Evolving Computer Programs for Knowledge Discovery

Crina Grosan[1] and Ajith Abraham[2]

[1]*Department of Computer Science,*
Babes-Bolyai University Cluj-Napoca, Romania, cgrosan@cs.ubbcluj.ro

[2]*IITA Professorship Program,*
School of Computer Science and Engineering,
Chung-Ang University, Korea, ajith.abraham@ieee.org

Abstract: Genetic Programming (GP) offers practical advantages to the researcher facing difficult optimization problems. These advantages are multifold, including the simplicity of the approach, its robust response to changing circumstance, its flexibility, and many other facets. GP can be applied to problems where heuristic solutions are not available or generally lead to unsatisfactory results. As a result, GP has recently received increased interest, particularly with regard to the manner in which they may be applied for practical problem solving. This paper illustrates how some GP variants could be used for knowledge discovery and function approximation.

1. Introduction and Biological Motivation

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 [9]. 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 operation.

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. A population of candidate solutions (for the optimization task to be solved) is initialized. New solutions are created by applying reproduction operators (crossover and/or mutation). The fitness (how good the solutions are) of the resulting solutions is evaluated and suitable selection strategy is then applied to determine which solutions will be maintained into the next generation [1].

2. 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 [10]. 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 GP and GA is the representation of the solution. GP creates computer programs in the LISP or scheme computer languages as the solution. LISP is an acronym for LISt Processor. 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. 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. Example, the simple program '$a + b * c$' would be represented as shown in Figure 1. 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 and constants of the programs (example, *A*,*B* and *C* in Figure 1).

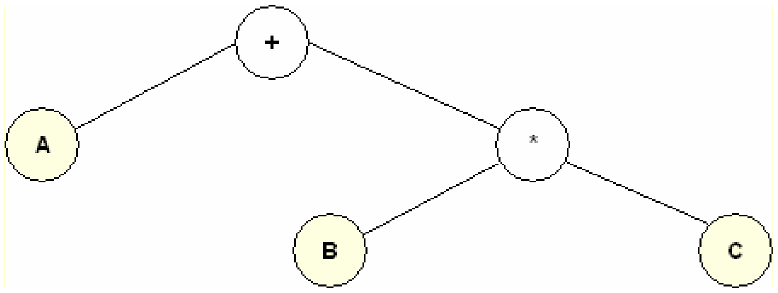

Figure 1. Tree structure of GP

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 follows:
$A + B$
A different method is the prefix notation. Here the operation symbol is written down first, followed by its required arguments:
$+AB$
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 *argument* can be either a constant or a variable or either another symbolic expression as shown below:
(*operator, argument* (*operator, argument*) (*operator, argument*))

2.1. Genetic Programming Basics

A parse tree is a structure that develops 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 as illustrated in Figure 2. 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.

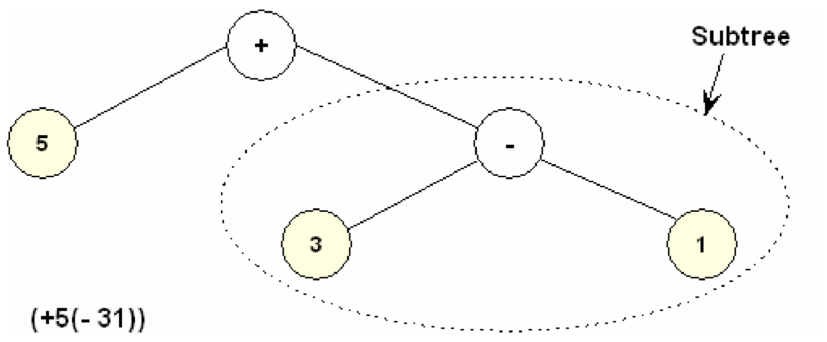

Figure 2. Architecture of a GP sub-tree

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 are 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. 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 offsprings
- Procedure is iterated until the required solution is found or the termination conditions have reached (specified number of generations).

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 Figure 3. In GP the crossover operation is implemented by taking randomly selected sub trees in the individuals and exchanging them.

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. Figure 4 explains the concepts of mutation.

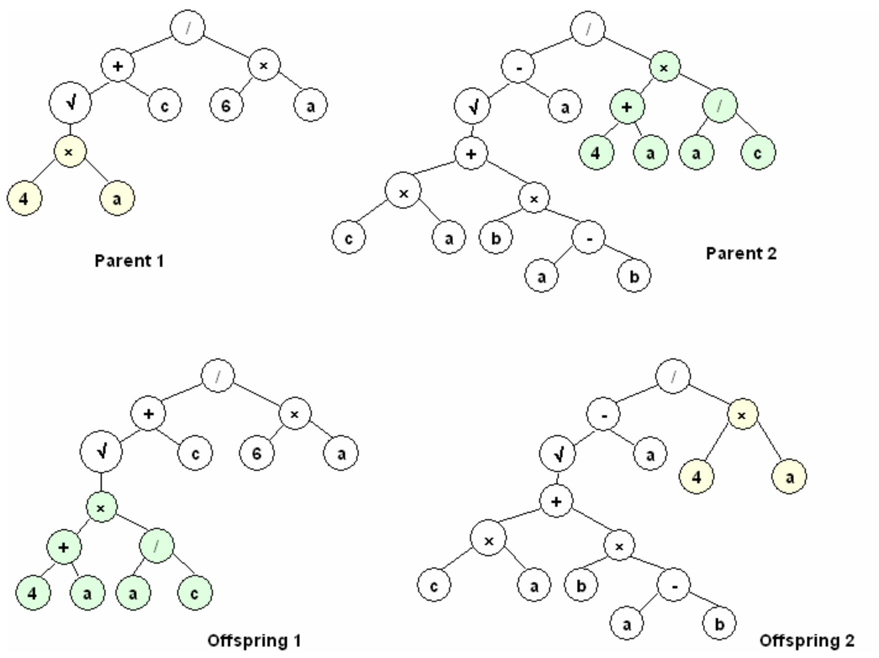

Figure 3. GP recombination operation

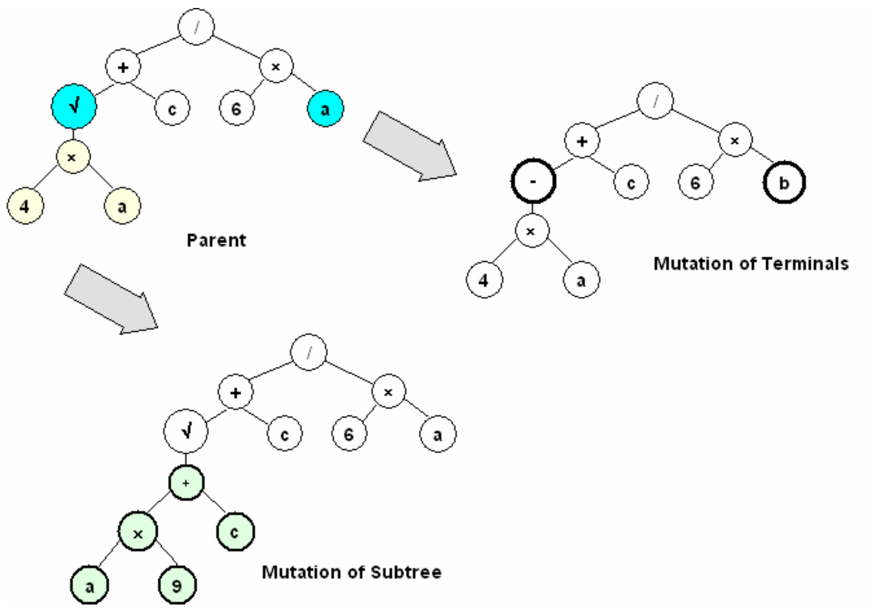

Figure 4. GP mutation operation

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

2.2. 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).

2.2.1. Linear Genetic Programming

Linear genetic programming is a variant of the GP technique that acts on linear genomes [2]. 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.

2.2.2. 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) [3]. 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:

$$t = (n - 1)h + 1$$

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.

2.2.3. Multi Expression Programming

A GP chromosome generally encodes a single expression (computer program). A Multi Expression Programming (MEP) chromosome encodes several expressions. The best of the encoded solution is chosen to represent the chromosome [4]. 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 transcripted). 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. An example of chromosome using the sets $F = \{+, *\}$ and $T = \{a, b, c, d\}$ is given below:
1: *a*
2: *b*
3: + 1, 2
4: *c*
5: *d*
6: + 4, 5
7: * 3, 6
The maximum number of symbols in MEP chromosome is given by the formula:
Number_of_Symbols = (*n* + 1) * (*Number_of_Genes* – 1) + 1,
where *n* is the number of arguments of the function with the greatest number of arguments. The maximum number of effective symbols is achieved when each gene (excepting the first one) encodes a function symbol with the highest number of arguments. The minimum number of effective symbols is equal to the number of genes and it is achieved when all genes encode terminal symbols only.
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.
For instance, genes 1, 2, 4 and 5 in the previous example encode simple expressions formed by a single terminal symbol. These expressions are:
$E_1 = a$,
$E_2 = b$,
$E_4 = c$,
$E_5 = d$,
Gene 3 indicates the operation + on the operands located at positions 1 and 2 of the chromosome. Therefore gene 3 encodes the expression: $E_3 = a + b$. Gene 6 indicates the operation + on the operands located at positions 4 and 5. Therefore gene 6 encodes the expression: $E_6 = c + d$. Gene 7 indicates the operation * on the operands located at position 3 and 6. Therefore gene 7 encodes the expression: $E_7 = (a + b) * (c + d)$. E_7 is the expression encoded by the whole chromosome.
There is neither practical nor theoretical evidence that one of these expressions is better than the others. This is why each MEP chromosome is allowed to encode a number of expressions equal to the chromosome length (number of genes). The chromosome described above encodes the following expressions:
$E_1 = a$,
$E_2 = b$,
$E_3 = a + b$,
$E_4 = c$,
$E_5 = d$,
$E_6 = c + d$,
$E_7 = (a + b) * (c + d)$.
The value of these expressions may be computed by reading the chromosome top down.
Partial results are computed by dynamic programming and are stored in a conventional manner.

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.

The chromosome fitness is usually defined as the fitness of the best expression encoded by that chromosome. For instance, if we want to solve symbolic regression problems, the fitness of each sub-expression E_i may be computed using the formula:

$$f(E_i) = \sum_{k=1}^{n} \left| o_{k,i} - w_k \right|,$$

where $o_{k,i}$ is the result obtained by the expression E_i for the fitness case k and w_k is the targeted result for the fitness case k. In this case the fitness needs to be minimized. The fitness of an individual is set to be equal to the lowest fitness of the expressions encoded in the chromosome:
When we have to deal with other problems, we compute the fitness of each sub-expression encoded in the MEP chromosome. Thus, the fitness of the entire individual is supplied by the fitness of the best expression encoded in that chromosome.

2.2.4. Cartesian Genetic Programming

Cartesian Genetic Programming (CGP) uses a network of nodes (indexed graph) to achieve an input to output mapping [6]. 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.

2.2.5. Traceless Genetic Programming (TGP)

The main difference between Traceless Genetic Programming and GP is that TGP does not explicitly store the evolved computer programs [7]. 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.

2.2.6. Grammatical Evolution

Grammatical evolution 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) [5]. 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.

2.2.7. Mathematical Expressions Evolver (MEXE)

Mathematical Expressions Evolver (MEXE) uses a string of operands and operators (on alternative positions) for representing solutions [13]. The odd positions of the chromosome consist of operands and the even positions consist of operators. The chromosome length is constant during the search process. The genetic operators used are crossover and mutation. By mutation, an operand can be replaced with another operand from the considered set and an operator can be replaced with another operator from the considered operators set. Each of the sub-expressions encoded in a MEXE chromosome is considered as a potential problem solution. For instance the chromosome

$C = x + y * y - x / y$

encodes the expressions:

$E_1 = x$;
$E_2 = x+y$;
$E_3 = x+y*y$;
$E_4 = x+y*y-x$;
$E_5 = x+y*y-x / y$;

The value of these expressions may be computed by reading the chromosome from left to right. Partial results are computed by dynamic programming and stored in a conventional manner. The chromosome fitness is usually defined as the fitness of the best expression detected in that chromosome.

2.2.8. Genetic Algorithm for Deriving Software (GADS)

Genetic algorithm for deriving software is a GP technique where the genotype is distinct from the phenotype. The GADS genotype is a list of integers representing productions in a syntax [8]. 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 non-terminals.

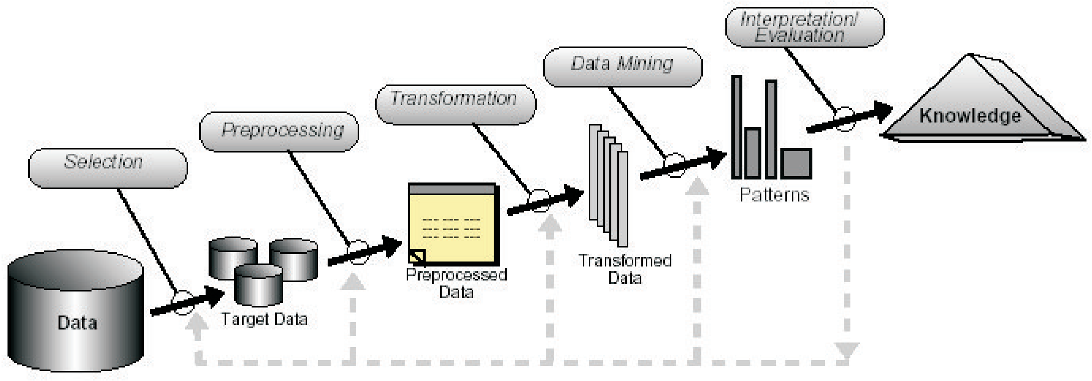

Figure 5. Steps of the knowledge discovery process.

3. Knowledge Discovery

Historically the notion of finding useful patterns in data has been given a variety of names including data mining, knowledge extraction, information discovery, and data pattern processing. The term data mining has been mostly used by statisticians, data analysts, and the management information systems (MIS) communities. The term knowledge discovery in databases (KDD) refers to the overall process of discovering knowledge from data, while data mining refers to a particular step of this process. Data mining is the application of specific algorithms for extracting patterns from data. The additional steps in the KDD process, such as data selection, data cleaning, incorporating appropriate prior knowledge, and proper interpretation of the results are essential to ensure that useful knowledge is derived form the data [14]. KDD has evolved from the intersection of research fields such as machine learning, pattern recognition, databases, statistics, artificial intelligence, and more recently it gets new inspiration from computational intelligence.

Steps of Knowledge Discovery

We broadly outline some of the basic steps of knowledge discovery as depicted in Figure 5 taken from [15].

1. Developing and understanding the application domain, the relevant prior knowledge, and identifying the goal of the KDD process.
2. Creating training/target data set.
3. Data cleaning and preprocessing: basic operations such as the removal of noise, handling missing data fields.
4. Data reduction and projection: finding useful features to represent the data depending the goal of the task. Using dimensionality reduction or transformation methods to reduce the effective number of variables under consideration or to find invariant representation of data.
5. Matching the goals of the KDD process to a particular data mining method: Although the boundaries between prediction and description are not sharp, the distinction is

useful for understanding the overall discovery goal. The goals of knowledge discovery are achieved via the following data mining methods:

- Clustering: Identification of a finite set of categories or clusters to describe the data. Closely related to clustering is the method of probability density estimation. Clustering quantizes the available input-output data to get a set of prototypes and use the obtained prototypes (signatures, templates, etc., and many writers refer to as codebook) and use the prototypes as model parameters.
- Summation: finding a compact description for subset of data, e.g. the derivation of summary for association of rules and the use of multivariate visualization techniques.
- Dependency modeling: finding a model which describes significant dependencies between variables (e.g. learning of belief networks).
- Regression: learning a function which maps a data item to a real-valued prediction variable and the discovery of functional relationships between variables.
- Classification: learning a function that maps (classifies) a data item into one of several predefined classes.
- Change and Deviation Detection: Discovering the most significant changes in the data from previously measured or normative values.

Model evaluation criteria: qualitative statements or fit functions of how well a particular pattern (a model and its parameters) meet the goals of the KDD process. For example, predictive models can often judged by the empirical prediction accuracy on some test set. Descriptive models can be evaluated along the dimensions of predictive accuracy, novelty, utility, and understandability of the fitted model.

4. Knowledge Discovery Using GP Techniques

We will consider several applications for illustrating the large area of problems which can be approached using GP techniques.

4.1 GP Applied for Function Approximation

We consider the following function: $x^4+2x^3-10x^2+6x$. The training set contains 20 instances between 0 and 1. Parameters used by GP are:

- Number of generations: 100
- Population size: 100
- Crossover frequency: (%) 80
- Mutation frequency: (%) 40
- Maximum depth for new individuals: 6
- Maximum depth for individuals after crossover: 20
- Maximum depth for new subtrees in mutants: 4
- Functions used: +, -, *, /.

The quality of the approximation is given by the sum of the absolute differences of the target function values and the approximation function values at the sample points. The evolution of fitness function during the search process is depicted in Figure 6. Comparison between the function evolved by GP and the target function is presented in Figure 7.

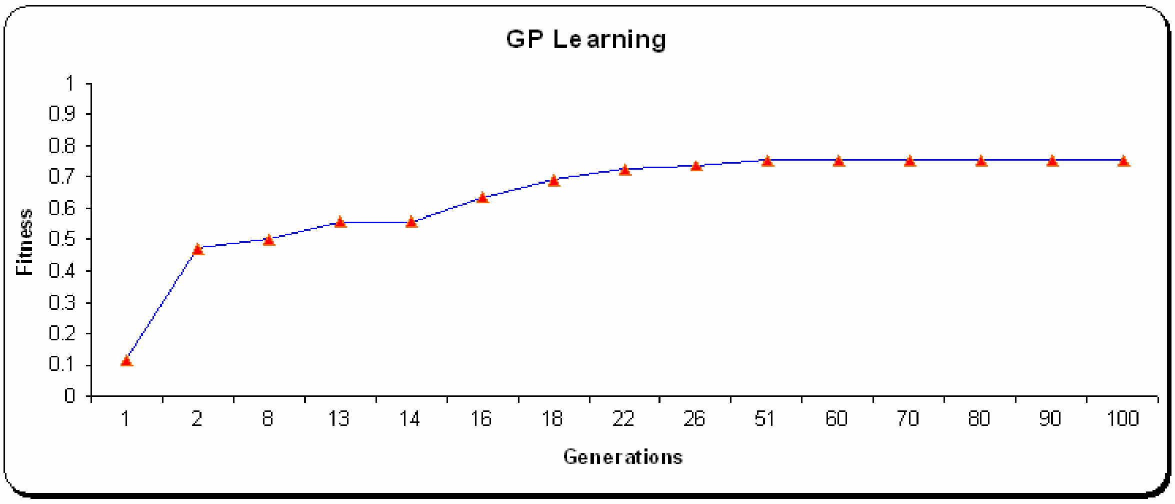

Figure 6. Relationship between fitness value and number of generations for GP.

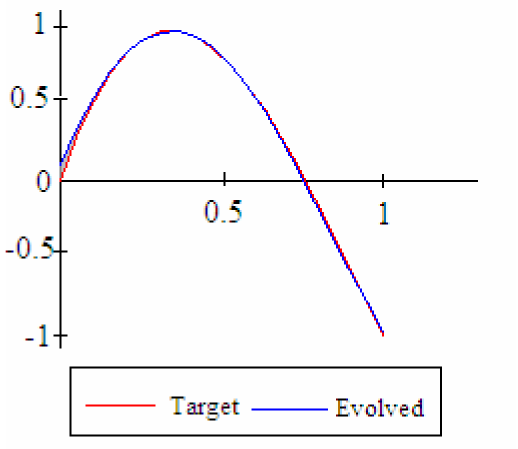

Figure 7. Function evolved by GP.

4.2 LGP Applied for Classification of Diabetics Patients

The data is extracted from the "Pima Indians Diabetes Database", archived by National Institute of Diabetes and Digestive and Kidney Diseases, USA. The diagnostic, binary-valued variable (0 for no and 1 for yes) investigated is whether the patient shows signs of diabetes according to World Health Organization criteria (i.e., if the 2 hour post-load plasma glucose was at least 200 mg/dl at any survey examination or if found during routine medical care). From the 250 data samples, 24 datasets were randomly extracted to represent the test set. Each data set comprises of 8 input variables (all numeric-valued) with the following details:

- Number of times pregnant

- Plasma glucose concentration a 2 hours in an oral glucose tolerance test
- Diastolic blood pressure (mm Hg)
- Triceps skin fold thickness (mm)
- 2-Hour serum insulin (mu U/ml)
- Body mass index (weight in kg/(height in m)2)
- Diabetes pedigree function
- Age (years)
- Output class variable (0 or 1) is interpreted as "tested positive for diabetes").

LGP method was used to build the diabetic patients classifier. 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. After a trial and error approach, the following parameter settings were used for the experiments:

- Population size: 500
- No of tournaments: 50,000
- Mutation frequency: 95%
- Crossover frequency: 50%
- Number of demes: 10
- Maximum program size: 256
- Target subset size: 100

The experiments were repeated 10 times and the worst results are reported. In Figures 7-9 the average of fitness values obtained for train data set (Figure 7), the values of the best individual obtained for training (Figure 8) and the values of the best individual obtained for test data (Figure 9) are depicted.

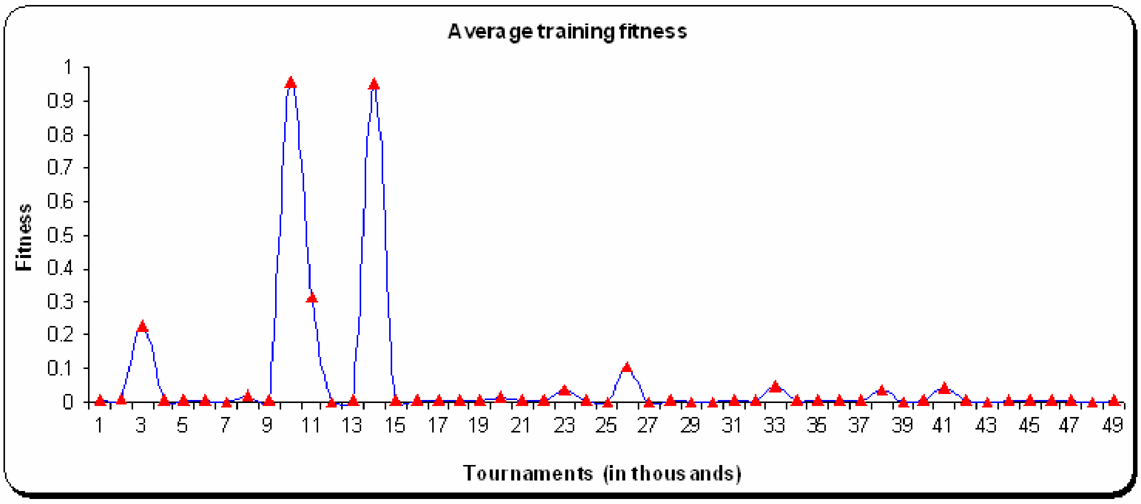

Figure 7. Average training fitness during 50,000 tournaments

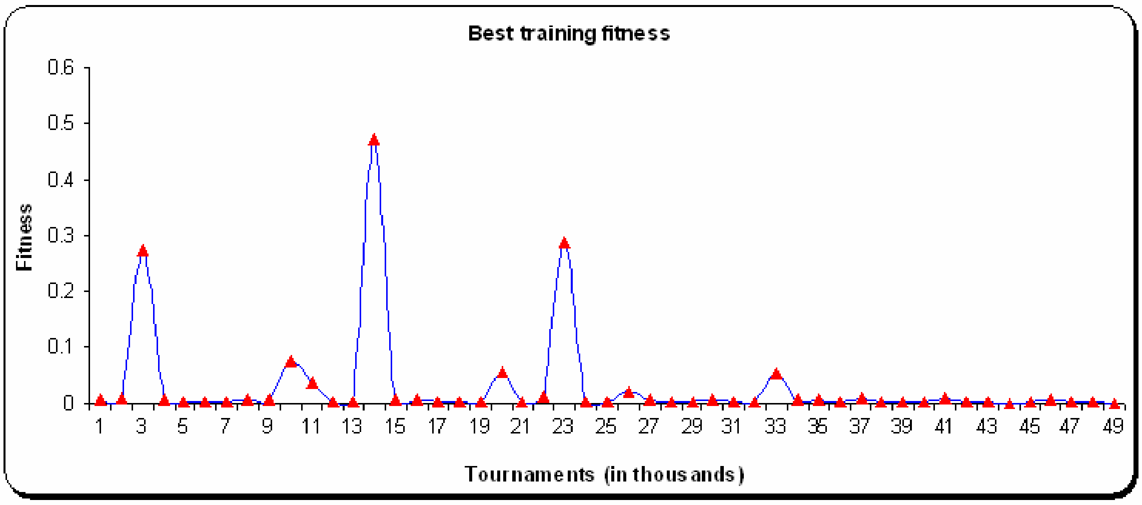

Figure 8. Best training fitness during 50,000 tournaments

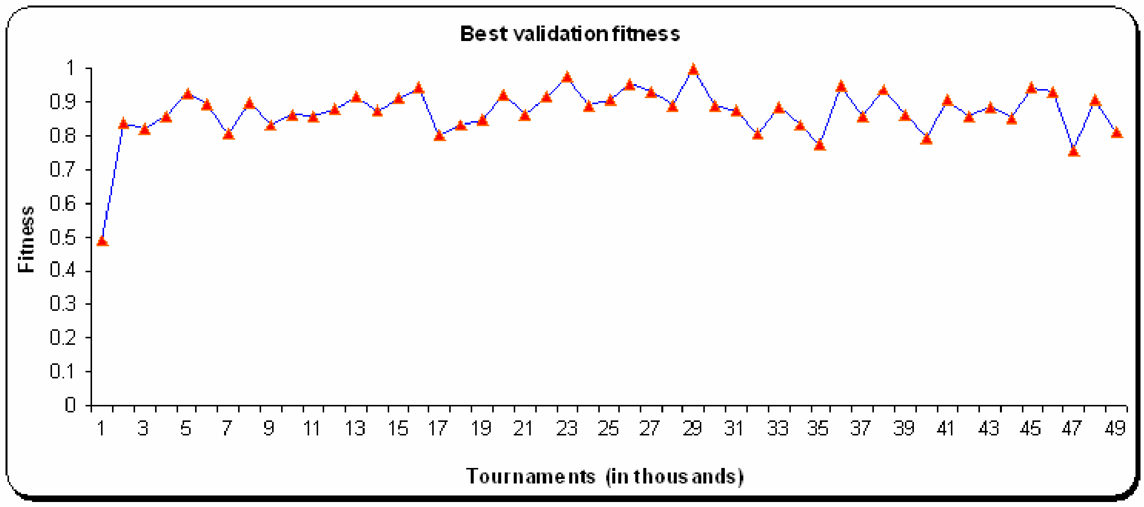

Figure 9. Best validation fitness during 50,000 tournaments

The classification accuracy obtained by LGP for boths training and test data set are presented in Table 1.

Classification accuracy	
Training data	**Test data**
80.88 %	79%

Table 1. Classification accuracy for training and test data.

4.3. MEP Applied for Mackey-Glass Chaotic Time Series

The Mackey-Glass differential equation [10] is a chaotic time series for some values of the parameters *x(0)* and *τ*.

$$\frac{dx(t)}{dt} = \frac{0.2x(t-\tau)}{1+x^{10}(t-\tau)} - 0.1x(t).$$

We used the value *x(t-18), x(t-12), x(t-6), x(t)* to predict *x(t+6)*. Fourth order Runge-Kutta method was used to generate 1000 data series. The time step used in the method is 0.1 and initial condition were $x(0)$=1.2, τ=17, $x(t)$=0 for t<0. First 500 data sets were used for training and remaining data for testing.

Parameters used by MEP are:

- Population size: 200
- Crossover probability: 0.9
- Number of mutations / chromosome: 4
- Number of generations: 150
- Functions used: +, - , *, /, sin, cos, sqrt, ln, lg, log2, min, max, abs.

Our goal is to optimize the Root Mean Squared Error (RMSE) and Correlation Coefficient (CC):

$$RMSE = \sqrt{\sum_{i=1}^{N} \left| P_{actual,i} - P_{predicted,i} \right|}, \qquad (a)$$

$$CC = \frac{\sum_{i=1}^{N} P_{predicted,i}}{\sum_{i=1}^{N} P_{actual,i}}, \qquad (b)$$

The value of RMSE obtained by MEP is: 0.064862 and the value obtained for CC is 0.9995. Results obtained by MEP are depicted in Figure 10.

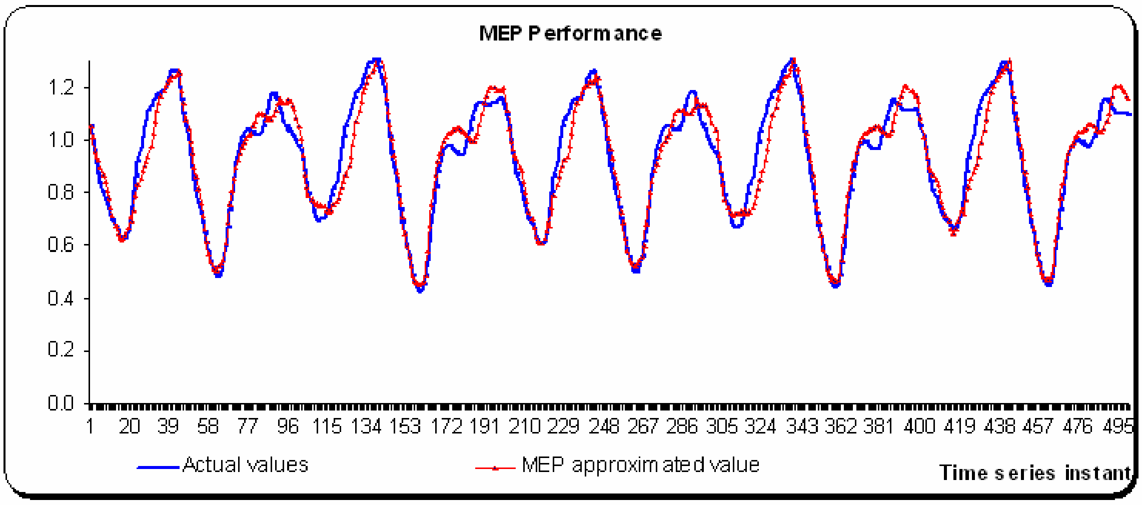

Figure 10. Comparison between values obtained by MEP and target values for Mackey-Glass Chaotic Time Series

4.4. MEXE Applied for Function Detection

We consider the quartic function $x^4 + x^3 + x^2 + x$. The training set for quartic function contains 21 fitness cases randomly generated over the interval [0, 1]. Parameters used by MEXE are:

- Population size: 50
- Number of generations: 50
- Mutation probability: 0.3
- Crossover probability: 0.9.

The MEXE success rate obtained for different chromosomes length is depicted in Figure 11. For greater lengths of the chromosomes the success rate is 100%.

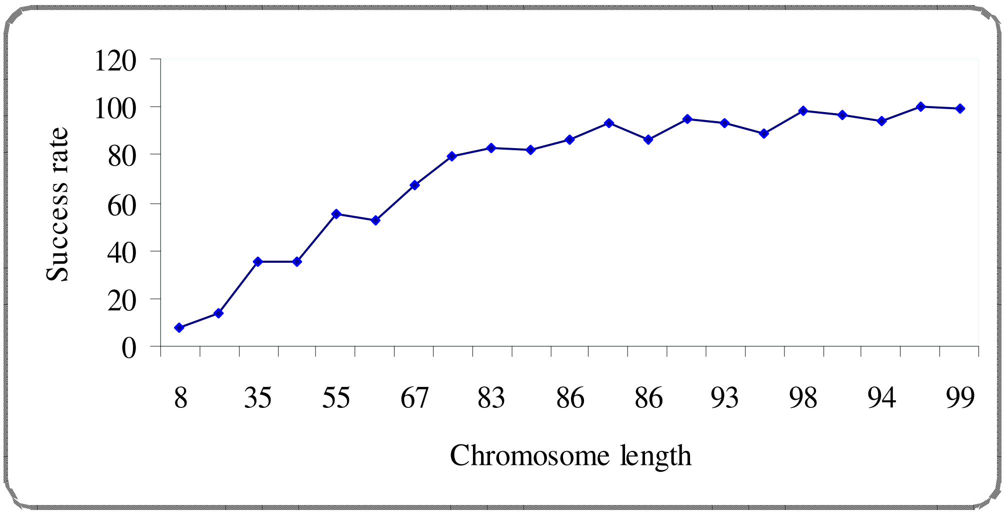

Figure 11. Comparison of different success rates obtained by MEXE considering different chromosomes lengths

4.5 GEP for Gas Furnace Time Series Data

This time series was used to predict the CO_2 (carbon dioxide) concentration *y(t+1)* [11]. In a gas furnace system, air and methane are combined to form a mixture of gases containing CO_2. Air fed into the gas furnace is kept constant, while the methane feed rate *u(t)* can be varied in any desired manner. After that, the resulting CO_2 concentration y(t) is measured in the exhaust gases at the outlet of the furnace. Data is represented as [*u(t), y(t), y(t+1)*] The time series consists of 292 pairs of observation and 50% of data was used for training and remaining for testing.

Our goal is to optimize several error measures: Root Mean Squared Error (RMSE) (equation *(a)*, Section 4.3) Correlation Coefficient (CC) (equation *(b)* Section 4.3) and Maximum Absolute Percentage Error given by:

$$MAE = \max\left(\frac{\left| P_{actual,\, i} - P_{predicted,\, i} \right|}{P_{predicted,\, i}} \times 100 \right)$$

Results obtained by GEP for RMSE, CC and MAE for different population sizes are depicted in Figures 12 (for train data set) and 13 (for test data set).

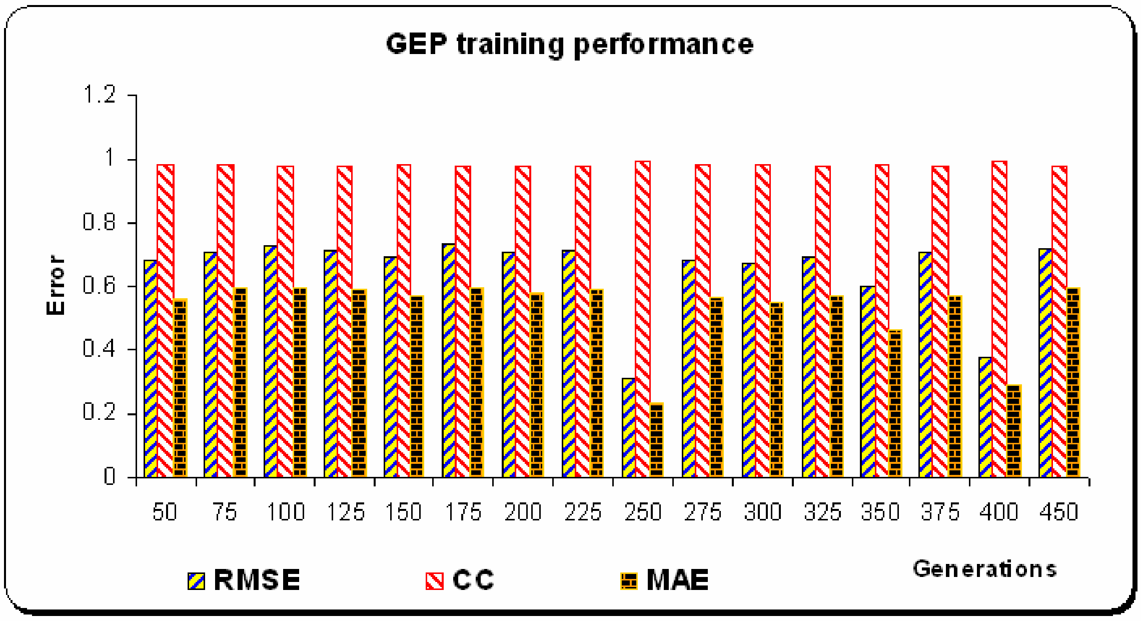

Figure 13. Values of RMSE, CC and MAE obtained by GEP for different population sizes of train data set.

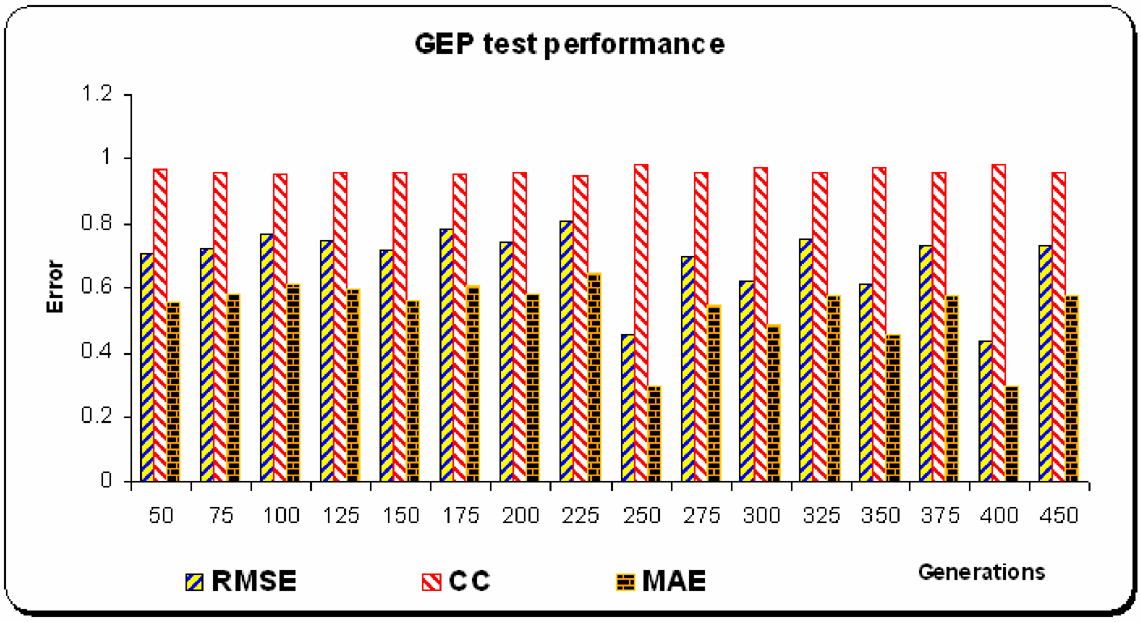

Figure 14. Values of RMSE, CC and MAE obtained by GEP considerent different population sizes of test data set.

5. Conclusions

This paper illustrated the basic principles of genetic programming and it variants. We further attempted to formulate some of the GP models for practical knowledge discovery purposes. As evident from the illustrations and empirical results, GP techniques are a promising tool for solving complex classification problems, prediction, functions approximation and symbolic regression problems.

References

[1] Abraham, A., Evolutionary Computation, 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] 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.

[3] Ferreira, C., Gene Expression Programming: A new adaptive algorithm for solving problems - Complex Systems, Vol. 13, No. 2,pp. 87-129, 2001.

[4] 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.

[5] C. Ryan, J. J. Collins, and M. O'Neill, 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.

[6] Miller, JF Thomson, P., Proceedings of the European Conference on Genetic Programming, Lecture Notes In Computer Science; Vol. 1802 pp. 121 - 132, 2000.

[7] Oltean, M., Solving Even-Parity Problems using Traceless Genetic Programming, IEEE Congress on Evolutionary Computation, Portland, 19-23 June, edited by G. Greenwood et. al. IEEE Press, pp. 1813-1819, 2004.

[8] Paterson, NR and Livesey, M., Distinguishing Genotype and Phenotype in Genetic Programming, Late Breaking Papers at the Genetic Programming 1996, Koza JR (Ed.), pp. 141-150,1996.

[9] Fogel, D. B. (1999) Evolutionary Computation: Toward a New Philosophy of Machine Intelligence. IEEE Press, Piscataway, NJ, Second edition, 1999.

[10] Koza. J.R. (1992) Genetic Programming. The MIT Press, Cambridge, Massachusetts.

[11] Mackey MC and Glass L (1977), Oscillation and Chaos in Physiological Control Systems, Science Vol 197, pp.287-289.

[12] Box G E P and Jenkins G M, *Time Series Analysis, Forecasting and Control*, San Francisco: Holden Day, 1970.

[13] Grosan, C., Evolving mathematical expressions using Genetic Algorithms. Genetic and Evolutionary Computation Conference (GECCO), T. Riopka et al. (Eds), Springer-Verlag Germany, Seattle, USA, 2004.

[14] Abonyi J., Feil, B. and Abraham, A., 'Computational Intelligence in Data Mining', Informatica: An International Journal of Computing and Informatics, Vol. 29, No. 1, pp. 3-12, 2005.

[15] U. Fayyad, G. Piatestku-Shapio, P. Smyth, Knowledge discovery and data mining: Towards a unifying framework, in: Advances in Knowledge Discovery and Data Mining, AAAI/MIT Press, 1994.