
Evolutionary Computation: from Genetic Algorithms to Genetic Programming

Ajith Abraham¹, Nadia Nedjah² and Luiza de Macedo Mourelle³

¹ School of Computer Science and Engineering Chung-Ang University 410,
2nd Engineering Building 221, Heukseok-dong,
Dongjak-gu Seoul 156-756, Korea

ajith.abraham@ieee.org, <http://www.ajith.softcomputing.net>

² Department of Electronics Engineering and Telecommunications,
Engineering Faculty,
State University of Rio de Janeiro,
Rua São Francisco Xavier, 524, Sala 5022-D,
Maracanã, Rio de Janeiro, Brazil

nadia@eng.uerj.br, <http://www.eng.uerj.br/~nadia>

³ Department of System Engineering and Computation,
Engineering Faculty,
State University of Rio de Janeiro,
Rua São Francisco Xavier, 524, Sala 5022-D,
Maracanã, Rio de Janeiro, Brazil

ldmm@eng.uerj.br, <http://www.eng.uerj.br/~ldmm>

Evolutionary computation, offers practical advantages to the researcher facing difficult optimization problems. These advantages are multi-fold, including the simplicity of the approach, its robust response to changing circumstance, its flexibility, and many other facets. The evolutionary approach can be applied to problems where heuristic solutions are not available or generally lead to unsatisfactory results. As a result, evolutionary computation have received increased interest, particularly with regards to the manner in which they may be applied for practical problem solving.

In this chapter, we review the development of the field of evolutionary computations from standard genetic algorithms to genetic programming, passing by evolution strategies and evolutionary programming. For each of these orientations, we identify the main differences from the others. We also, describe the most popular variants of genetic programming. These include linear genetic programming (LGP), gene expression programming (GEP), multi-expression programming (MEP), Cartesian genetic programming (CGP), traceless genetic programming (TGP), gramatical evolution (GE) and genetic algorithm for deriving software (GADS).

1.1 Introduction

In nature, evolution is mostly determined by natural selection or different individuals competing for resources in the environment. Those individuals that are better are more likely to survive and propagate their genetic material. The encoding for genetic information (genome) is done in a way that admits asexual reproduction which results in offspring that are genetically identical to the parent. Sexual reproduction allows some exchange and re-ordering of chromosomes, producing offspring that contain a combination of information from each parent. This is the recombination operation, which is often referred to as crossover because of the way strands of chromosomes cross over during the exchange. The diversity in the population is achieved by mutation.

Evolutionary algorithms are ubiquitous nowadays, having been successfully applied to numerous problems from different domains, including optimization, automatic programming, machine learning, operations research, bioinformatics, and social systems. In many cases the mathematical function, which describes the problem is not known and the values at certain parameters are obtained from simulations. In contrast to many other optimization techniques an important advantage of evolutionary algorithms is they can cope with multi-modal functions.

Usually grouped under the term evolutionary computation [1] or evolutionary algorithms, we find the domains of genetic algorithms [9], evolution strategies [17, 19], evolutionary programming [5] and genetic programming [11]. They all share a common conceptual base of simulating the evolution of individual structures via processes of selection, mutation, and reproduction. The processes depend on the perceived performance of the individual structures as defined by the problem.

A population of candidate solutions (for the optimization task to be solved) is initialized. New solutions are created by applying reproduction operators (mutation and/or crossover). The fitness (how good the solutions are) of the resulting solutions are evaluated and suitable selection strategy is then applied to determine which solutions will be maintained into the next generation. The procedure is then iterated and is illustrated in Fig. 1.1.

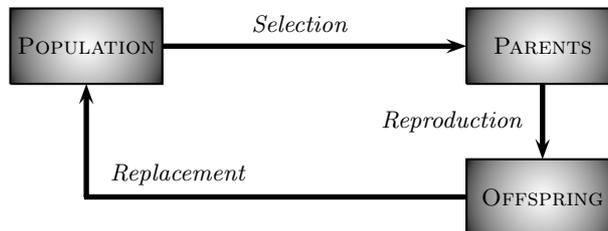


Fig. 1.1. Flow chart of an evolutionary algorithm

1.1.1 Advantages of Evolutionary Algorithms

A primary advantage of evolutionary computation is that it is conceptually simple. The procedure may be written as difference equation (1.1):

$$x[t + 1] = s(v(x[t])) \quad (1.1)$$

where $x[t]$ is the population at time t under a representation x , v is a random variation operator, and s is the selection operator [6].

Other advantages can be listed as follows:

- Evolutionary algorithm performance is representation independent, in contrast with other numerical techniques, which might be applicable for only continuous values or other constrained sets.
- Evolutionary algorithms offer a framework such that it is comparably easy to incorporate prior knowledge about the problem. Incorporating such information focuses the evolutionary search, yielding a more efficient exploration of the state space of possible solutions.
- Evolutionary algorithms can also be combined with more traditional optimization techniques. This may be as simple as the use of a gradient minimization used after primary search with an evolutionary algorithm (for example fine tuning of weights of a evolutionary neural network), or it may involve simultaneous application of other algorithms (e.g., hybridizing with simulated annealing or tabu search to improve the efficiency of basic evolutionary search).
- The evaluation of each solution can be handled in parallel and only selection (which requires at least pair wise competition) requires some serial processing. Implicit parallelism is not possible in many global optimization algorithms like simulated annealing and Tabu search.
- Traditional methods of optimization are not robust to dynamic changes in problem the environment and often require a complete restart in order to provide a solution (e.g., dynamic programming). In contrast, evolutionary algorithms can be used to adapt solutions to changing circumstance.
- Perhaps the greatest advantage of evolutionary algorithms comes from the ability to address problems for which there are no human experts. Although human expertise should be used when it is available, it often proves less than adequate for automating problem-solving routines.

1.2 Genetic Algorithms

A typical flowchart of a Genetic Algorithm (GA) is depicted in Fig. 1.2. One iteration of the algorithm is referred to as a generation. The basic GA is very generic and there are many aspects that can be implemented differently according to the problem (For instance, representation of solution or chromosomes, type of encoding, selection strategy, type of crossover and mutation

operators, etc.) In practice, GAs are implemented by having arrays of bits or characters to represent the chromosomes. The individuals in the population then go through a process of simulated evolution. Simple bit manipulation operations allow the implementation of crossover, mutation and other operations. The number of bits for every gene (parameter) and the decimal range in which they decode are usually the same but nothing precludes the utilization of a different number of bits or range for every gene.

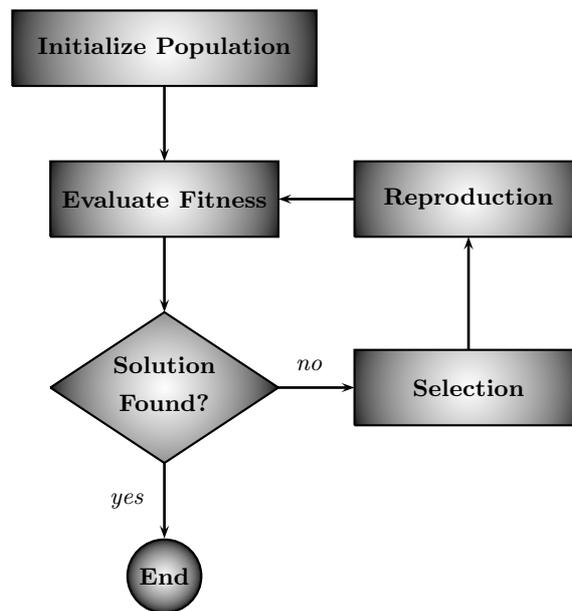


Fig. 1.2. Flow chart of basic genetic algorithm iteration

When compared to other evolutionary algorithms, one of the most important GA feature is its focus on fixed-length character strings although variable-length strings and other structures have been used.

1.2.1 Encoding and Decoding

In a typical application of GA's, the given problem is transformed into a set of genetic characteristics (parameters to be optimized) that will survive in the best possible manner in the environment. Example, if the task is to optimize the function given in 1.2.

$$\min f(x_1, x_2) = (x_1 - 5)^2 + (x_2 - 2)^2, -3 \leq x_1 \leq 3, -8 \leq x_2 \leq 8 \quad (1.2)$$

The parameters of the search are identified as x_1 and x_2 , which are called the phenotypes in evolutionary algorithms. In genetic algorithms, the phenotypes (parameters) are usually converted to genotypes by using a coding procedure. Knowing the ranges of x_1 and x_2 each variable is to be represented using a suitable binary string. This representation using binary coding makes the parametric space independent of the type of variables used. The genotype (chromosome) should in some way contain information about solution, which is also known as encoding. GA's use a binary string encoding as shown below.

Chromosome A: 110110111110100110110
 Chromosome B: 110111101010100011110

Each bit in the chromosome strings can represent some characteristic of the solution. There are several types of encoding (example, direct integer or real numbers encoding). The encoding depends directly on the problem.

Permutation encoding can be used in ordering problems, such as Travelling Salesman Problem (TSP) or task ordering problem. In permutation encoding, every chromosome is a string of numbers, which represents number in a sequence. A chromosome using permutation encoding for a 9 city TSP problem will look like as follows:

Chromosome A: 4 5 3 2 6 1 7 8 9
 Chromosome B: 8 5 6 7 2 3 1 4 9

Chromosome represents order of cities, in which salesman will visit them. Special care is to taken to ensure that the strings represent real sequences after crossover and mutation. Floating-point representation is very useful for numeric optimization (example: for encoding the weights of a neural network). It should be noted that in many recent applications more sophisticated genotypes are appearing (example: chromosome can be a tree of symbols, or is a combination of a string and a tree, some parts of the chromosome are not allowed to evolve etc.)

1.2.2 Schema Theorem and Selection Strategies

Theoretical foundations of evolutionary algorithms can be partially explained by schema theorem [9], which relies on the concept of schemata. Schemata are templates that partially specify a solution (more strictly, a solution in the genotype space). If genotypes are strings built using symbols from an alphabet A , schemata are strings whose symbols belong to $A \cup \{*\}$. This extra-symbol $*$ must be interpreted as a *wildcard*, being loci occupied by it called undefined. A chromosome is said to match a schema if they agree in the defined positions.

For example, the string 10011010 matches the schemata 1***** and **011*** among others, but does not match *1*1*** because they differ in the second gene (the first defined gene in the schema). A schema can be viewed

as a hyper-plane in a k -dimensional space representing a set of solutions with common properties. Obviously, the number of solutions that match a schema H depend on the number of defined positions in it. Another related concept is the *defining-length* of a schema, defined as the distance between the first and the last defined positions in it. The GA works by allocating strings to best schemata exponentially through successive generations, being the selection mechanism the main responsible for this behaviour. On the other hand the crossover operator is responsible for exploring new combinations of the present schemata in order to get the fittest individuals. Finally the purpose of the mutation operator is to introduce fresh genotypic material in the population.

1.2.3 Reproduction Operators

Individuals for producing offspring are chosen using a selection strategy after evaluating the fitness value of each individual in the selection pool. Each individual in the selection pool receives a reproduction probability depending on its own fitness value and the fitness value of all other individuals in the selection pool. This fitness is used for the actual selection step afterwards. Some of the popular selection schemes are discussed below.

Roulette Wheel Selection

The simplest selection scheme is roulette-wheel selection, also called stochastic sampling with replacement. This technique is analogous to a roulette wheel with each slice proportional in size to the fitness. The individuals are mapped to contiguous segments of a line, such that each individual's segment is equal in size to its fitness. A random number is generated and the individual whose segment spans the random number is selected. The process is repeated until the desired number of individuals is obtained. As illustrated in Fig. 1.3, chromosome₁ has the highest probability for being selected since it has the highest fitness.

Tournament Selection

In tournament selection a number of individuals is chosen randomly from the population and the best individual from this group is selected as parent. This process is repeated as often as individuals to choose. These selected parents produce uniform at random offspring. The tournament size will often depend on the problem, population size etc. The parameter for tournament selection is the tournament size. Tournament size takes values ranging from 2 – number of individuals in population.

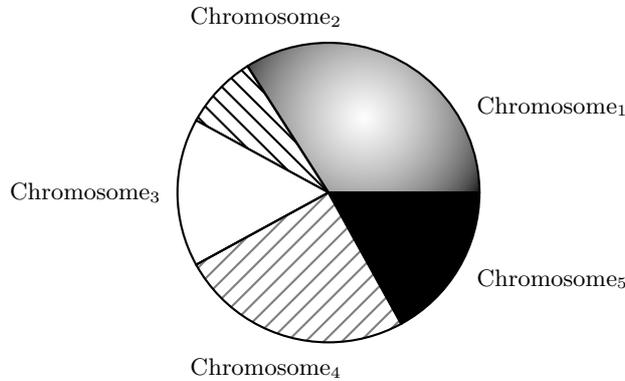


Fig. 1.3. Roulette wheel selection

Elitism

When creating new population by crossover and mutation, we have a big chance that we will lose the best chromosome. Elitism is name of the method that first copies the best chromosome (or a few best chromosomes) to new population. The rest is done in classical way. Elitism can very rapidly increase performance of GA, because it prevents losing the best-found solution.

Genetic Operators

Crossover and mutation are two basic operators of GA. Performance of GA very much depends on the genetic operators. Type and implementation of operators depends on encoding and also on the problem. There are many ways how to do crossover and mutation. In this section we will demonstrate some of the popular methods with some examples and suggestions how to do it for different encoding schemes.

Crossover. It selects genes from parent chromosomes and creates a new offspring. The simplest way to do this is to choose randomly some crossover point and everything before this point is copied from the first parent and then everything after a crossover point is copied from the second parent. A single point crossover is illustrated as follows (| is the crossover point):

Chromosome A:	11111 00100110110
Chromosome B:	10011 11000011110
Offspring A:	11111 11000011110
Offspring B:	10011 00100110110

As illustrated in Fig. 1.4, there are several crossover techniques. In a uniform crossover bits are randomly copied from the first or from the second

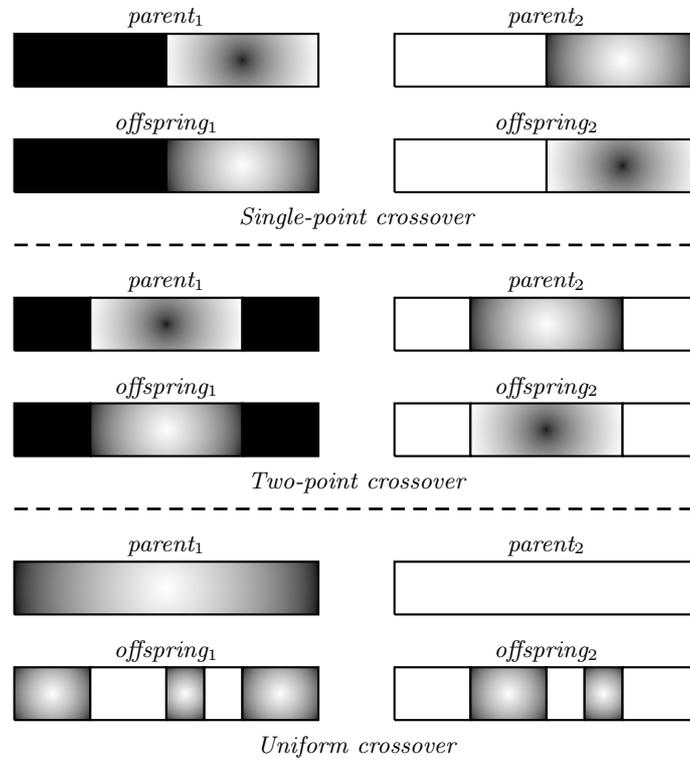


Fig. 1.4. Types of crossover operators

parent. Specific crossover made for a specific problem can improve the GA performance.

Mutation. After crossover operation, mutation takes place. Mutation changes randomly the new offspring. For binary encoding mutation is performed by changing a few randomly chosen bits from 1 to 0 or from 0 to 1. Mutation depends on the encoding as well as the crossover. For example when we are encoding permutations, mutation could be exchanging two genes. A simple mutation operation is illustrated as follows:

Chromosome A: 1101111000011110
 Chromosome B: 1101100100110110

Offspring A: 1100111000011110
 Offspring B: 1101101100110110

For many optimization problems there may be multiple, equal, or unequal optimal solutions. Sometimes a simple GA cannot maintain stable populations at different optima of such functions. In the case of unequal optimal solutions, the population invariably converges to the global optimum. *Niching* helps to maintain subpopulations near global and local optima. A niche is viewed as an organism's environment and a species as a collection of organisms with similar features. Niching helps to maintain subpopulations near global and local optima by introducing a controlled competition among different solutions near every local optimal region. Niching is achieved by a sharing function, which creates subdivisions of the environment by degrading an organism's fitness proportional to the number of other members in its neighbourhood. The amount of sharing contributed by each individual into its neighbour is determined by their proximity in the decoded parameter space (phenotypic sharing) based on a distance measure.

1.3 Evolution Strategies

Evolution Strategy (ES) was developed by Rechenberg [17] at Technical University, Berlin. ES tend to be used for empirical experiments that are difficult to model mathematically. The system to be optimized is actually constructed and ES is used to find the optimal parameter settings. Evolution strategies merely concentrate on translating the fundamental mechanisms of biological evolution for technical optimization problems. The parameters to be optimized are often represented by a vector of real numbers (object parameters – op). Another vector of real numbers defines the strategy parameters (sp) which controls the mutation of the objective parameters. Both object and strategic parameters form the data-structure for a single individual. A population P of n individuals could be described as $P = (c_1, c_2, \dots, c_{n-1}, c_n)$, where the i th chromosome c_i is defined as $c_i = (op, sp)$ with $op = (o_1, o_2, \dots, o_{n-1}, o_n)$ and $sp = (s_1, s_2, \dots, s_{n-1}, s_n)$.

1.3.1 Mutation in Evolution Strategies

The mutation operator is defined as component wise addition of normal distributed random numbers. Both the objective parameters and the strategy parameters of the chromosome are mutated. A mutant's object-parameters vector is calculated as $o_p(mut) = o_p + N_0(s_p)$, where $N_0(s_i)$ is the Gaussian distribution of mean-value 0 and standard deviation s_i . Usually the strategy parameters mutation step size is done by adapting the standard deviation s_i . For instance, this may be done by $s_p(mut) = (s_1 \times A_1, s_2 \times A_2, \dots, s_{n-1} \times A_{n-1}, s_n \times A_n)$, where A_i is randomly chosen from α or $1/\alpha$ depending on the

value of equally distributed random variable E of $[0,1]$ with $A_i = \alpha$ if $E < 0.5$ and $A_i = 1/\alpha$ if $E \geq 0.5$. The parameter α is usually referred to as *strategy parameters adaptation value*.

1.3.2 Crossover (Recombination) in Evolution Strategies

For two chromosomes $c_1 = (o_p(c_1), s_p(c_1))$ and $c_2 = (o_p(c_2), s_p(c_2))$ the crossover operator is defined $R(c_1, c_2) = c = (o_p, s_p)$ with $o_p(i) = (o_p(c_1), i|o_p(c_2), i)$ and $s_p(i) = (s_p(c_1), i|s_p(c_2), i)$. By defining $o_p(i)$ and $s_p(i) = (x|y)$ a value is randomly assigned for either x or y (50% selection probability for x and y).

1.3.3 Controlling the Evolution

Let P be the number of parents in generation 1 and let C be the number of children in generation i . There are basically four different types of evolution strategies: $P, C, P+C, P/R, C$ and $P/R+C$ as discussed below. They mainly differ in how the parents for the next generation are selected and the usage of crossover operators.

P, C Strategy

The P parents produce C children using mutation. Fitness values are calculated for each of the C children and the best P children become next generation parents. The best individuals of C children are sorted by their fitness value and the first P individuals are selected to be next generation parents ($C \geq P$).

P + C Strategy

The P parents produce C children using mutation. Fitness values are calculated for each of the C children and the best P individuals of both parents and children become next generation parents. Children and parents are sorted by their fitness value and the first P individuals are selected to be next generation parents.

P/R, C Strategy

The P parents produce C children using mutation and crossover. Fitness values are calculated for each of the C children and the best P children become next generation parents. The best individuals of C children are sorted by their fitness value and the first P individuals are selected to be next generation parents ($C \geq P$). Except the usage of crossover operator this is exactly the same as P, C strategy.

***P/R + C* Strategy**

The P parents produce C children using mutation and recombination. Fitness values are calculated for each of the C children and the best P individuals of both parents and children become next generation parents. Children and parents are sorted by their fitness value and the first P individuals are selected to be next generation parents. Except the usage of crossover operator this is exactly the same as $P + C$ strategy.

1.4 Evolutionary Programming

Fogel, Owens and Walsh's book [5] is the landmark publication for Evolutionary Programming (EP). In the book, Finite state automata are evolved to predict symbol strings generated from Markov processes and non-stationary time series. The basic evolutionary programming method involves the following steps:

1. Choose an initial population (possible solutions at random). The number of solutions in a population is highly relevant to the speed of optimization, but no definite answers are available as to how many solutions are appropriate (other than > 1) and how many solutions are just wasteful.
2. New offspring's are created by mutation. Each offspring solution is assessed by computing its fitness. Typically, a stochastic tournament is held to determine N solutions to be retained for the population of solutions. It should be noted that evolutionary programming method typically does not use any crossover as a genetic operator.

When comparing evolutionary programming to genetic algorithm, one can identify the following differences:

1. GA is implemented by having arrays of bits or characters to represent the chromosomes. In EP there are no such restrictions for the representation. In most cases the representation follows from the problem.
2. EP typically uses an adaptive mutation operator in which the severity of mutations is often reduced as the global optimum is approached while GA's use a pre-fixed mutation operator. Among the schemes to adapt the mutation step size, the most widely studied being the "meta-evolutionary" technique in which the variance of the mutation distribution is subject to mutation by a fixed variance mutation operator that evolves along with the solution.

On the other hand, when comparing evolutionary programming to evolution strategies, one can identify the following differences:

1. When implemented to solve real-valued function optimization problems, both typically operate on the real values themselves and use adaptive reproduction operators.

2. EP typically uses stochastic tournament selection while ES typically uses deterministic selection.
3. EP does not use crossover operators while ES (P/R,C and P/R+C strategies) uses crossover. However the effectiveness of the crossover operators depends on the problem at hand.

1.5 Genetic Programming

Genetic Programming (GP) technique provides a framework for automatically creating a working computer program from a high-level problem statement of the problem [11]. Genetic programming achieves this goal of automatic programming by genetically breeding a population of computer programs using the principles of Darwinian natural selection and biologically inspired operations. The operations include most of the techniques discussed in the previous sections. The main difference between genetic programming and genetic algorithms is the representation of the solution. Genetic programming creates computer programs in the LISP or scheme computer languages as the solution. LISP is an acronym for LISt Processor and was developed by John McCarthy in the late 1950s [8]. Unlike most languages, LISP is usually used as an interpreted language. This means that, unlike compiled languages, an interpreter can process and respond directly to programs written in LISP. The main reason for choosing LISP to implement GP is due to the advantage of having the programs and data have the same structure, which could provide easy means for manipulation and evaluation.

Genetic programming is the extension of evolutionary learning into the space of computer programs. In GP the individual population members are not fixed length character strings that encode possible solutions to the problem at hand, they are programs that, when executed, are the candidate solutions to the problem. These programs are expressed in genetic programming as parse trees, rather than as lines of code. For example, the simple program “ $a + b * f(4, a, c)$ ” would be represented as shown in Fig. 1.5. The terminal and function sets are also important components of genetic programming. The terminal and function sets are the alphabet of the programs to be made. The terminal set consists of the variables (example, a, b and c in Fig. 1.5) and constants (example, 4 in Fig. 1.5).

The most common way of writing down a function with two arguments is the infix notation. That is, the two arguments are connected with the operation symbol between them as $a + b$ or $a * b$. A different method is the prefix notation. Here the operation symbol is written down first, followed by its required arguments as $+ab$ or $*ab$. While this may be a bit more difficult or just unusual for human eyes, it opens some advantages for computational uses. The computer language LISP uses symbolic expressions (or S-expressions) composed in prefix notation. Then a simple S-expression could be (*operator, argument*) where *operator* is the name of a function and

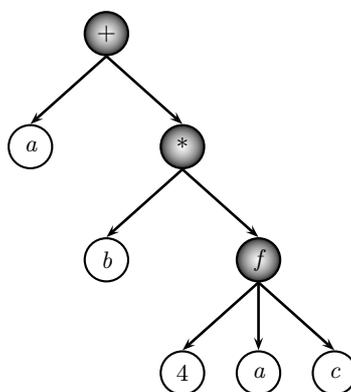


Fig. 1.5. A simple tree structure of GP

argument can be either a constant or a variable or either another symbolic expression as $(operator, argument(operator, argument)(operator, argument))$. Generally speaking, GP procedure could be summarized as follows:

- Generate an initial population of random compositions of the functions and terminals of the problem;
- Compute the fitness values of each individual in the population ;
- Using some selection strategy and suitable reproduction operators produce two offspring;
- Procedure is iterated until the required solution is found or the termination conditions have reached (specified number of generations).

1.5.1 Computer Program Encoding

A parse tree is a structure that grasps the interpretation of a computer program. Functions are written down as nodes, their arguments as leaves. A subtree is the part of a tree that is under an inner node of this tree. If this tree is cut out from its parent, the inner node becomes a root node and the subtree is a valid tree of its own.

There is a close relationship between these parse trees and S-expression; in fact these trees are just another way of writing down expressions. While functions will be the nodes of the trees (or the operators in the S-expressions) and can have other functions as their arguments, the leaves will be formed by terminals, that is symbols that may not be further expanded. Terminals can be variables, constants or specific actions that are to be performed. The process of selecting the functions and terminals that are needed or useful for finding a solution to a given problem is one of the key steps in GP. Evaluation

of these structures is straightforward. Beginning at the root node, the values of all sub-expressions (or subtrees) are computed, descending the tree down to the leaves.

1.5.2 Reproduction of Computer Programs

The creation of an offspring from the crossover operation is accomplished by deleting the crossover fragment of the first parent and then inserting the crossover fragment of the second parent. The second offspring is produced in a symmetric manner. A simple crossover operation is illustrated in Fig. 1.6. In GP the crossover operation is implemented by taking randomly selected sub trees in the individuals and exchanging them.

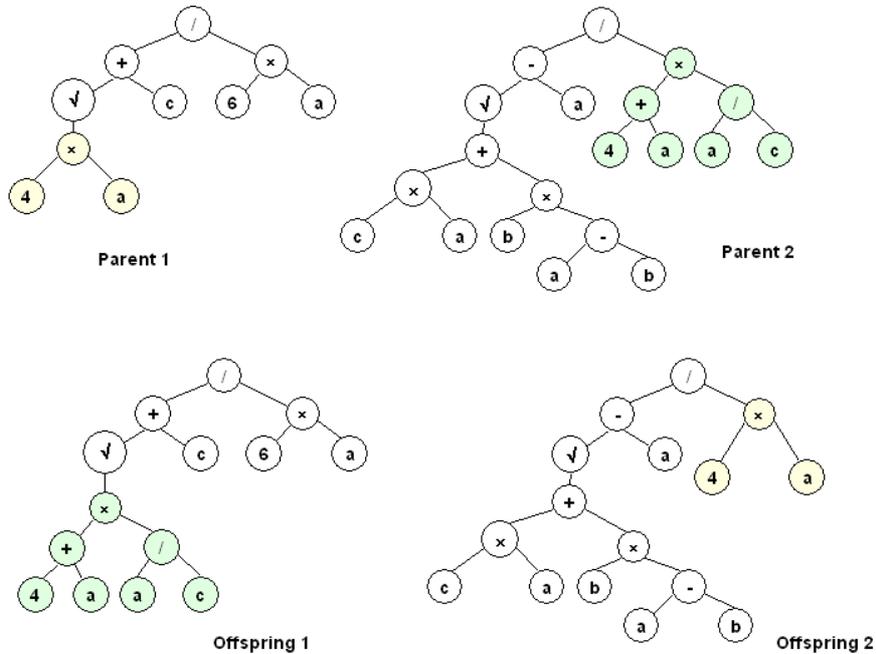


Fig. 1.6. Illustration of crossover operator

Mutation is another important feature of genetic programming. Two types of mutations are commonly used. The simplest type is to replace a function or a terminal by a function or a terminal respectively. In the second kind an entire subtree can replace another subtree. Fig. 1.7 explains the concept of mutation.

GP requires data structures that are easy to handle and evaluate and robust to structural manipulations. These are among the reasons why the class

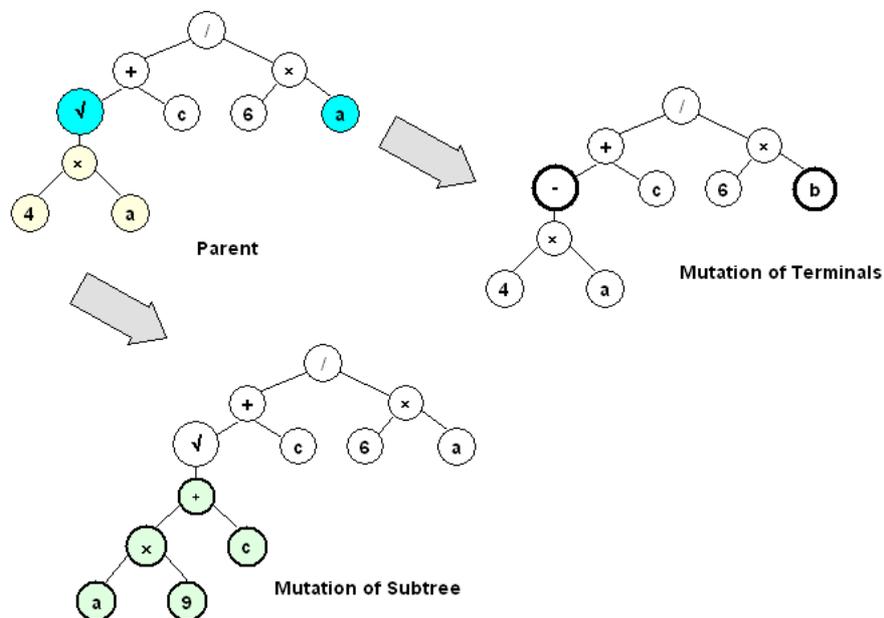


Fig. 1.7. Illustration of mutation operator in GP

of S-expressions was chosen to implement GP. The set of functions and terminals that will be used in a specific problem has to be chosen carefully. If the set of functions is not powerful enough, a solution may be very complex or not to be found at all. Like in any evolutionary computation technique, the generation of first population of individuals is important for successful implementation of GP. Some of the other factors that influence the performance of the algorithm are the size of the population, percentage of individuals that participate in the crossover/mutation, maximum depth for the initial individuals and the maximum allowed depth for the generated offspring etc. Some specific advantages of genetic programming are that no analytical knowledge is needed and still could get accurate results. GP approach does scale with the problem size. GP does impose restrictions on how the structure of solutions should be formulated.

1.6 Variants of Genetic Programming

Several variants of GP could be seen in the literature. Some of them are Linear Genetic Programming (LGP), Gene Expression Programming (GEP), Multi Expression Programming (MEP), Cartesian Genetic Programming (CGP), Traceless Genetic Programming (TGP) and Genetic Algorithm for Deriving Software (GADS).

1.6.1 Linear Genetic Programming

Linear genetic programming is a variant of the GP technique that acts on linear genomes [3]. Its main characteristics in comparison to tree-based GP lies in that the evolvable units are not the expressions of a functional programming language (like LISP), but the programs of an imperative language (like C/C++). This can tremendously hasten the evolution process as, no matter how an individual is initially represented, finally it always has to be represented as a piece of machine code, as fitness evaluation requires physical execution of the individuals. The basic unit of evolution here is a native machine code instruction that runs on the floating-point processor unit (FPU). Since different instructions may have different sizes, here instructions are clubbed up together to form instruction blocks of 32 bits each. The instruction blocks hold one or more native machine code instructions, depending on the sizes of the instructions. A crossover point can occur only between instructions and is prohibited from occurring within an instruction. However the mutation operation does not have any such restriction. LGP uses a specific linear representation of computer programs. A LGP individual is represented by a variable length sequence of simple C language instructions. Instructions operate on one or two indexed variables (registers) r , or on constants c from predefined sets.

An important LGP parameter is the number of registers used by a chromosome. The number of registers is usually equal to the number of attributes of the problem. If the problem has only one attribute, it is impossible to obtain a complex expression such as the quartic polynomial. In that case we have to use several supplementary registers. The number of supplementary registers depends on the complexity of the expression being discovered. An inappropriate choice can have disastrous effects on the program being evolved. LGP uses a modified steady-state algorithm. The initial population is randomly generated. The settings of various linear genetic programming system parameters are of utmost importance for successful performance of the system. The population space has been subdivided into multiple subpopulation or demes. Migration of individuals among the subpopulations causes evolution of the entire population. It helps to maintain diversity in the population, as migration is restricted among the demes. Moreover, the tendency towards a bad local minimum in one deme can be countered by other demes with better search directions. The various LGP search parameters are the mutation frequency, crossover frequency and the reproduction frequency: The crossover operator acts by exchanging sequences of instructions between two tournament winners. Steady state genetic programming approach was used to manage the memory more effectively

1.6.2 Gene Expression Programming (GEP)

The individuals of gene expression programming are encoded in linear chromosomes which are expressed or translated into expression trees (branched

entities)[4]. Thus, in GEP, the genotype (the linear chromosomes) and the phenotype (the expression trees) are different entities (both structurally and functionally) that, nevertheless, work together forming an indivisible whole. In contrast to its analogous cellular gene expression, GEP is rather simple. The main players in GEP are only two: the chromosomes and the Expression Trees (ETs), being the latter the expression of the genetic information encoded in the chromosomes. As in nature, the process of information decoding is called translation. And this translation implies obviously a kind of code and a set of rules. The genetic code is very simple: a one-to-one relationship between the symbols of the chromosome and the functions or terminals they represent. The rules are also very simple: they determine the spatial organization of the functions and terminals in the ETs and the type of interaction between sub-ETs. GEP uses linear chromosomes that store expressions in breadth-first form. A GEP gene is a string of terminal and function symbols. GEP genes are composed of a *head* and a *tail*. The head contains both function and terminal symbols. The tail may contain terminal symbols only. For each problem the head length (denoted h) is chosen by the user. The tail length (denoted by t) is evaluated by:

$$t = (n - 1)h + 1, \quad (1.3)$$

where n is the number of arguments of the function with more arguments.

GEP genes may be linked by a function symbol in order to obtain a fully functional chromosome. GEP uses mutation, recombination and transposition. GEP uses a generational algorithm. The initial population is randomly generated. The following steps are repeated until a termination criterion is reached: A fixed number of the best individuals enter the next generation (elitism). The mating pool is filled by using binary tournament selection. The individuals from the mating pool are randomly paired and recombined. Two offspring are obtained by recombining two parents. The offspring are mutated and they enter the next generation.

1.6.3 Multi Expression Programming

A GP chromosome generally encodes a single expression (computer program). A Multi Expression Programming (MEP) chromosome encodes several expressions [14]. The best of the encoded solution is chosen to represent the chromosome. The MEP chromosome has some advantages over the single-expression chromosome especially when the complexity of the target expression is not known. This feature also acts as a provider of variable-length expressions. MEP genes are represented by substrings of a variable length. The number of genes per chromosome is constant. This number defines the length of the chromosome. Each gene encodes a terminal or a function symbol. A gene that encodes a function includes pointers towards the function arguments. Function arguments always have indices of lower values than the position of the function itself in the chromosome.

The proposed representation ensures that no cycle arises while the chromosome is decoded (phenotypically transcribed). According to the proposed representation scheme, the first symbol of the chromosome must be a terminal symbol. In this way, only syntactically correct programs (MEP individuals) are obtained. The maximum number of symbols in MEP chromosome is given by the formula:

$$\text{Number_of_Symbols} = (n + 1) \times (\text{Number_of_Genes} - 1) + 1, \quad (1.4)$$

where n is the number of arguments of the function with the greatest number of arguments. The translation of a MEP chromosome into a computer program represents the phenotypic transcription of the MEP chromosomes. Phenotypic translation is obtained by parsing the chromosome top-down. A terminal symbol specifies a simple expression. A function symbol specifies a complex expression obtained by connecting the operands specified by the argument positions with the current function symbol.

Due to its multi expression representation, each MEP chromosome may be viewed as a forest of trees rather than as a single tree, which is the case of Genetic Programming.

1.6.4 Cartesian Genetic Programming

Cartesian Genetic Programming (CGP) uses a network of nodes (indexed graph) to achieve an input to output mapping [13]. Each node consists of a number of inputs, these being used as parameters in a determined mathematical or logical function to create the node output. The functionality and connectivity of the nodes are stored as a string of numbers (the genotype) and evolved to achieve the optimum mapping. The genotype is then mapped to an indexed graph that can be executed as a program.

In CGP there are very large number of genotypes that map to identical genotypes due to the presence of a large amount of redundancy. Firstly there is node redundancy that is caused by genes associated with nodes that are not part of the connected graph representing the program. Another form of redundancy in CGP, also present in all other forms of GP is, functional redundancy.

1.6.5 Traceless Genetic Programming (TGP)

The main difference between Traceless Genetic Programming and GP is that TGP does not explicitly store the evolved computer programs [15]. TGP is useful when the trace (the way in which the results are obtained) between the input and output is not important. TGP uses two genetic operators: crossover and insertion. The insertion operator is useful when the population contains individuals representing very complex expressions that cannot improve the search.

1.6.6 Grammatical Evolution

Grammatical evolution [18] is a grammar-based, linear genome system. In grammatical evolution, the Backus Naur Form (BNF) specification of a language is used to describe the output produced by the system (a compilable code fragment). Different BNF grammars can be used to produce code automatically in any language. The genotype is a string of eight-bit binary numbers generated at random and treated as integer values from 0 to 255. The phenotype is a running computer program generated by a genotype-phenotype mapping process. The genotype-phenotype mapping in grammatical evolution is deterministic because each individual is always mapped to the same phenotype. In grammatical evolution, standard genetic algorithms are applied to the different genotypes in a population using the typical crossover and mutation operators.

1.6.7 Genetic Algorithm for Deriving Software (GADS)

Genetic algorithm for deriving software is a GP technique where the genotype is distinct from the phenotype [16]. The GADS genotype is a list of integers representing productions in a syntax. This is used to generate the phenotype, which is a program in the language defined by the syntax. Syntactically invalid phenotypes cannot be generated, though there may be phenotypes with residual nonterminals.

1.7 Summary

This chapter presented the biological motivation and fundamental aspects of evolutionary algorithms and its constituents, namely genetic algorithm, evolution strategies, evolutionary programming and genetic programming. Most popular variants of genetic programming are introduced. Important advantages of evolutionary computation while compared to classical optimization techniques are also discussed.

References

1. Abraham, A., Evolutionary Computation, In: Handbook for Measurement, Systems Design, Peter Sydenham and Richard Thorn (Eds.), John Wiley and Sons Ltd., London, ISBN 0-470-02143-8, pp. 920–931, 2005. 2
2. Bäck, T., Evolutionary algorithms in theory and practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms, Oxford University Press, New York, 1996.
3. Banzhaf, W., Nordin, P., Keller, E. R., Francone, F. D., Genetic Programming : An Introduction on The Automatic Evolution of Computer Programs and its Applications, Morgan Kaufmann Publishers, Inc., 1998. 16

4. Ferreira, C., Gene Expression Programming: A new adaptive algorithm for solving problems - Complex Systems, Vol. 13, No. 2, pp. 87–129, 2001. 17
5. Fogel, L.J., Owens, A.J. and Walsh, M.J., Artificial Intelligence Through Simulated Evolution, John Wiley & Sons Inc. USA, 1966. 2, 11
6. Fogel, D. B. (1999) Evolutionary Computation: Toward a New Philosophy of Machine Intelligence. IEEE Press, Piscataway, NJ, Second edition, 1999. 3
7. Goldberg, D. E., Genetic Algorithms in search, optimization, and machine learning, Reading: Addison-Wesley Publishing Corporation Inc., 1989.
8. History of Lisp, <http://www-formal.stanford.edu/jmc/history/lisp.html>, 2004. 12
9. Holland, J. Adaptation in Natural and Artificial Systems, Ann Harbor: University of Michigan Press, 1975. 2, 5
10. Jang, J.S.R., Sun, C.T. and Mizutani, E., Neuro-Fuzzy and Soft Computing: A Computational Approach to Learning and Machine Intelligence, Prentice Hall Inc, USA, 1997.
11. Koza. J. R., Genetic Programming. The MIT Press, Cambridge, Massachusetts, 1992. 2, 12
12. Michalewicz, Z., Genetic Algorithms + Data Structures = Evolution Programs, Berlin: Springer-Verlag, 1992.
13. Miller, J. F. Thomson, P., Cartesian Genetic Programming, Proceedings of the European Conference on Genetic Programming, Lecture Notes In Computer Science, Vol. 1802 pp. 121–132, 2000. 18
14. Oltean M. and Grosan C., Evolving Evolutionary Algorithms using Multi Expression Programming. Proceedings of The 7th. European Conference on Artificial Life, Dortmund, Germany, pp. 651–658, 2003. 17
15. Oltean, M., Solving Even-Parity Problems using Traceless Genetic Programming, IEEE Congress on Evolutionary Computation, Portland, G. Greenwood, et. al (Eds.), IEEE Press, pp. 1813–1819, 2004. 18
16. Paterson, N. R. and Livesey, M., Distinguishing Genotype and Phenotype in Genetic Programming, Late Breaking Papers at the Genetic Programming 1996, J. R. Koza (Ed.), pp. 141–150, 1996. 19
17. Rechenberg, I., Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution, Stuttgart: Fromman-Holzboog, 1973. 2, 9
18. Ryan, C., Collins, J. J. and O’Neill, M., Grammatical Evolution: Evolving Programs for an Arbitrary Language, Proceedings of the First European Workshop on Genetic Programming (EuroGP’98), Lecture Notes in Computer Science 1391, pp. 83–95, 1998. 19
19. Schwefel, H.P., Numerische Optimierung von Computermodellen mittels der Evolutionsstrategie, Basel: Birkhaeuser, 1977. 2
20. Törn A. and Zilinskas A., Global Optimization, Lecture Notes in Computer Science, Vol. 350, Springer-Verlag, Berlin, 1989.