Research Journal of Information Technology

ISSN 1815-7432

Research Journal of Information Technology 4 (4): 235-240, 2012
ISSN 1815-7432 / DOI: 10.3923/rjit.2012.235.240

The Evolution of Reusable Programs Using Genetic Algorithm

Yousif Al-Bastaki
Department of Computer Science, Information Technology College, University of Bahrain, Bahrain

ABSTRACT

Genetic programming is an automatic programming technique that is used to evolve computer programs by applying genetic algorithm. There are a number of representation methods to illustrate these programs, such as LISP expressions. This study investigates the effectiveness of genetic programming in solving the symbolic regression problem by using different representation scheme, in which, the population programs are expressed as integer sequences rather than lisp expressions. This approach is called Genetic Algorithm for Developing Software (GADS). Furthermore, this study introduces the concept of reusability to GADS and explains how to evolve reusable programs using GADS. Different architecture altering operations are applied such as function deletion and arguments duplication.

Key words: Genetic algorithm, genetic programming, automatically defined functions

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; Weijie *et al*., 2011) is a domain-independent problem-solving approach in which computer programs are evolved to solve, or approximately solve, problems. It is an application of Genetic Algorithm (GA) (Mosavi, 2011; Mahi and Izabatene, 2011), which is a search algorithm based on the mechanics of natural selection and natural genetic and based on the Darwinian principle of reproduction and survival of the fittest and analogs of naturally occurring genetic operations such as crossover and mutation. GA was suggested by Holland in the seventies (Holland, 1975). Over the last twenty years, it has been used to solve a wide range of search, optimization and machine learning problems (Goldberg, 1989).

GP is adaptive algorithm, where the structure under adaption is a population of chromosomes represent the candidate programs (solutions).

There are different representation methods used to represent the chromosomes, the most common on the Lisp expressions (Koza, 1992, 1995, 1999) uses LISP expressions to represent the population programs. However, there is another representation method, which is proposed by Paterson and Livesey (1996, 1997), They introduced a new method for program representation. He suggested a different approach using Backus Naur Form (BNF) definition which is a notation for expressing the grammar of a language in the form of production rules. 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). This approach is called Genetic Algorithm for Developing Software by Peterson (GADS). The GADS (Al-Bastaki and Awad, 2010) 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.

In this study, we use the representation method of GADS in order to represent the programs. The main objective is to investigate effectiveness of applying different architecture altering operations introduced by Koza (1999) with GADS. Thus, by using the proposed approach, more complex structured programs can be evolved efficiently comparable with traditional GADS and GP.

The symbolic regression problem has been chosen as case study here. GP has been used in wide area of applications (Pugazhenthi and Rajagopalan, 2007), 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.

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 (Gene expression programming, 2008). In this study constrained GADS, which is inspired from the concept of strongly-typed GP (Haynes *et al.*, 1995), with automatically defined functions (ADF) is presented.

GA AND GP

One of the component methodologies of computational intelligence is evolutionary computation. There are number of evolutionary computation techniques, such as GA, Genetic Programming (GP) (Kumarci *et al.*, 2010), Cultured Algorithms and Differential Evolution algorithms. Regardless of the technique used, evolutionary computation applications follow a similar procedure:

- Initialize the population
- Evaluate each individual in the population
- Select individuals
- Produce a new population by applying a number of operations on selected individuals
- loop to step b until some condition is met

GA is a general, probabilistic and adaptive search algorithm. GA is a stochastic search algorithm that is based on the Darwinian principle of survival of the fittest. It works on population of individuals (chromosomes) that represent the candidate solutions.

Gas have been applied to solve complex problems (Christy and Thambidurai, 2006; Chettih *et al.*, 2008) which has been used in a large number of scientific and engineering problems, such as optimization, automatic programming and machine learning.

GP is an application of GA in which the chromosomes are the candidate programs. It used to solve problems by generating the program that can be used to solve this problem. Thus, with GP the structure under adaption is more complex.

GADS WITH ADF

To work with GP, we have to determine the function library and representation scheme, in addition to the fitness function.

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

The representation method used here is variable length chromosomes, where the genes are integer numbers represent the number of the syntax rules. The syntax rules (BNF) used is presented in Table 1 and the function and terminal sets, F and T are:

$$T = \{ X, n \}$$
$$F = \{+ , - , * , \% \}$$

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 work 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.

In this Table 1, 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).

In order to generate a well formed expression, constrained GADS is used. Thus, The syntax of the programs should be preserved during the initial population 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 syntax 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.

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

In this study, different genetic operations (Wasan, 2008) can be used to modify the populations. The operations used are as follows:

- **The crossover operator:** It must be implemented so that two chromosomes (genotypes) that are syntactically correct to produce two offsprings that are also syntactically correct. The following are the steps needed to perform the modified brood crossover operator:

Step 1: Select two parents from the population
Step 2: Select a gene randomly from the first parent and select a gene randomly from the second parent
Step 3: Test that genes for the syntactic constraint
Step 4: If it is correct then exchange the genes, otherwise, select another gene from the second parent until the correct gene is found
Step 5: Steps 2-5 is repeated NB times to generate 2*NB offspring
Step 6: 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:** It 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

- **The function deletion operator:** It used to delete an ADF. A random ADF is selected randomly
- **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 following fields: one-dimensional array of integers (chromosome), ADF definitions (if any), chromosome length and chromosome fitness value. Where the fitness function used is:

$$\text{Fit}(x) = 1/(\text{ABS}(\text{actualO}-\text{desiredO})+1) \quad (1)$$

where, actualO is the actual output of the chromosome x and desiredO is the desired output.

Table 2: Results of different experiments

	Ex1	Ex2	Ex3	Ex4
Average number of generations	33	10	13	16

The genetic parameters used are: population size = 100, crossover probability = 1, mutation and other operators are of probability = 0.05.

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:

171502170202151802021 70202

The corresponding phenotype is:

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

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

17150217020219150317 0202

The corresponding phenotype is:

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

where, ADF1 (1+(x*x)).

Table 2 presents the results of different experiments listed bellow and these results are the average number of generations needed to find the correct solution (program) which are obtained by executing the proposed method 100 times:

- Ex1: Without ADF
- Ex2: With ADF and the genetic operations used are crossover and mutation only
- Ex3: With ADF and the genetic operations used are crossover, mutation and ADF deletion
- Ex4: With ADF and the genetic operations used are crossover, mutation and changing the number of parameters

It is clear that GADS with ADF gives the best results in term of number of generations and GADS without ADF is inefficient especially for complex problems and expressions.

CONCLUSION

In this study, GADS has been used in which 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. Hence, 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.

The main objective of this study is to introduce the concept of reusability and ADF 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 different altering architecture operators. 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 GADS without ADF. For future work, other genetic operations and applications will be studied.

REFERENCES

Al-Bastaki, Y. and W. Awad, 2010. GADS and reusability. J. Artif. Intell., 3: 67-72.

Chettih, S., M. Khiat and A. Chaker, 2008. Optimal distribution of the reactive power and voltages control in Algerian network using the genetic algorithm method. Inform. Technol. J., 7: 1170-1175.

Christy, A. and P. Thambidurai, 2006. Efficient information extraction using machine learning and classification using genetic and C4.8 algorithms. Inform. Technol. J., 5: 1023-1027.

Goldberg, D.E., 1989. Genetic Algorithms in Search Optimization and Machine Learning. Addison-Wesley, New York.

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 Computer by Means of Natural Selection. MIT Press, USA.

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

Koza, J.R., 1999. Genetic Programming III: Darwinian Invention and Problem Solving. Morgan Kaufmann, USA, ISBN: 9781558605435, Pages: 1154.

Kumarci, K., P.K. Dehkordi and I. Mahmodi, 2010. Calculation of plate natural frequency by genetic programming. J. Applied Sci., 10: 451-461.

Mahi, H. and H.F. Izabatene, 2011. Segmentation of satellite imagery using RBF neural network and genetic algorithm. Asian J. Applied Sci., 4: 186-194.

Mosavi, M.R., 2011. Applying genetic algorithm to fast and precise selection of GPS satellites. Asian J. Applied Sci., 4: 229-237.

Paterson, N. and M. Livesey, 1996. Distinguishing genotype and phenotype in genetic programming late breaking. Proceedings of the 1st Annual Conference on Genetic Programming, July 28-31, 1996, Stanford University, San Francisco, CA., USA., pp: 141-150.

Paterson, N. and M. Livesey, 1997. Evolving caching algorithms in C by genetic programming. Proceedings of the 2nd Annual Conference on Genetic Programming, July 13-16 1997, Stanford University, San Francisco, CA., USA., pp: 262-267.

Pugazhenthi, D. and S.P. Rajagopalan, 2007. Machine learning technique approaches in drug discovery, design and development. Inform. Technol. J., 6: 718-724.

Wasan, A.S., 2008. Finding linear equivalence of keystream generators using genetic simulated annealing. Inform. Technol. J., 7: 541-544.

Weijie, P., L. Shaobo, X. Qingsheng and Y. Guanci, 2011. Multi-objective optimization of RFID network based on genetic programming. Inf. Technol. J., 10: 2427-2433.