is located at a particular *locus* (position) on the chromosome. Multiple chromosomes and or the complete collection of genetic material are called the organism's *genome*. A *genotype* refers to the particular set of genes contained in a genome.

In GAs a *chromosome* refers to an individual in the population, which is often encoded as a string or an array of bits. Most applications of GAs employ haploid individuals, which are single-chromosome individuals.

5.3.1 Search Spaces and Fitness

The term "search space" refers to some collection of candidates to a problem and some notion of "distance" between candidate solutions. GAs assume that the best candidates from different regions of the search space can be combined via crossover, to produce high-quality offspring of the parents. "Fitness landscape" is another important concept; evolution causes populations to move along landscapes in particular ways and adaptation can be seen as the movement toward local peaks.

5.3.2 Encoding and Decoding

In a typical application of GAs the genetic characteristics are encoded into bits of strings. The encoding is done to keep those characteristics in the environment. If we want to optimize the function $f(x) = x^2$ with $0 \leq x < 32$, the parameter of the search space is x and is called the phenotype in an evolutionary algorithm. In

Table 5.1 Chromosome encoded information

Decimal number	Binary encoded
5	00101
20	10100
7	01011

GAs the phenotypes are usually converted to genotypes with a coding procedure. By knowing the range of x we can represent it with a suitable binary string. The chromosome should contain information about the solution, also known as encoding (Table 5.1).

Although each bit in the chromosome can represent a characteristic in the solution here we are only representing the numbers in a binary way. There are several types of encoding, which depend heavily on the problem, for example, permutation encoding can be used in ordering problems, whereas floating-point encoding is very useful for numeric optimization.

5.4 Genetic Algorithm Stages

There are different forms of GAs, however it can be said that most methods labeled as GAs have at least the following common elements: population of chromosomes, selection, crossover and mutation (Fig. 5.1). Another element rarely used called inversion is only vaguely used in newer methods. A common application of a GA is the optimization of functions, where the goal is to find the global maximum or minimum.

A GA [5] can be divided into four main stages:

- *Initialization*. The initialization of the necessary elements to start the algorithm.
- *Selection*. This operation selects chromosomes in the population for reproduction by means of evaluating them in the fitness function. The fitter the chromosome, the more times it will be selected.

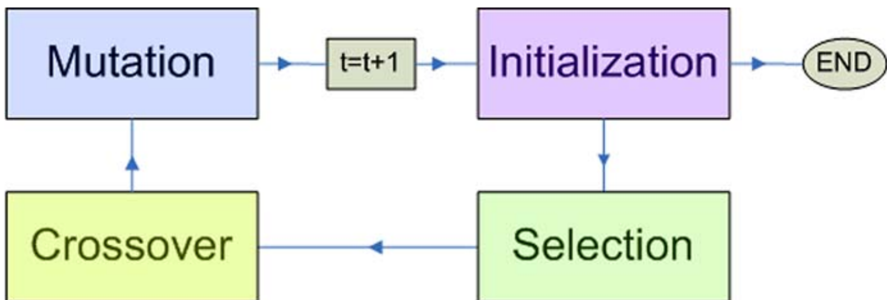

Fig. 5.1 GA main stages

- *Crossover*. Two individuals are selected and then a random point is selected and the parents are cut, then their tails are crossed. Take as an example 100110 and 111001: the 3 position from left to right is selected, they are crossed, and the offspring is 100001, 111110.
- *Mutation*. A gene, usually represented by a bit is randomly complemented in a chromosome, the possibility of this happening is very low because the population can fall into chaotic disorder.

These stages will be explained in more detail in the following sections.

5.4.1 Initialization

In this stage (shown in Fig. 5.2) the initial individuals are generated, and the constants and functions are also initiated, as shown in Table 5.2.

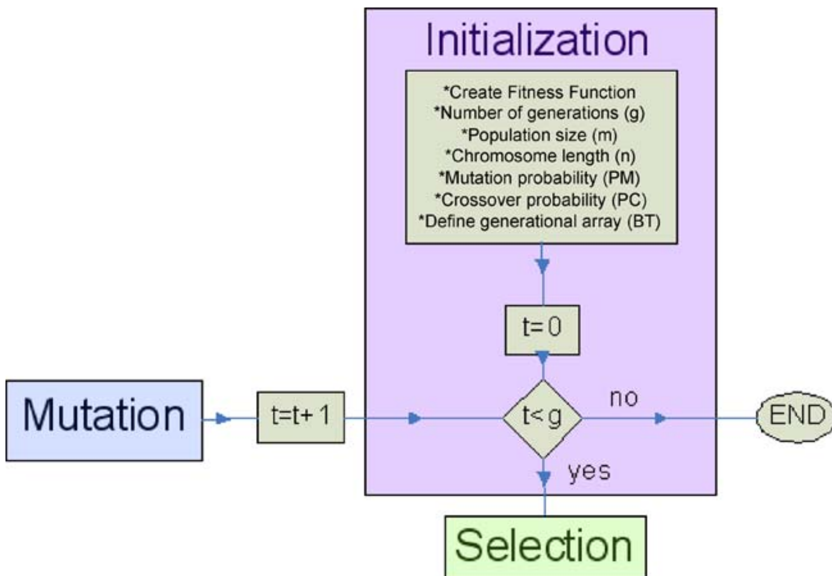

Fig. 5.2 GA initialization stage

Table 5.2 GA initialization parameters

Parameter	Description
g	The number of generations of the GA.
m	Size of the population.
n	The length of the string that represents each individual: $s = \{0, 1\}^n$. The strings are binary and have a constant length.
PC	The probability of crossing of 2 individuals.
PM	The probability of mutation of every gen.

5.4.2 Selection

A careful selection of the individuals must be performed because the domination of a single high-fit individual may sometimes mislead the search process. There are several methods that will help avoid this problem, where individual effectiveness plays a very negligible role in selection. There are several selection methods like scaling transformation and rank-based, tournaments, and probabilistic procedures.

By scaling we mean that we can modify the fitness function values as required to avoid the problems connected with proportional selection. It may be static or dynamic; in the latter, the scaling mechanism is reconsidered for each generation.

Rank-based selection mechanisms are focused on the rank ordering of the fitness of the individuals. The individuals in a population of n size are ranked according to their suitability for search purposes. Ranking selection is natural for those situations in which it is easier to assign subjective scores to the problem solutions rather than to specify an exact objective function.

Tournament selection was proposed by Goldberg in 1991, and is a very popular ranking method of performing selection. It implies that two or more individuals

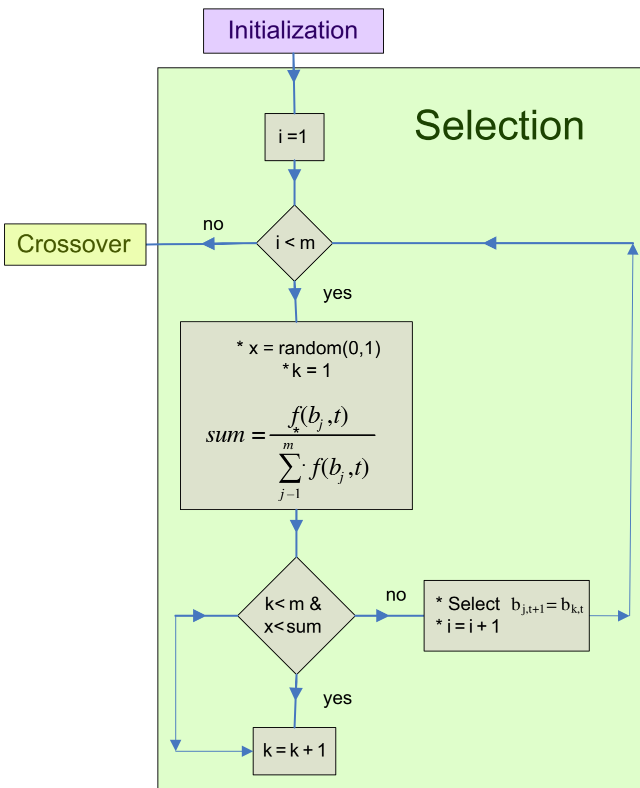

Fig. 5.3 GA selection stage

compete for selection. The tournament may be done with or without reinsertion of competing individuals into the original population.

Finally the roulette-wheel selection process is another popular method used as selection stage, where the fittest individuals have a higher probability to be selected. In this method individuals are assigned a probability to be selected, then a random number is calculated and the probability of individuals is accumulated. Once that value of the random number is reached, the individual presently used is the one that is selected.

However, in order to perform selection it is necessary to introduce a measure of the performance of individuals. By selection we aim to maximize the performance of individuals. Figure 5.3 shows the diagram of this stage.

Fitness Function

As we already mentioned selection methods need a tool to measure the performance of individuals. The search must concentrate on regions of the search space where the best individuals are located. This concentration accomplishes the exploitation of the best solutions already found, which is exactly the purpose of selection. For selection purposes a performance value is associated with each individual in the current population, and represents the *fitness* of the function.

A *fitness function* is usually used to measure explicitly the performance of chromosomes, although in some cases the fitness can be measured only in an implicit way, using information about the performance of systems. Chromosomes in a GA take the form of bit strings; they can be seen as points in the search space. This population is processed and updated by the GA, which is mainly driven by a fitness function, a mathematical function, a problem or in general, a certain task where the population has to be evaluated.

5.4.3 Crossover

In order to increase population diversity, other operators are used such as the crossover operation. By perturbing and recombining the existent individuals, the search operators allow the search process to explore the neighboring regions or to reach further promising regions.

Crossover operations achieve the recombination of the selected individuals by combining segments belonging to chromosomes corresponding to parents. Figure 5.4 shows a diagram of the crossover stage, which creates an information exchange between the parent chromosomes. Thus the descendent obtained will possess features from both parents.

The role of recombination is to act as an impetus to the search progress and to ensure the exploration of the search space. Various crossover operations have been proposed, and here we will explain the most employed variant used in binary encoded frameworks: the one-point crossover. The crossover probability (CP) is

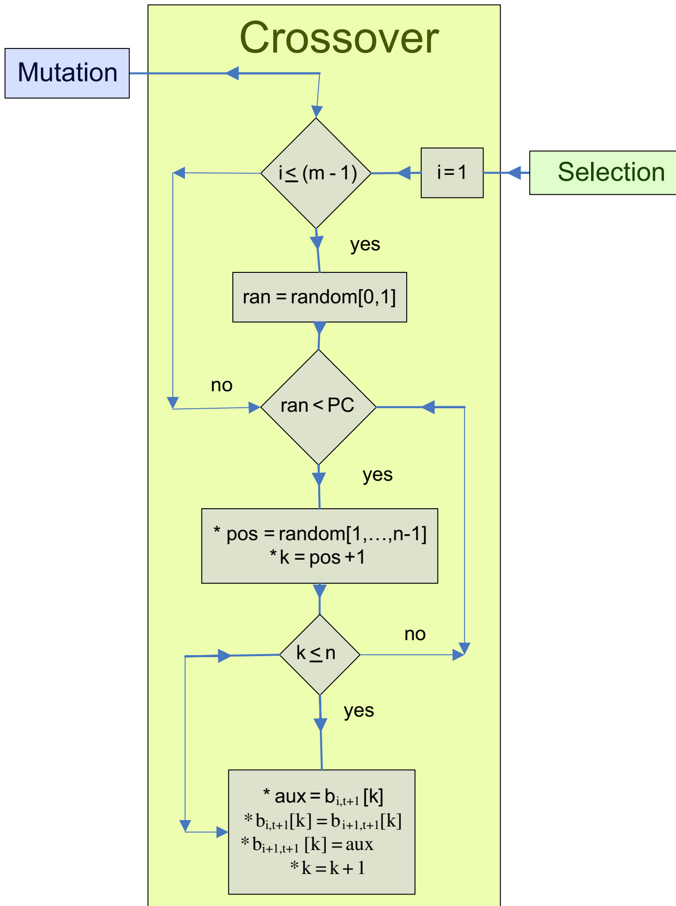

Fig. 5.4 GA crossover stage

compared with a random number between (0,1) and with this, it is determined if is going to be crossover or not. When a crossover is made, the positions in which the parents are going to be cut in a random position are then interchanged.

5.4.4 Mutation

In classical genetics, mutation is identified by an altered phenotype, and in molecular genetics mutation refers to any alternation of a segment of DNA. Spontaneous mutagenesis is normally not adaptive, and mutations normally do not provide a selective advantage. Changes may destroy the genome structure, where other changes tend to create and integrate new functions.

Changes representing a selective disadvantage occur considerably more often and can affect life processes in various degrees. The extreme situation leads to lethality. Often the alteration of the chromosome remains without immediate consequences on life processes. This kind of mutations is called *natural* or *silent*. Neutral mutations however may play an evolutionary role.

Within the framework of binary encoding mutation is considered the second most important genetic operator. The effect of this operator is to change a single position (gene) within a chromosome. If it were not for mutation, other individuals could not

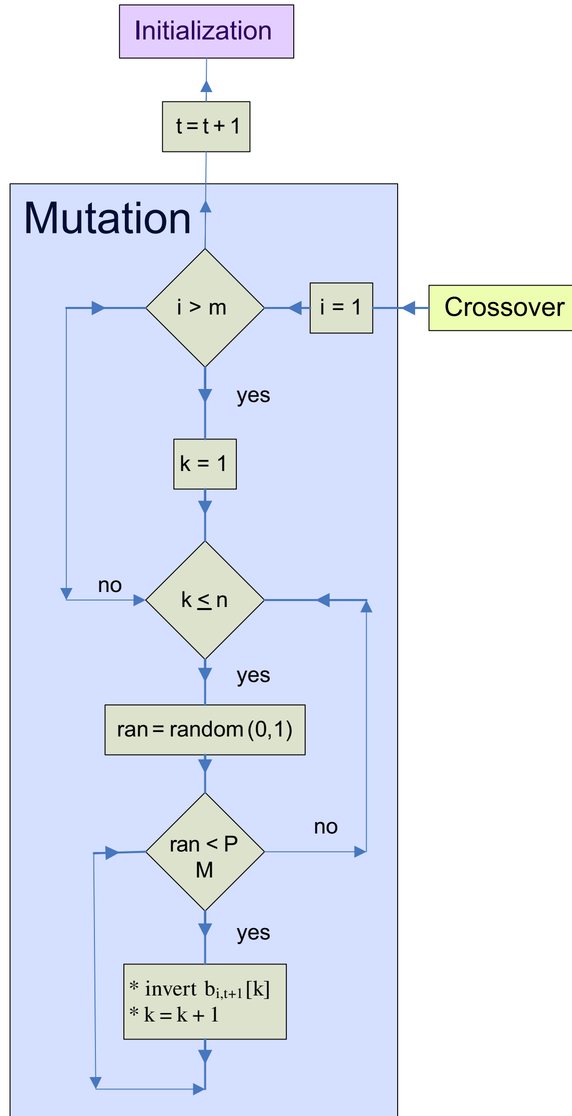

Fig. 5.5 Mutation stage

be generated through other mechanisms, which are then introduced to the population. The mutation operator assures that a full range of individuals is available to the search.

One of the simplest executions of mutation is when the mutation probability (MP) is compared with a random number between (0,1). If it is going to be a mutation, a randomly chosen bit of the string is inverted. A diagram showing this stage is in Fig. 5.5.

Example 5.1. This is an example of a GA using the Intelligent Control Toolkit for LabVIEW (ICTL). A base algorithm created for searching the maximum and minimum in the $f(x) = x^2$ function will be explained. This program is included as

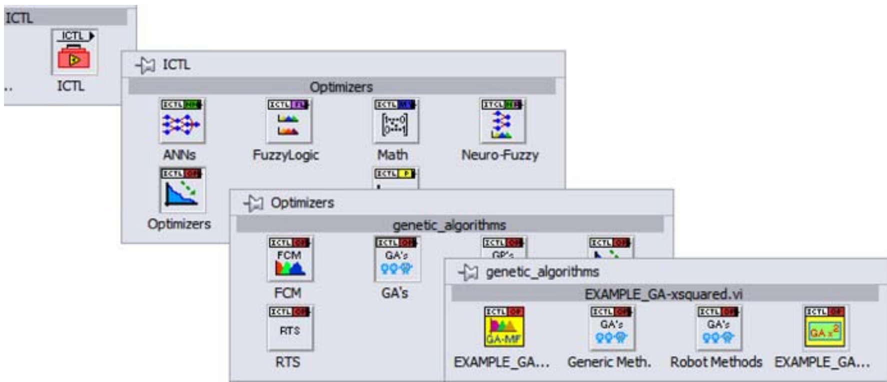

Fig. 5.6 Localization of GA methods on the toolkit

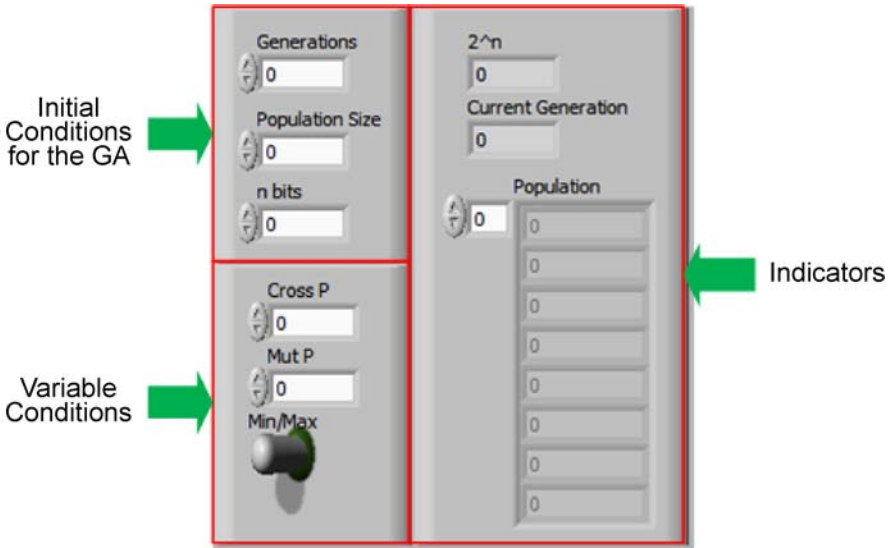

Fig. 5.7 GA-X squared front panel

Table 5.3 Initial conditions and variables for the x^2 example

Variable	Description
Generations	The number of repetitions of the algorithm.
Population Size	The number of individuals per generation, for example 6.
n bits	In this case it is how many bits will be in the binary string that represents the individuals, for example, if the individuals will be numbers from 0 to 31, then $n = 5$, because $25 - 1 = 31$.
Cross P	The crossover probability if $CP = 1$ every individual selected will get combined with other selected individuals.
Mut P	Is the mutation probability and should be small; 0.001 is a good value.
Min/Max	Allows us to select if we want to minimize or maximize the function.

a toolkit example but we will explain the development in detail. Figure 5.6 shows where the GA methods can be found in the ICTL. First we build the front panel where we will have the controls and indicators of all the variables.

Figure 5.7 shows an image of how to build the front panel. The initial conditions and variables that we will include are shown in Table 5.3. Now we need to start building the code of our program. First, we need to generate an initial population before we start the algorithm, therefore we create a series of random numbers, depending on the initial conditions. The code is shown in Fig. 5.8.

Basically, a series of random numbers are created, where the top is given by the number of bits used, and later they are transformed into chains of bits. We also need a decoding function and a fitness function. The decoding will convert bits to numbers again and the fitness function will raise those numbers to the second power, as shown in Fig. 5.9.

Now we need to perform selection, crossover, and mutation, the basic operations of a GA. As shown in Fig. 5.10, we find the GA methods in the path Optimizers ≫ GAs Palette ≫ Generic Methods. Our decoded and fitness-evaluated individuals will be fed to the selection method, later we will start a loop where the selected

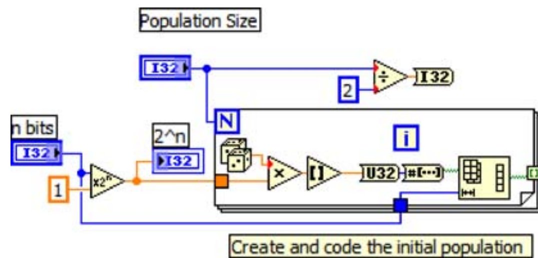

Fig. 5.8 Creation and coding of initial population

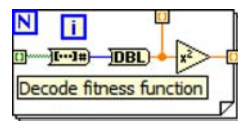

Fig. 5.9 Decoding and fitness function

Fig. 5.10 GA methods included in the ICTL

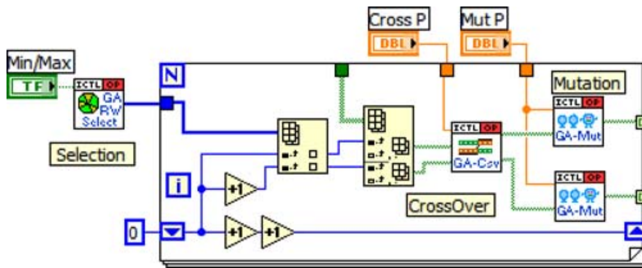

Fig. 5.11 Selection, crossover, and mutation stages code

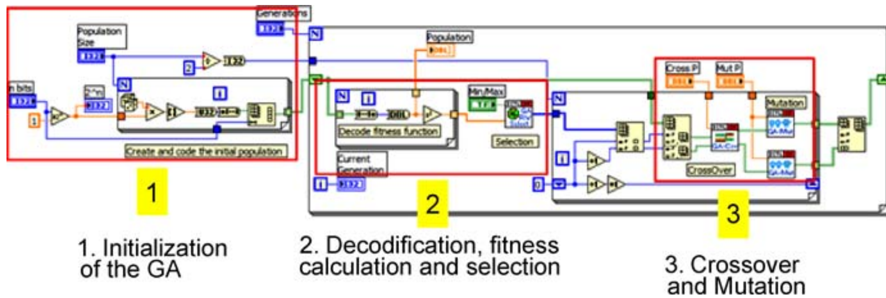

Fig. 5.12 Complete block diagram of the GA-X squared example

individuals will be crossed over and mutated (Fig. 5.11). From this we also see that we can perform operations in parallel, e. g., the mutation, thus allowing us to increase the operation time of the program.

The complete code is found in Fig. 5.12. Now we can set the initial conditions and run our program and see that after 100 generations the maximum of the function $f(x) = x^2$ for $0 \leq x \leq 31$ has been found; Fig. 5.13 shows some results. With this simple but powerful example we can apply GAs to other applications; the key will be in the coding and fitness function. □

5.5 Genetic Algorithms and Traditional Search Methods

There are at least three meanings of *search* in which we should be interested:

1. *Search stored data*. The problem to be solved here is to efficiently retrieve stored information; an example can be search in computer memory. This can be applied to enormous databases, which nowadays can be found in many forms on the

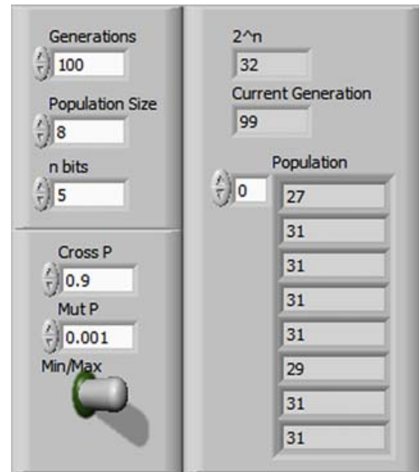

Fig. 5.13 GA-X squared showing some results

Internet. For example, what is the best way to search for hotels in a particular city? *Binary search* is one method for efficiently finding a desired record in a database.

2. *Search paths and goals*. This search form can be seen as the movements to go from a desired initial state to a final state, like the shortest path in a maze.
3. *Search for solutions*. This is a more general search form of the search for paths and goals. This happens because a path through a search tree can be encoded as a candidate solution. The idea is to efficiently find a solution to a problem in a large space of candidate solutions. These are the most common problems to which GAs are applied.

5.6 Applications of Genetic Algorithms

Even though the foundations of GAs are very simple, certain variations have been used in a large number of scientific and engineering applications. Some of them are given below:

- *Optimization*. The optimization of mathematical functions, combinatorial and numerical problems such as circuit layout and job-shop schedule.
- *Automatic programming*. Used to evolve computer programs, and to design computational structures.
- *Machine learning*. Classification and prediction tasks, such as weather or protein structure. GAs have been used to evolve aspects of particular machine learning systems such as the weights in neural networks, rules for learning classifier systems, sensors and robots.
- *Economics*. Development of bidding strategies, model processes of innovation, strategies, emergence of markets.

- *Immune systems*. The evolution of evolutionary time in multi-gene families, somatic mutation, natural immune systems.
- *Ecology*. Model ecological phenomena such as symbiosis, host-parasite co-evolution, and resource flow.
- *Evolution and learning*. Used in the study of how individual learning and species evolutions affect one another.
- *Social systems*. Used to study evolutionary aspects of social systems, like the evolution of social behavior in insect colonies, and the evolution of cooperation and communication in multi-agent systems.

5.7 Pros and Cons of Genetic Algorithms

There are numerous advantages to using a GA, such as not depending on analytical knowledge, robustness, and intuitive operation. All of these characteristics have made GAs strong candidates in search and optimization problems. However, there are also several disadvantages to using GAs that have made researchers turn to other search techniques, such as:

- Probabilistic.
- Expensive in computational resources.
- Prone to premature convergence.
- Difficult to encode a problem in the form of a chromosome.

There are several alternatives that have been found, especially due to the difficulty of encoding the problems. Messy GAs and genetic programming are two techniques that are based on the framework of GAs.

5.8 Selecting Genetic Algorithm Methods

The representation, recombination, mutation, and selection are complex balances between exploitation and exploration. It is a matter of precision to maintain this balance, thus there are several key factors that can help us correctly choose the encoding technique.

Encoding. This is a key issue with most evolutionary algorithms, whether to choose a suitable encoding scheme, which could be binary, floating-point, or grammatical. On the one hand, Holland supports the idea of a genome with a smaller number of alleles with long strings, rather than a numeric scheme with a larger number of alleles but short (floating-point) strings. On the other hand, M. Mitchell points out that for real-world applications it is more natural to use decimals or symbolic representation [6]. The conclusion from Z. Michalewicz about this is that a floating-point scheme is faster, more consistent between runs, and can provide a higher precision for large-domain applications.

Operator choice. There are a few general guidelines to follow when choosing an operator. Many advantages can be obtained from a two-point crossover operation, by reducing the disruption of schemas with a long length. The choice of the mutation depends heavily on the application, a practical alternative is an adaptive mutation parameterized within the genome.

Elitism. Another common technique in many GAs is to retain the best candidate in each generation and pass it to the next; this can give a significant boost to the performance of the GA, although with the risk of reducing diversity.

5.9 Messy Genetic Algorithm

There are several approaches with regard to the modification of certain aspects of GAs, with the aim of improving their performance. Messy GAs were proposed by D. Goldberg and co-workers in 1989, where they used variable-length binary encodings of chromosomes.

Each gene of the chromosome is represented by a pair (position and value), ensuring the adaptation of the algorithm to a larger variety of situations. Moreover this representation prevents the problems generated by recombination. A messy GA adapts its representation to the problem being solved.

The operators used in this algorithm are generalizations of standard genetic operators that use binary encoding. The main disadvantage against fixed-length representations is that they lack dynamic variability; thus, by limiting the string length the search space is limited. To overcome this, variable-length representations allow us to deal with partial information or to use contradictory information.

Messy encoding. Chromosome length is variable and genes may be arranged in any order (messy), where the last characteristic is the one that gives the algorithm its name. Each gene is represented by a pair of numbers. The first component is a natural number that encodes the gene location; the second number represents the gene value, which usually is either 0 or 1.

Example 5.2. Considering the binary encoded chromosome $x = (01101)$, which can be transformed into the following sequence: $x' = ((1,0),(2,1),(3,1),(4,0),(5,1))$. The meaning of this chromosome does not change if the pairs are arranged in a different order, for instance the following chromosome: $x'' = ((2,1),(3,1),(1,0),(4,0),(5,1))$. □

Incompleteness and ambiguity. As chromosomes have a flexible structure, we may consider missing one or more genes, which is called an *underspecified* string. This allows us to encode and deal with incomplete information. The opposite situation is overspecification, which occurs when a string contains multiple pairs for the same gene creating redundant or even contradictory genes.

To deal with overspecification, certain rules can be applied such as the tie-breaking mechanism that essentially says "first-come, first served," so that only the first of the repeated genes is taken into consideration. To deal with underspecified strings several possibilities exist, like looking for the complete chromosome that is

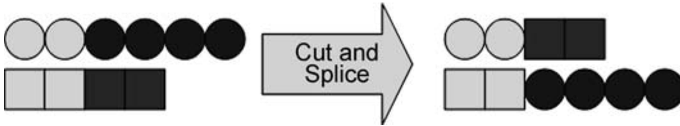

Fig. 5.14 Cut and splice operation

closest to the underspecified string. Another way is to try to approximate the absent value or to identify the probability p that the missing gene has the value of 1; if the value is 0 then the probability will be $(1 - p)$. Another way to do it is by using competitive templates, considered as locally optimal strings.

Crossover. The classical n-point crossover is replaced by the *cut-and-splice* operator, which acts very similarly to a one-point crossover. Two parents are cut in two and the resulting substrings are recombined. The position for the crossover is chosen with a probability that is uniform to the string length. The difference is that the crossover points are independent from the two parents.

The *splice* operation concatenates the substrings obtained through cutting, where Fig. 5.14 shows this operation. There is no restriction regarding the way in which substrings are combined. The tie-breaking rule along with competitive templates is used to handle overspecified and underspecified strings.

Messy GAs have provided results for difficult problems. In the case of deceptive functions, messy GAs perform better than simple GAs, usually finding the best solution. An important computational problem within messy algorithms is the dimension of the search space, i. e., since large chromosomes may appear, the dimension could be very high. The search space size is a polynomial. In parallel implementations, the search time is reduced and it is logarithmic with respect to the number of variables of the search space.

5.10 Optimization of Fuzzy Systems Using Genetic Algorithms

A brief explanation on how GAs can be applied to optimize the performance of a fuzzy controller is given in this section.

5.10.1 Coding Whole Fuzzy Partitions

There is always knowledge of the desired configuration, for example, the number of clusters and the labels for each one, where a natural order of the fuzzy sets can be established. By including the proper constraints, the initial conditions can be preserved while reducing the number of degrees of freedom in order to maintain the interpretability of a fuzzy system. Thus, we can encode a whole fuzzy partition as shown in (5.1), where σ is the upper boundary for the size of the offset for example $\sigma = (b - a)/2$. Figure 5.15 shows the coded triangular membership function.

$$C_{n,[0,\partial]}(x_1) \to C_{n,[0,\partial]}(x_2 - x_1) \cdots C_{n,[0,\partial]}(x_{2N-2} - x_{2N-3}) \;. \tag{5.1}$$

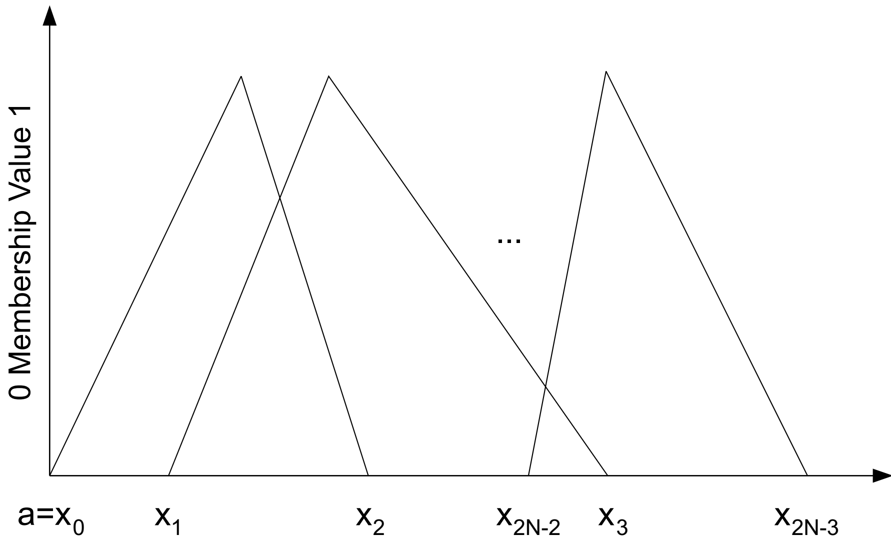

Fig. 5.15 Coded triangular membership functions

5.10.2 Standard Fitness Functions

We define the fitness function as a function that will minimize the distance between a set of representative inputs and outputs, and the values computed by the next function, where the sum of quadratic errors is calculated using (5.2):

$$f(\boldsymbol{v}) = \sum_{i=1}^{k} (F(\boldsymbol{v}, x_i) - y_i)^2 . \tag{5.2}$$

Here, $F(\boldsymbol{v}, x_i)$ is the function that computes the output with respect to the parameter vector, (x_i, y_i) is the sample data given as a list of couples, $1 \leq i \leq k$, and k is the number of samples.

5.10.3 Coding Rule Bases

So far we have explained in which way membership functions can be encoded to be optimized with GAs, but if we find a proper method to encode rule bases into a string of fixed lengths we can apply the previously explained GAs to optimize them without modification.

We must assume that the number of linguistic values of all linguistic variables are finite. A rule base is represented as a list for one input variable, as a matrix for two variables, and as a tensor in the case of more than two variables. This rule can be represented by a matrix as shown in Fig. 5.16. Consider the rule base of the form

Fig. 5.16 Example decision table

	B_1 . . . B_{N2}
A_1	$C_{1,1}$ . . . $C_{i,N2}$
.	. . .
.	. . .
.	. . .
A_{N1}	$C_{N1,1}$ . . . $C_{N1,N2}$

in (5.3):

$$\text{IF } x_1 \text{ is } A_i \text{ AND } x_2 \text{ is } B_j \text{ THEN } C_{i,j} \, . \tag{5.3}$$

We can now assign indices to the linguistic values associated with elements of the set $\{C_{i,j}\}$. We can later write the decision table as an integer string, and convert those numbers to bits, where the previously mentioned GAs are perfectly suitable to optimize the rule base.

5.11 An Application of the ICTL for the Optimization of a Navigation System for Mobile Robots

A navigation system based on Bluetooth technology was designed for controlling a quadruped robot in unknown environments which has ultrasonic sensor as inputs for avoiding static and dynamic obstacles. The robot Zil I is controled by a fuzzy logic controller Sugeno Type, which is shown in Fig. 2.24 and Zil I is shown in Fig. 5.17, the form of the membership functions for the inputs are triangular and the

Fig. 5.17 Robot Zil I was controlled by a Fuzzy Logic Controller adjusted using Genetic AlgorithmsC

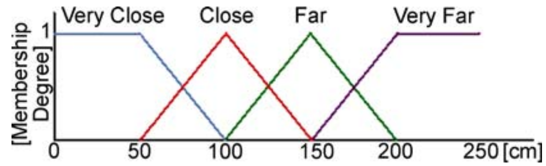

Fig. 5.18 Triangular membership functions

initial membership function's domain and shape are shown in Fig. 5.18. The block diagram of the fuzzy controller is shown in Sect. 5.11 ICTL for the Optimization of a Navigation System for Mobile Robots.

The navigation system is based on a Takagi–Sugeno controller, which is shown in Fig. 5.17. The form of the membership function is triangular, and the initial limits are shown in Fig. 5.18. The block diagram of the fuzzy controller is shown in Fig. 5.19. Based on the scheme of optimization of fuzzy systems using GAs, the fuzzy controller was optimized. Some initial individuals where created using expert knowledge and others were randomly created. In Fig. 5.20 we see the block diagram of the GA.

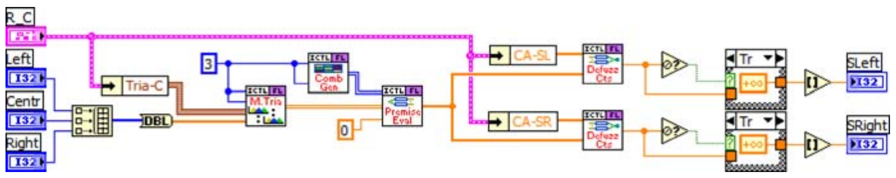

Fig. 5.19 Block diagram of the Takagi–Sugeno controller

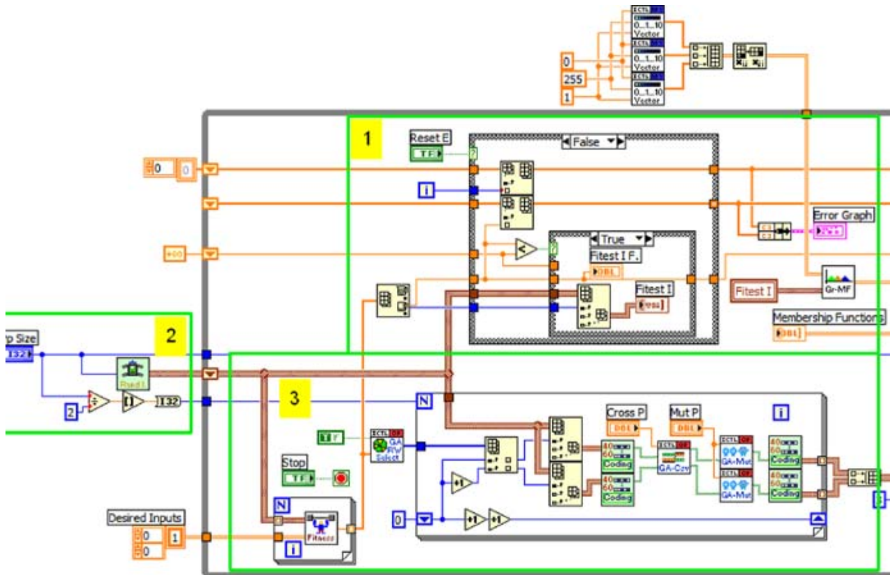

Fig. 5.20 Block diagram of the GA

Inspecting the block diagram we find that the GA created previously, used for the optimization of the $f(x) = x^2$ function, remains the same. The things that change here are the coding and decoding functions, as well as the fitness function. There is also some code used to store the best individuals. After running the program for a while, the form of the membership functions will vary from our initial guess, as shown in Fig. 5.21, and it will find an optimized solution that will fit the constrains set by the human expert knowledge and the requirements for the application. The solutions are shown in Fig. 5.22.

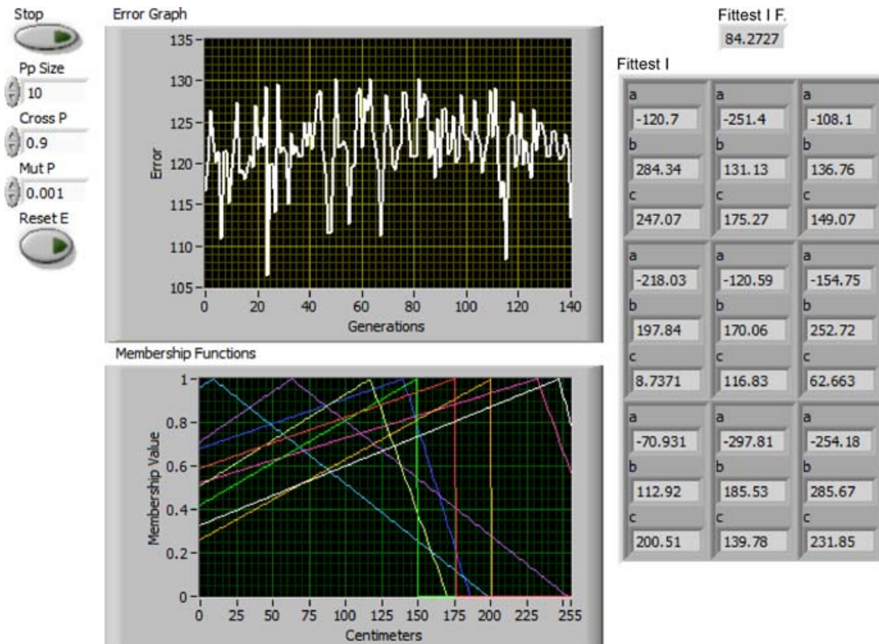

Fig. 5.21 Results shown by the GA after some generations

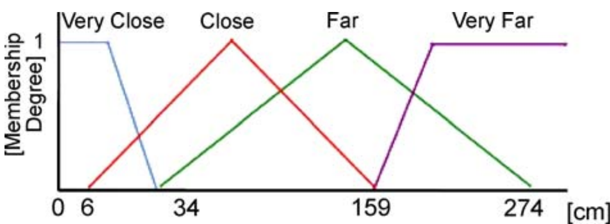

Fig. 5.22 Optimized membership functions

5.12 Genetic Programming Background

Evolution is mostly determined by natural selection, which can be described as individuals competing for all kinds of resources in the environment. The better the individuals, the more likely they will propagate their genetic material. Asexual reproduction creates individuals identical to their parents; this is done by the encoding of genetic information. Sexual reproduction produces offspring that contain a combination of information from each parent, and is achieved by combining and reordering the chromosomes of both parents.

Evolutionary algorithms have been applied to many problems such as optimization, machine learning, operation research, bioinformatics and social systems, among many others. Most of the time the mathematical function that describes the system is not known and the parameters that are known are found through simulation.

Genetic programming, evolutionary programming, evolution strategies and GAs are usually grouped under the term evolutionary computation, because they all share the same base of simulating the evolution of individual structures. This process depends on the way that performance is perceived by the individual structures as defined by the problem.

Genetic programming deals with the problem of automatic programming; the structures that are being evolved are computers programs. The process of problem solving is regarded as a search in the space of computer programs, where genetic programming provides a method for searching the fittest program with respect to a problem. Genetic programming may be considered a form of *program discovery.*

5.12.1 Genetic Programming Definition

Genetic programming is a technique to automatically create a working computer program from a high-level statement of the problem. This is achieved by genetically breeding a population of computer programs using the principles of Darwinian natural selection and biologically inspired operators. It is the extension of evolutionary learning into the space of computer programs.

The individual population members are not fixed-length character strings that encode possible solutions of the problem, they are programs that when executed are the candidate solutions to the problem. These programs are represented as trees. There are other important components of the algorithm called terminal and function sets. The terminal set consists of variables and constants. The function sets are the connectors and operators that relate the constants and variables.

Individuals evolved from genetic programming are program structures of variable sizes. A user-defined language with appropriate operators, variables, and constants may be defined for the particular problem to be solved. This way programs will be generated with an appropriate syntax and the program search space limited to feasible solutions.

5.12.2 Historical Background

A.M. Turing in 1950, considered the fact that genetic or evolutionary searches could automatically develop intelligent computer programs, like chess player programs and other general purpose intelligent machines. Later in 1980, Smith proposed a classifier system that could find good poker playing strategies using variable-sized strings that could represent the strategies. In 1985, Cramer considered a tree structure as a program representation in a genotype. The method uses tree structures and subtree crossover in the evolutionary process.

Genetic programming was first proposed by Cramer in 1985 [7], and further developed by Koza [8], as an alternative to fixed-length evolutionary algorithms by introducing trees of different shapes and sizes. The symbols used to create these structures are more varied than zeros and ones used in GAs. The individuals are represented by genotype/phenotype forms, which make them non-linear. They are more like protein molecules in their complex and unique hierarchical representation. Although parse trees are capable of exhibiting a great variety of functionalities, they are highly constrained due to the form of tree, the branches are the ones that are modified.

5.13 Industrial Applications

Some interesting applications of genetic programming in the industry are mentioned here. In 2006 J.U. Dolinsky and others [9] presented a paper with an application of genetic programming to the calibration of industrial robots. They state that most of the proposed methods address the calibration problem by establishing models followed by indirect and often ill-conditioned numeric parameter identification. They proposed an inverse static kinematic calibration technique based on genetic programming, used to establish and identify model parameters.

Another application is the use of genetic programming for drug discovery in the pharmaceutical industry [10]. W.B. Langdon and S.K. Barrett employed genetic programming while working in conjunction with GlaxoSmithKline (GSK). They were invited to predict biochemical activity using their favorite machine learning technique. Their genetic programming was the best of 12 tested, which marginally improved the existing system of GSK.

5.14 Advantages of Evolutionary Algorithms

Probably the greatest advantage of evolutionary algorithms is their ability to address problems for which there are no human experts. Although human expertise is to be used when available, it has proven less than adequate for automating problem-solving routines.

A primary advantage of this kind of algorithm is that they are simple to represent. They can be modeled as a difference equation $x[t+1] = s(r(x[t]))$, which can

be understood as: $x[t]$ is the population at time t under the representation x, is the random variation operator and s is the selection operator.

The representation does not affect the performance of the algorithm, in contrast with other numerical techniques, which are biased on continuous values or constrained sets. They offer a framework to easily incorporate known knowledge of the problem, which could yield in a more efficient exploration and response of the search space.

Evolutionary algorithms can be combined with simple or complex traditional optimization techniques. Most of the time the solution can be evaluated in parallel, and only the selection must be processed serially. This is an advantage over other optimization techniques like tabu search and simulated annealing. Evolutionary algorithms can be used to adapt solutions to changing circumstances, because traditional methods are not robust to dynamic changes and often require a restart to provide the solution.

5.15 Genetic Programming Algorithm

In 1992, J.R. Koza developed a variation of GAs that is able to automate the generation of computer programs [8]. Evolutionary algorithms, also known as evolutionary computing, are the general principles of natural evolution that can be applied to completely artificial environments. GAs and genetic programming are types of evolutionary computing.

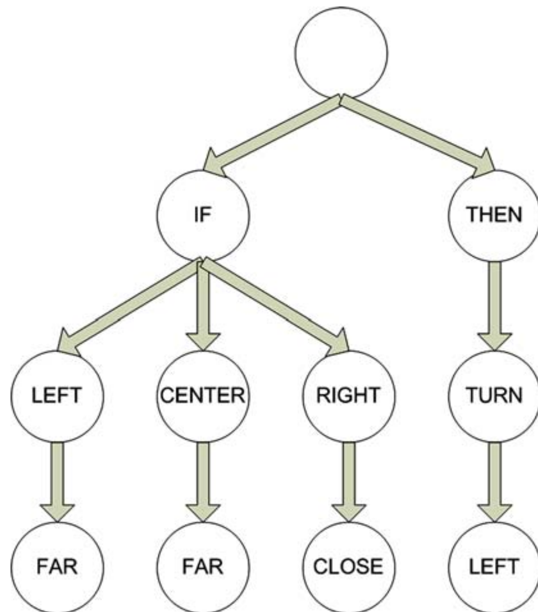

Fig. 5.23 Tree representation of a rule

Genetic programming is a computing method, which provides a system with the possibility of generating optimized programs or computer codes. In genetic programming *IF-THEN* rules are coded into individuals, which often are represented as trees. For example, a rule for a wheeled robot may be *IF left is far AND center if far AND right is close THEN turn left*. This rule is represented as a tree in Fig. 5.23.

According to W. Banzhaf "genetic programming, shall include systems that constitute or contain explicit references to programs (executable code) or to programming language expressions."

5.15.1 Length

In GAs the length of the chromosome is fixed, which can restrict the algorithm to a non-optimal region of the problem in search space. Because of the tree representation, genetic programming can create chromosomes of almost any length.

5.16 Genetic Programming Stages

Genetic programming uses four steps to solve problems:

1. Generate an initial population of random compositions of functions and terminals of the problem (computer programs).
2. Execute each program in the population and assign it a fitness value according to how well it solves the problem.
3. Create a new population of computer programs:
   a. Copy the best existing programs.
   b. Create new programs by mutation.
   c. Create new computer programs by crossover.
4. The best computer program that appeared in any generation, the best-so-far solution, is designated the result of genetic programming [8].

Just like in GAs, in genetic programming the stages are initialization, selection, crossover, and mutation.

5.16.1 Initialization

There are two methods for creating the initial population in a genetic programming system:

1. *Full* selects nodes from only the function set until a node is at a specified maximum depth.
2. *Grow* randomly selects nodes from the function and terminal set, which are added to a new individual.

5.16.2 Fitness

It could be the case that a function to be optimized is available, and we will just need to program it. But for many problems it is not easy to define an objective function. In such a case we may use a set of training examples and define the fitness as an error-based function. These training examples should describe the behavior of the system as a set of input/output relations.

Considering a training set of k examples we may have (x_i, y_i), $i = 1, \ldots, k$, where x_i is the input of the ith training sample and y_i is the corresponding output. The set should be sufficiently large to provide a basis for evaluating programs over a number of different significant situations.

The fitness functionmay also be defined as the total sum of squared errors; it has the property of decreasing the importance of small deviations from the target outputs. If we define the error as $e_i = (y_i - o_i)^2$ where y_i is the desired output and o_i the actual output, then the fitness will be defined as $\sum_{i=1}^{k} e_k = \sum_{i=1}^{k} (y_i - o_i)^2$. The fitness function may also be scaled, thus allowing amplification of certain differences.

5.16.3 Selection

Selection operators within genetic programming are not specific; the problem under consideration imposes a particular choice. The choice of the most appropriate selection operator is one of the most difficult problems, because generally this choice is problem-dependent. However, the most-used method for selecting individuals in genetic programming is tournament selection, because it does not require a centralized fitness comparison between all individuals. The best individuals of the generation are selected.

5.16.4 Crossover

The form of the recombination operators depends on the representation of individuals, but we will restrict ourselves to tree-structured representations. An elegant and rather straightforward recombination operator acting on two parents swaps a subtree of one parent with a subtree of the other parent.

There is a method proposed by H. Iba and H. Garis to detect regularities in the tree program structure and to use them as guidance for the crossover operator. The method assigns a performance value to a subtree, which is used to select the crossover points. Thus, the crossover operator learns to choose good sites for crossover.

Simple crossover operation. In a random position two trees interchange their branches, but it should be in a way such that syntactic correctness is maintained. Each offspring individual will pass to the selection process of the next generation. In Fig. 5.24 a representation of a crossover is shown.

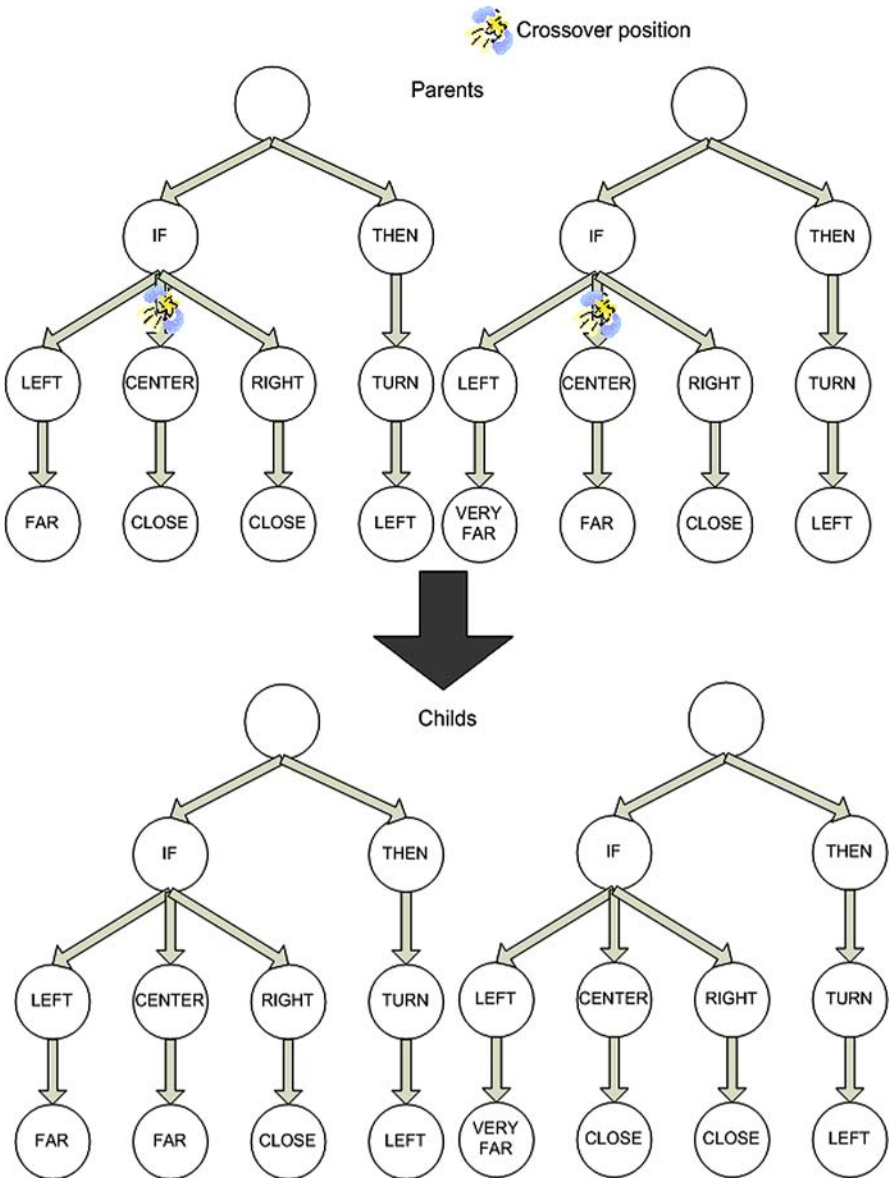

Fig. 5.24 Tree representation of a genetic programming crossover stage

5.16.5 Mutation

There are several mutation techniques proposed for genetic programming. An example is the mutation of tree-structured programs; here the mutation is applied to a single program tree to generate an offspring. If our program is linearly repre-

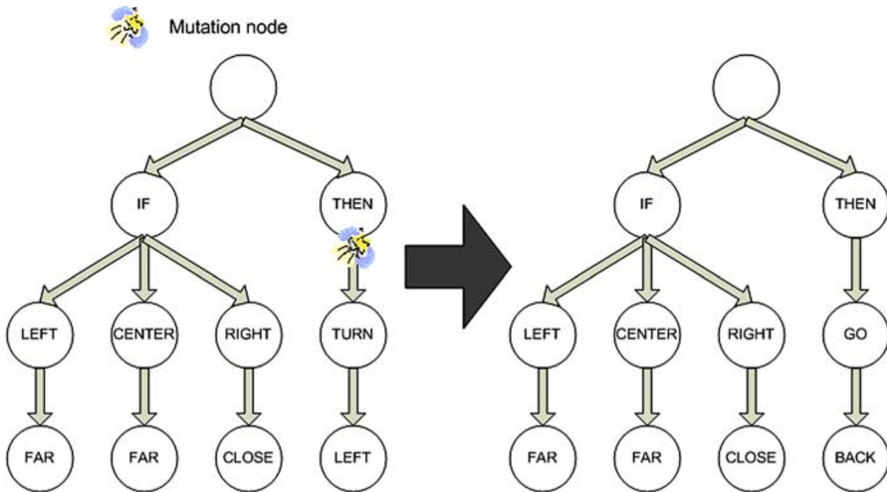

Fig. 5.25 Tree representation of a genetic programming mutation stage

sented, then the mutation operator selects an instruction from the individual chosen for mutation. Then, this selected instruction is randomly perturbed, or is changed to another instruction randomly chosen from a pool of instructions.

The usual strategy is to complete the offspring population with a crossover operation. On this kind of population the mutation is applied with a specific mutation probability. A different strategy considers a separate application of crossover and mutation. In this case it seems to be emphasized with respect to the previous, standard technology.

In genetic programming, the generated individuals are selected with a very low probability of being mutated. When an individual is mutated, one of its nodes is selected randomly and then the current subtree at that point is replaced with a new randomly generated subtree. It is important to state that just as in biological mutation, in genetic programming mutation the genotype may not change but the resulting genotype could be completely different (Fig. 5.25).

5.17 Variations of Genetic Programming

Several variations of genetic programming can be found in the literature. Some of them are linear genetic programming, a variant that acts on linear genomes rather than trees; gene expression programming, where the genotype (a linear chromosome) and the phenotype (expression trees) are different entities that form an indivisible whole; multi-expression programming encodes several solutions into a chromosome; Cartesian genetic programming uses a network of nodes (indexed graph) to achieve an input-to-output mapping; and traceless genetic programming, which does not store explicitly the evolved computer programs, and is useful when the relation between the input and output is not important.

5.18 Genetic Programming in Data Modeling

The main purpose of evolutionary algorithms is to imitate natural selection and evolution, allowing the most efficient individuals to reproduce more often. Genetic programming is very similar to GAs; the main difference is that genetic programming uses different coding of potential solutions. By using knowledge from great amounts of data collected from different places, we can discover patterns and represent them in a way that humans can understand them.

By mathematical modeling we understand that certain equations fit some numerical data. It is used in a variety of scientific problems, where the theoretical foundations are not enough to give answers to experiments. Sometimes using traditional methods is not enough because these methods assume a specific form of model. Genetic programming allows the search for a good model in a different and more "intelligent" way, and can be used to solve highly complex, non-linear, chaotic problems.

5.19 Genetic Programming Using the ICTL

Here we will continue with the optimization example of fuzzy systems. As we have mentioned previously, a Takagi–Sugeno is the core controller of a navigation system that maneuvers the movements of a quadruped robot in order to avoid obstacles. We have previously optimized the form of the membership functions using GAs and now we will evolve the form of the rules and modify their operators. The rules for the controllers used in the quadruped robot have the form of (5.4):

$$\textit{IF}\ \text{Left is}\ \textit{ALi Conn}\ \text{Central is}\ \textit{ACi Conn}\ \text{Right is}\ \textit{ARi THEN SLeft}, \textit{SRight}, \tag{5.4}$$

where $ALi = ACi = ARi$ are the number of fuzzifying membership functions for the inputs, in this case they are the same, and *Conn* is the operation to be performed. There are four options: (1) *min*, (2) *max*, (3) *product*, and (4) *sum*. *SLeft, SRight* are the speeds used to control the movements of the robot.

Genetic programming will be applied using the following convention to code-decode the individuals:

- 3 bits that will help us determine if the set is complemented or not. A bi-dimensional array must be generated with the same form of the CM-A of the **input-combinator-generator.vi** that is used to evaluate the different sets of the possible rule combinations. The first dimension contains the number of rules, the second the number of inputs to the system.
- 2 bits are used for the premise evaluation. The premises of the next connection operation are used: (0) *min*, (1) *max*, (2) *product*, and (3) *sum*.
- 10 bits are used to obtain the outputs of the rule, 5 for each output to obtain a constant between 0 and 31.

Table 5.4 Individual rule coding for genetic programming ICTL example

Three bits *IS, IS NOT*			Two bits *Conn*		10 bits for rule output									
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

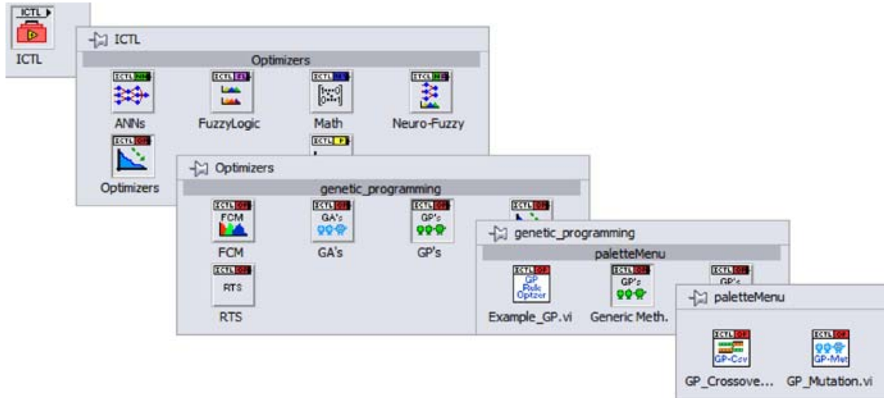

Fig. 5.26 Localization of genetic programming methods on the ICTL

The rule in bits is shown in Table 5.4. As shown in Fig. 5.26, the methods for genetic programming are found at Optimizers ≫ GPs ≫ Generic Methods. An individual contains 27 of these rules; the **initial_population.vi** initializes a population with random individuals. The code is shown in Fig. 5.27. Fixed individuals may be added based on human expert knowledge.

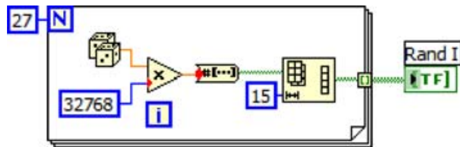

Fig. 5.27 Block diagram for the initialization of random individuals

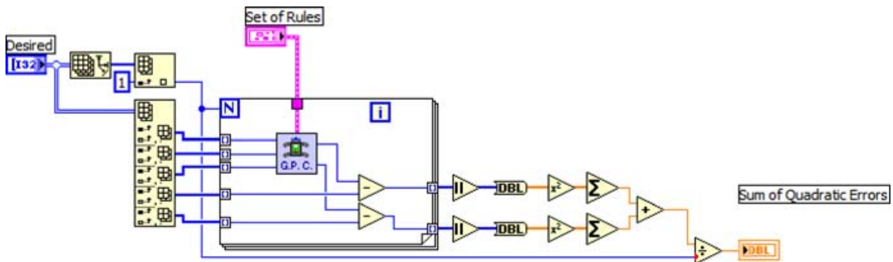

Fig. 5.28 Block diagram of fitness function

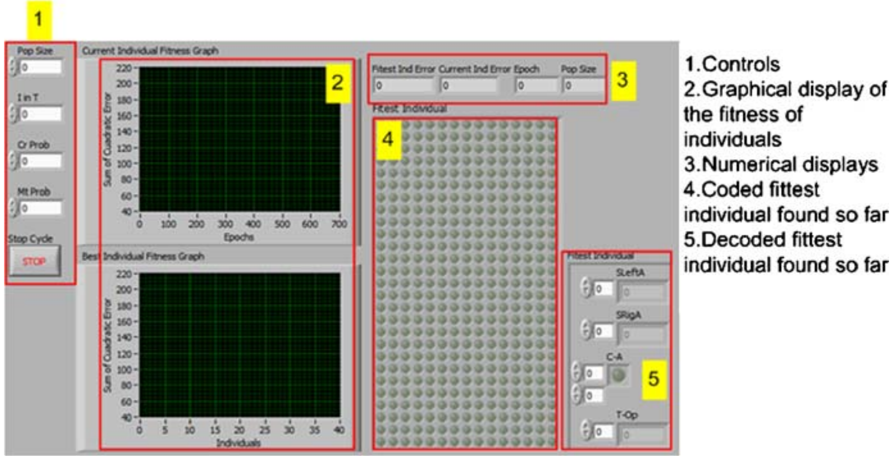

Fig. 5.29 Front panel of the genetic programming example

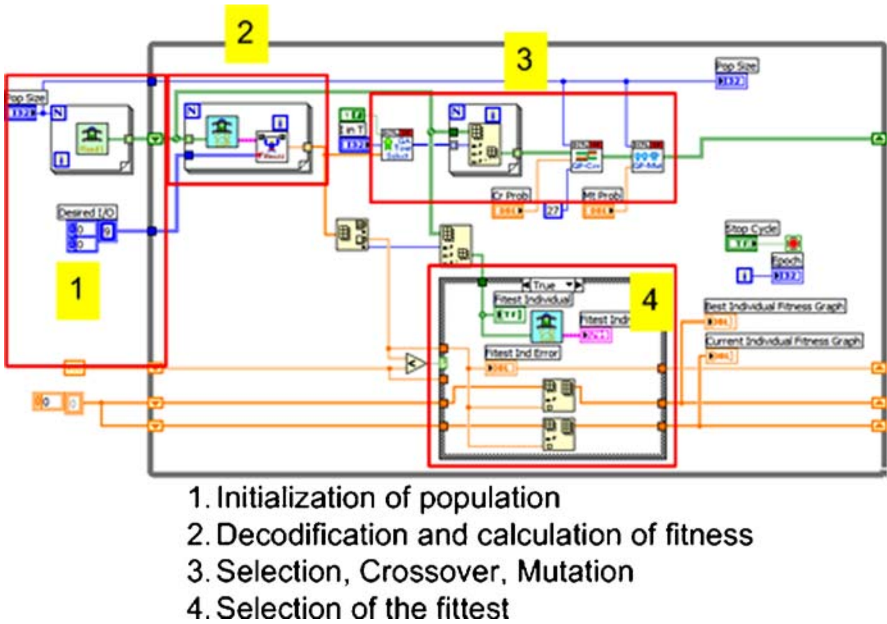

Fig. 5.30 Block diagram of the genetic programming example

The fitness function (Figs. 5.28–5.30) compares a series of desired inputs and outputs with the corresponding performance of the controller, calculating the quadratic errors difference with each point, summing them and dividing by the number of evaluated points to obtain the fitness value for a given individual.

The *selection* function executes the tournament variation by randomly selecting a desired number of individuals and selecting the two fittest. This process is repeated until the same number of initial individuals is obtained.

Table 5.5 Controlled variables in the genetic programming example

Variable	Description
Pop Size	The number of individuals in the algorithm.
I in T	The number of individuals randomly taken for the tournament selection.
Cr Prob	The probability of crossing [0, 1].
Mt Prob	The probability of mutation for each bit [0, 1].

The *crossover* executes a one-stage interchange of tails, by taking two individuals from the mating pool, and depending on the possibility of crossing, the two individuals will or will not perform the crossover again. This process is repeated until the same number of initial individuals is obtained.

The *mutation* process executes a bit-to-bit operation on every one of the rules of each individual, and depending on the probability of mutation the bit will or will not change. During the execution of this algorithm, the individual with the best fitness is always stored to ensure that this information is never lost. Table 5.5 shows the variables to be controlled.

References

1. Wang F, et al. (2006) Design optimization of industrial motor drive power stage using genetic algorithms. Proceedings of CES/IEEE 5th I nternational Power Electronics and Motion Control Conference (IPEMC), 1–5 Aug 2006, vol 1, pp 14–16
2. Cho D-H, et al. (2001) Induction motor design of electric vehicle using a niching genetic algorithm. IEEE Trans Ind Appl USA 37(4):994–999
3. Cavalieri S (1999) A genetic algorithm for job-shop scheduling in a semiconductor manufacturing system. Proceedings of the 25th Annual Conference of the IEEE on Industrial Electronics Society (IECON) 1999, Italy, 29 Nov to 3 Dec 1999, vol 2, pp 957–961
4. Colla V, et al. (1998) Model parameter optimization for an industrial application: a comparison between traditional and genetic algorithms. Proceedings of the IEEE 2nd UKSIM European Symposium on Computer Modeling and Simulation, 8–10 Sept 2008, pp 34–39
5. Haupt RL, Haupt SE (1998) Practical genetic algorithms. Wiley-Interscience, New York
6. Mitchell M (1998) An introduction to genetic algorithms. MIT Press, Cambridge, MA
7. Cramer NL (1985) A representation for the adaptive generation of simple sequential programs. In: Grefenstette JJ (ed) Proceedings of the First International Conference on Genetic Algorithms and Their Applications. Erlbaum, Mahwah, NJ
8. Koza JR (1992) Genetic programming: on the programming of computers by means of natural selection. MIT Press, Cambridge, MA
9. Dolinsky JU, et al. (2007) Application of genetic programming to the calibrating of industrial robots. ScienceDirect Comput Ind 58(3):255–264
10. Langdon WB, Buxton BF (2003) The application of genetic programming for drug discovery in the pharmaceutical industry. EPSRC RIAS project with GlaxoSmithKline. London, UK, September 2003

Futher Reading

Dumitrescu D, et al. (2000) Evolutionary computation. CRC, Boca Raton, FL

Ghanea-Hercock R (2003) Applied evolutionary algorithms in Java. Springer, Berlin Heidelberg New York

Nedjah N, et al. (2006) Genetic systems programming theory and experiences. Springer, Berlin Heidelberg New York

Reeves CR, Rowe JE (2004) Genetic algorithms principles and perspectives: A guide to GA theory. Kluwer, Dordrecht