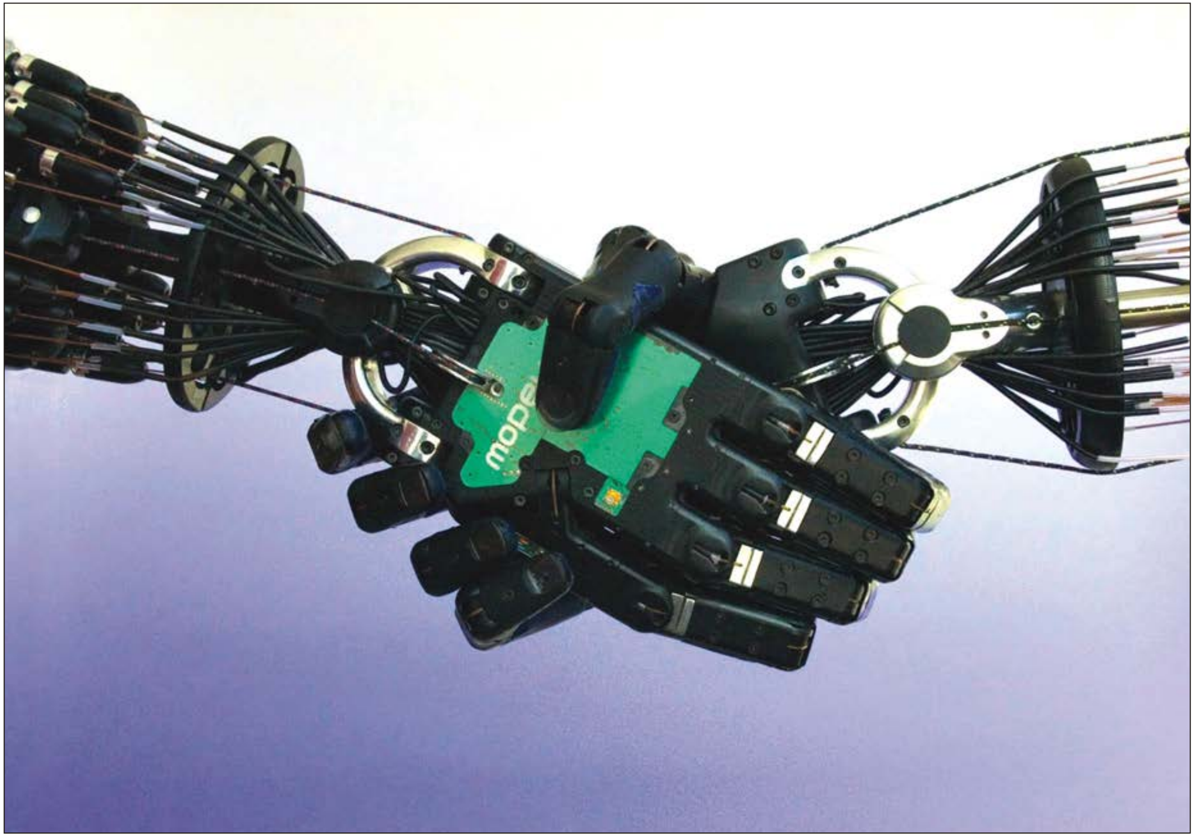

Journal of Artificial Intelligence

ISSN 1994-5450

science alert

ANSInet
an open access publisher
http://ansinet.com

Journal of Artificial Intelligence 3 (2): 67-72, 2010
ISSN 1994-5450
© 2010 Asian Network for Scientific Information

GADS and Reusability

Y. Al-Bastaki and W. Awad
Information Technology College, University of Bahrain, Bahrain

Abstract: Genetic programming is a domain-independent method that genetically breeds population of computer programs to solve problems. Genetic programming is considered to be a machine learning technique used to optimize a population of computer programs according to a fitness landscape determined by a program's ability to perform a given computational task. There are a number of representation methods to illustrate these programs, such as LISP expressions and integer lists. This study investigated the effectiveness of genetic programming in solving the symbolic regression problem where, the population programs are expressed as integer sequences rather than lisp expressions. This study also introduced the concept of reusable program to genetic algorithm for developing software.

Key words: Genetic programming, reusability, GADS

INTRODUCTION

One of the central challenges of computer science is to get a computer to solve a problem without explicitly programming it. Genetic Programming (GP) (Koza, 1992) is a domain-independent problem-solving approach in which computer programs are evolved to solve, or approximately solve, problems. It is based on the Darwinian principle of reproduction and survival of the fittest and analogs of naturally occurring genetic operations such as crossover and mutation. The Darwinian principle of reproduction and survival of the fittest and the genetic operation of crossover are used to create a new offspring population of individual computer programs from the current population of programs. The GP has been used in wide area of applications, one of them is to solve the symbolic regression problem. Symbolic regression can be viewed as the process of shaping an equation from a given set of points. For example, the equation $y = x^2$ is regressed from the pairs (1, 1), (2, 4), (3, 9), (4, 16), etc. A genotype is examined by contrasting the results. It generates with the results generated by the goal equation. The differences are summed and the lower this final sum, the better the fitness of the individual. Equations and other forms of Genetic Programs are represented in tree structures. In a program tree, the interior nodes contain operators (+,-,*) or functions, anything that can take parameters. The leaves contain the terminals: identifiers, strings, numbers or anything that has a value. Figure 1 shows the trees representing two different equations.

The number of children that any given node has is dependant upon the number of parameters that the associated function or operator takes. In the case of addition, multiplication and division operators there are two children for each node. The absolute value operator takes one value as input. The constant and variable nodes have no children since, there is no way for them to evaluate children.

Genetic programming is an application of Genetic Algorithm (GA). The GA is a search algorithm based on the mechanics of natural selection and natural genetic. The GA was

Corresponding Author: Y. Al-Bastaki, Information Technology College, University of Bahrain, Bahrain

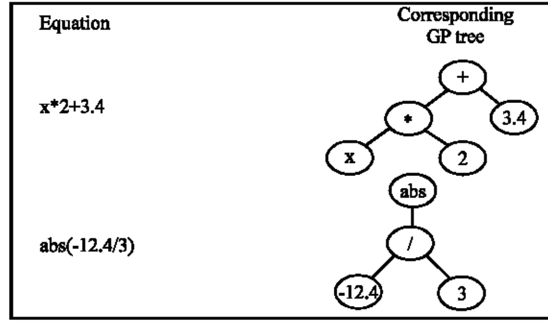

Fig. 1: Genetic programming representation

suggested by Holland (1975) in the seventies. Over the last 20 years, it has been used to solve a wide range of search, optimization and machine learning problems (Goldberg, 1989).

Koza (1992, 1995) and Koza *et al.* (1999) uses LISP expression to represent the papulation programs. However, there is another representation method (Paterson and Livesey, 1996, 1997) introduced new method for program representation. He suggested a different approach using Backus Nour Form (BNF) definition which is a notation for expressing the grammar of a language in the form of production rules. The BNF grammars consist of Non-terminals, Terminals, Start symbol and Production rules so {N, T, S, P}. There has been a Genetic Algorithm for Developing Software by Peterson (GADS). He used a fixed length chromosome which encodes production rules, where the genotype (genetic search space element i.e., chromosome) is distinct from the phenotype (solutions space element i.e., program). The GADS genotype is a list of integers which, when input by a suitable generator, causes that generator to output the program that is the corresponding phenotype. The mapping from genotype to phenotype is called ontogenic mapping. The genotype is operated on by the genetic operators (crossover, mutation and so on) in the usual range of ways available to GA.

In our previous work (Al-Bastki and Awad, 2003) GADS has been used to solve the symbolic regression problem, in which a simple set of syntax rules has been used with only one function can be defined.

In this study, GADS is applied to solve the symbolic regression problem, with the introduction of a new concept which is as function reusability, such that a number of functions can be automatically defined in each genetic papulation program, with any number of parameters. Furthermore, a new operator is used which is altering function architecture, by changing the number of function parameters.

Many efforts have been made to use genetic algorithms to solve symbolic regression problems. One of the problems that plagues most of the efforts is finding a way to efficiently mutate and cross-breed symbolic expressions so that the resulting expressions have a valid mathematical syntax. One approach to this problem, is to perform a mutation, check the result and then try a different random mutation until a syntactically valid expression is generated. Obviously, this can be a time consuming process for complex expressions. A second approach is to limit what type of mutations can be performed for example, only exchanging complete sub-expressions. The problem with this approach is that if limited mutations are used, the evolution process is hindered and it may take a large number of generations to find a solution, or it may be completely unable to find the optimal solution. In this study constrained GADS is presented, which is inspired from the concept of strongly-typed GP (Haynes *et al.*, 1995).

Symbolic Regression

In ordinary mathematical regression, the procedure is given the form of the function to be fitted to the data. This could be a linear function for linear regression or a general mathematical function for nonlinear regression. The regression procedure computes the optimal values of parameters for the function to make the function fit a data set as well as possible, but the regression procedure does not alter the form of the function. For example, a linear regression problem with two variables has the form:

$$y = a+b*x$$

where, x is the independent variable, y is the dependent variable and a and b are parameters whose values are to be computed by the regression algorithm.

This type of procedure is classified as parametric regression, because the goal is to estimate parameters for a function whose form is known (or assumed). With nonparametric regression the form of the function is not known in advance and it is the goal of the procedure to find a function that will fit the data. So we are looking for f (•) that will best fit:

$$y = f(x_1,x_2,\ldots,x_n)$$

where, y is the dependent variable and there are n independent x variables.

There are many possible forms of nonparametric functions-neural networks and decision trees are types of nonparametric functions. Symbolic regression is a subset of nonparametric regression that restricts the functions to be mathematical or logical expressions.

GADS with Reusable Functions

Automatically Defined Functions (ADF) has been introduced by Koza (1995), where GP will automatically and dynamically evolve a combined structure containing ADF and a calling program capable of calling the ADF. In this study, ADF is a technique used with GADS.

When, an ADF is encountered in a genotype, a random number is generated which represents the number of function parameters, then the body of the function is constructed. Therefore, the phenotypes consists of: the root (ADFi, where i is the identification number of the function which is incremented whenever a new ADF is introduced), the list of parameters (the number of these parameters is generated randomly) and function definition. As example, consider the following function and terminal sets, F and T:

$$T = (X, n)$$

$$F = (+,\ -,\ *,\ \%)$$

The syntax rules (BNF) used is presented in Table 1.

In order to generate a well formed expression, constrained GADS is used. Thus, the syntax of the programs should be preserved during the initial papulation generation and by the genetic operation used to modify the population. Therefore, the generation of a gene in the chromosomes is simply based on some constraints (according the sysnax rules defined in Table 1), such that: if a_1, a_2, …., a_L is the genotype, the selection of gene a_{L+1} is not randomly, instead, it is dependent on the gene a_L. Therefore, each gene has a number of allowed genes to appear after it.

Table 1: The syntax rules

Syntax rules	Rule No.
<Sexp> : = < Input>	0
<Sexp> : = < Application>	1
<Inpute> : = X	2
<Input> : = n	3
<Application> : = n Call P	4
<Application> : = <Sexp> + <Sexp>	5
<Application> : = <Sexp> - <Sexp>	6
<Application> : = <Sexp> * <Sexp>	7
<Application> : = <Sexp> % <Sexp>	8
<Application> : = n ADF <Sexp>	9

n represent an integer number, X is a variable, P is the list of parameters and n Call P represents calling the function, ADFn with the parameter list P, in which P should be a list of constants (integers). Also, n of rule (9) represents the number of parameters (number of variables in the function definition)

The process of generating an ADF in a genotype of the initial population can be performed as follows:

- Generate a random number n which the number of function parameters
- Define the body of the function recursively, where the primitives that composed the function is either a function, or an integer number in the range 1....n

We need to mention here that wherever an ADFi is defined in a chromosome, it is replaced by the function i call P, i.e., rule number 4 and the function definition is stored in a separate array. Furthermore, it is not allowed to include the gene (4) in the chromosome unless an ADF is found in this chromosome.

Genetic Operations

The crossover operator must be implemented so that two chromosomes (genotypes), that are syntactically correct, produce two offsprings that are also syntactically correct. In this study, we need the following steps to perform the modified brood crossover operator: Pick two parents from the population.

- Select a gene randomly from the first parent
- Select a gene randomly from the second parent
- Test that genes for the syntactic constraint satisfaction
- If it conforms to syntax rules then exchange the genes, otherwise, select another gene from the second parent until the correct gene is found
- Steps 2-5 is repeated NB times to generate 2*NB offspring
- Evaluate each of the children for fitness. Select the best two, they are considered as the children of the parents. The remaining of the offspring are discarded

The mutation operator involves the selection of a gene randomly from a genotype and then generate a gene randomly to replace the selected gene. Check the left and right neighbors of that gene, if it satisfies the syntactic rules, then replace it, otherwise, select another gene.

In this method, the genotypes have a variable length. Thus, lengths of genotypes in the population are selected randomly and the max-length must be specified beforehand by the user and depends on the problem.

Furthermore, another operator may be applied, which is altering the AFD definition be changing the number of parameters. This can be performed as follows: when, an ADF is

selected for this operation, a random number is generated to be the new p of the ADF. Then, change the body of the ADF by replacing the integer numbers that represent the parameters by new numbers generated randomly.

Experimental Work

The proposed modified GADS has been implemented to solve the symbolic regression problem using C++ programming language. Each chromosome has been implemented as a structure of the fields: one-dimensional array of integers (chromosome), two-dimensional array to store ADF definition (if any), chromosome length and chromosome fitness value.

The genetic parameters used are: population size = 100, crossover probability = 1, mutation probability = 0.05 and the changing parameters number operation probability = 0.01.

For example, the expression to be evolved is:

$$X^4 + X^3 + X^2 + X$$

Using the syntax rules of Table 1, after 22 generations the following genotype has been obtained as

17150217020215180202170202

The corresponding phenotype is:

(x+(x*x))*((x%x)+(x*x))

In another run, after two generations, the following genotype has been obtained:

17150217020219150317020 2

The corresponding phenotype is:

(x+(x*x))*((1) call 1))

where ADF1 (1+(x*x)). Figure 2 shows the parse tree of this expression.

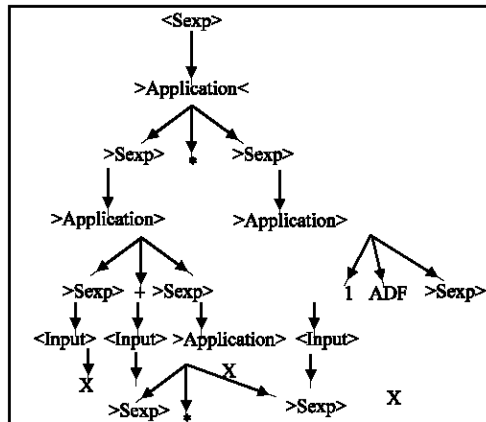

Fig. 2: Parse of the second expression

CONCLUSIONS

In this study, GADS has been used where, the structure under adaption is a population of strings, while in GP, the structure is a population of programs (LISP expressions). Thus, GADS uses the GA engine and works on simpler structure. Thus, we expect an improvement in the efficiency in terms of time and storage space, in addition to simplify the implementation of genetic operations such as crossover and mutation.

Furthermore, the concept of reusability has been introduced to GADS which, can improve the efficiency especially for complex problems. The function reusability has been introduced by using ADF with the call function, in addition to altering architecture operator.

Of course, to ensure the syntactically correct program, strongly-typed GP has been used. Such that, some constraints have been enforced in the generation of the initial population and applying the genetic operations.

The symbolic regression problem is considered here and we have observed that the number of generations needed to find the correct solution is minimized comparable with Koza (1992) work.

REFERENCES

Al-Bastki, Y. and W. Awad, 2003. New development in genetic algortihm for developing software. Stud. Informatics Control, 12: 277-284.

Goldberg, D.E., 1989. Genetic Algorithms in Search, Optimization and Machine Learning. 1st Edn., Addison-Wesley, New York, USA., ISBN: 0201157675.

Haynes, T., R. Wainwright, S. Sen and D. Schoenefeld, 1995. Strongly typed GP in evolving cooperation strategies. Proceedings of the 6th International Conference on Genetic Algorithms, July 15-19, Morgan Kaufmann, USA., pp: 271-278.

Holland, J.H., 1975. Adaptive in natural and artificial systems. Ann Arbor, University of Michigan.

Koza, J.R., 1992. Genetic Programming: On the Programming of Computers by Means of Nature Selection. MIT Press, Cambridge, MA., ISBN: 0-262-11170-5.

Koza, J.R., 1995. Genetic Programming II: Automatic Discovery of Reusable Programs. MIT Press, USA.

Koza, J.R., F.H Bennett III, D. Andre and M.A. Keane, 1999. Genetic Programming III: Darwinian Invention and Problem Solving. 1st Edn., Morgan Kaufmann, USA., ISBN-10: 1558605436. pp: 1154.

Paterson, N.R. and M. Livesey, 1996. Distinguishing Genotype and Phenotype in Genetic Programming. In: Late Breaking Papers at the Genetic Programming, Koza, J.R. (Ed.). Stanford Bookstore, USA., ISBN: 0-18-201031-7. pp: 141-150.

Paterson, N. and M. Livesey, 1997. Evolving Caching Algorithms in C by GP. In: Genetic Programming 1997, Koza, J.R., K. Deb, M. Dorigo, D.B. Fogel, M. Garzon, H. Iba and R.L. Riolo (Eds.). Morgan Kaufmann, USA., pp: 262-267.