New Developments in Genetic Algorithms for Developing Software

Yousif Al-Bastaki[1]
Wasn Shaker[2]

[1]Information Technology College

University of Bahrain

BAHRAIN

[2]Department of MIS

Information Technology College

University of Bahrain

Kingdom of Bahrain

Abstract: Genetic Programming is the extension of the genetic model of learning the space of programs. These programs are expressed as trees. In this work we use another representation, where, in contrast to the above, the chromosomes are a list of integers representing the programs rather than a tree representation. This is called GADS (Genetic Algorithm for Developing Software). GADS is a technique for genetic programming where the genotype is distinct from the phenotype. Our work involves the development of GADS in two directions. The first is called CGADS, where we mean constrained GADS. That is constraining the randomness by which the integers represented by the chromosomes are created. The second is called ADGADS, where in this case we mean Automatically Defined functions with GADS. That is part of a chromosome that here represents reusable functions where the rest of the chromosome is the main body of the program. In other words ADGADS may be seen as a development of CGADS. To make a reasonable comparison between the three methods GADS, CGADS, and ADGADS as well as comparison with Koza's work [4], the problem of symbolic regression is considered. Detailed analysis of this problem is performed and the results are clearly in favor of CGADS and ADGADS. But, it is also clear that in many situations ADGADS has the edge in better performance than CGADS.

Key words: Genetic algorithm, Genetic programming, Symbolic Recognition Problem, GADS

Dr. Yousif AL-Bastaki: received a BSc. degree from University of Bahrain, Msc from University of Leeds, UK and a PhD degree from University of Nottingham, UK. Currently he is the Dean of College of IT at the University of Bahrain. His research interests are Neural Networks, genetic algorithms and Distance Education.

Dr. Wasn Shaker: She received her BSs, Msc and PhD from Technology University, Baghdad, Iraq, Currently she is an Assistant Professor at MIS Department, College of IT at University of Bahrain, Her research interests are Genetic Algorithms, Genetic Programming and Security

Introduction

Genetic programming (GP) 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 [4].

GP has been used in wide area of applications, and Koza [4] gives a good illustration of these applications. Symbolic regression problem is a well known problem which was addressed by Koza as a good example of the application of GP.

Koza and others who applied GP state that they have found a good efficient solution to this difficult problem. However, here we give yet another approach to the problem, namely that of using Genetic Algorithm for Developing Software (GADS) [6,7]. GADS is an implementation of GP 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, as the input to 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. Figure (1) outlines the central piece of that GADS algorithm. The genotype is operated on by the genetic operators (crossover, mutation and so on) in the usual range of ways available to Genetic Algorithm (GA).

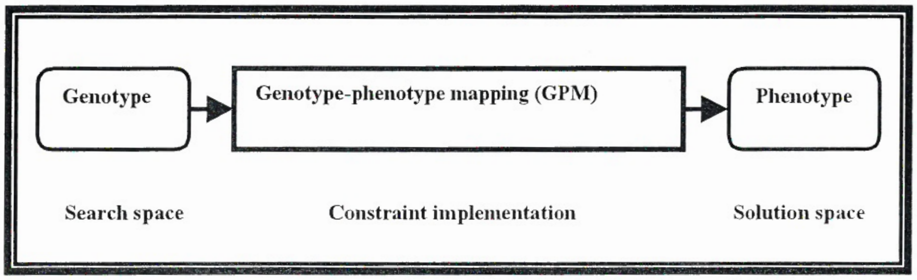

Figure (1): The Central Piece of the GADS Algorithm, the Genotype-Phenotype Mapping

The reason for investigating alternatives to tree genotype is that the behavior of GA system depends greatly on the way the genotypes represent the phenotypes. Thus, we should expect some change in the performance of the GP.

In this work, the original GADS, which we shall refer to as Traditional GADS (TGADS), was applied to the symbolic regression problem. To overcome some of the problems associated with the TGADS, we propose two differential modifications along the lines of:

1. Eliminating the loss of genes in TGADS,
2. Making the chromosomes with variable lengths,
3. Reducing the storage space, and
4. Representing the reusable functions.

The two modifications are:

a. Constrained GADS (CGADS) and
b. Automatic Defined function with GADS (ADGADS).

The application of GADS techniques (i.e., TGADS, CGADS, and ADGADS) to the symbolic regression problem resulted in better performance in the point of view of GP. For example Koza gave 34 iterations for the problem, while we got 1 iteration to all the three techniques for the same problem.

There are different approaches used by researchers in GP, such as the JB language, tree structure, Binary Genetic Programming (BGP), and Cellular encoding (or developmental genetic programming) [1]. In addition to these representation techniques, GADS can be considered as another representation technique that we consider here.

GADS Principles [6,7]

In the GADS, the chromosomes are lists of integers that represent the individual programs rather than using tree structure.

The main features of GADS are the following:

1. A clear distinction is drawn between genotype and phenotype.
2. In GADS, the genetic operators act on genotype, but in conventional GP no such distinction is made.
3. Using the genotype has the immediate advantage that conventional GA engine can be used to produce a program as a solution to a problem.

The phenotype language is the programming language in which the phenotypes produced by GADS are written. The choice of the phenotype language therefore depends on the problem domain which GADS is addressing.

There are many candidates for the ontogenic mapping, for example in the neural network problem each chromosome is a list of weights, each gene in the chromosome is a real number rather than a bit. To calculate the fitness of a given chromosome, the weights in the chromosome are assigned to the links in the corresponding network, the network is run on the training set and the sum of the squares of the errors is returned. Low error means high fitness [1,4].

The ontogenic mapping used here is the Backus Normal Form (BNF), where the productions of BNF are numbered from 0 to n, so that any production can be represented by a number in the range [0,n].

Every well-formed program in a language has a derivation according to the syntax of the language. A derivation is a sequence of productions which, when applied in turn transform the language's start symbol into the program. Thus, any program can be represented by a derivation, and any derivation can be represented by the sequence of integers which correspond to the productions in its derivation. In short, any program can be represented as a sequence of integers in the range [0,n].

The New GADS Techniques

Constrained GADS

Our main contribution is what we call CGADS. By this method, GADS is developed in order to improve TGADS such that:

1. There is no loss in the genes, and the losses will be after the tree building, i.e., the tail genes.
2. The chromosomes will be of variable lengths.
3. The storage space is reduced.

The concept of CGADS is the same as that of strongly typed GP (STGP) [3], in the sense that there are some constraints which are applied to the chromosomes that should be satisfied in the initial population generation, and genetic operations.

In this method, the generation of a gene in the chromosomes is simply dependent on some constraints. The constraint enforced here is:

If a_1, a_2, ... ,a_L is the genotype, the selection of gene a_{L+1} is not done randomly, instead, it is dependent on the gene a_L. Therefore, each gene has a number of allowed genes to appear after it.

In addition to syntactic constraint there is another constraint, which is: Each genotype in the initial population consists of at least five genes.

In this method, the genotypes have variable lengths generated randomly. Therefore, the size and the contents of these genotypes in the population can be altered through the successive generation due to the genetic operations.

The process of generating each genotype in the initial population is done as follows:

1. The length of the genotype is generated randomly, which is in the range 5-Max-length (where the Max-length is the maximum chromosome length) specified beforehand.
2. The first gene is generated randomly.
3. Until the genotype is fully created, repeat the following steps :
   a. The next gene is generated according to the neighbor gene created beforehand, and it is dependent on the syntactic constraint satisfied for that problem.
   b. Repeat the process of step (a) until all genes in that individual are created depending on the length specified in the first step.
4. If all individuals in the initial population are created, then continue to perform other operations upon the population created, otherwise, repeat steps 2-4.

Each genotype in the population created is syntactically correct. In this case, each chromosome in the population created can be defined using the C language structure as follows:

```
Struct Population
{
        int individual [ ];
        int length ;
        float fitness ;
}
```

The syntax rules definition and the phenotype generation are similar to those of TGADS, but here the conversion is faster than the conversion in TGADS. This is because there is no loss in the genes. For example, if the following chromosome is generated in TGADS;

10230335044530553

then, the equivalent chromosome can be generated in CGADS as follows:

150303

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 work, we need the following steps to perform the modified brood crossover operator:

1. Pick two parents from the population.
2. Select a gene randomly from the first parent.
3. Select a gene randomly from the second parent.
4. Test those genes for the syntactic constraint satisfaction.
5. If it conforms to CFG then exchange the genes, otherwise, select another gene from the second parent until the correct gene is found.

6. Steps 2-5 are repeated NB times to generate 2*NB offspring.
7. Evaluate each of the children for fitness. Select the best two; they are considered as the children of the parents. The remainders of the offspring are discarded.

The mutation operator involves the selection of a gene randomly from a genotype and generates a gene randomly to replace the selected gene. Check the left and right neighbors of that gene. If it conforms to the syntactic constraint of the CFG, 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 it depends on the problem.

Example

Consider the following genotype of length 25, which depends on Table I.

6140216020214170202160202

The corresponding phenotype generated as follows:

$(x+(x*x))*((x\%x)+(x*x)) = X^4 + X^3 + X^2 + X$

Table 1: The Syntax Rules of CGADS

Syntax Rules	Rule No.
<Sexp> : = < Input>	0
<Sexp> : = < Application>	1
<Inpute> : = X	2
<Input> : = 1	3
<Application> : = <Sexp> + <Sexp>	4
<Application> : = <Sexp> - <Sexp>	5
<Application> : = <Sexp> * <Sexp>	6
<Application> : = <Sexp> % <Sexp>	7

Automatically defined function with GADS

ADF is a technique used with CGADS. The phenotypes generated consist of two parts, main body and one ADF. Therefore, there are three types of points in the phenotypes, namely

- The root (which will always be the place-holding program symbol),
- Points in the first (left most) branch, i.e., ADF definition, and
- Points in the second (right most) branch, i.e., the value-returning branch.

The syntactic rules of the construction of the individuals are as follows:

1. The root of the tree generated (phenotype) must be the symbol "program".
2. The first function-defining branch of the tree is a composition of some function used in the returning branch. In this case the ADF consists of two parameters, so the terminal set consist of two variables which are X, and Y.
3. The second branch of the tree is a composition of the elements from the function set and the terminal set of the problem.

No values are returned by the first left most branch. They are merely function-defining branches which may or may not be called upon by the value-returning branch. The value returned by the entire tree is the value returned by the value-returning branch.

We do not specify what functions will be defined in the function-defining branches. We do not specify whether the defined function will actually be used (it being, of course, possible to solve this problem without any function definition by evolving the correct computer program in the retraining-branch).

Since a constrained syntactic structure is involved, we proceed here in the same manner as described in CGADS by first creating the initial random population so that every individual in the population has the required syntactic structure.

In this technique, the syntax used to convert the genotype to phenotype is the same, but in this case we need some modifications. So, we take the same example to explain this.

As we have said, the problem needs a terminal and function sets as follows:

T = { X, 1 }
F = {+ , - , * , % }

The syntax rule (BNF) used is presented in Table II.

Table 2: The Syntax Rules of ADGADS

Syntax Rules	Rule No.
<Sexp> : = < Input>	0
<Sexp> : = < Application>	1
<Inpute> : = X	2
<Input> : = 1	3
<Input> : = Y	4
<Input> : = M	5
<Application> : = <Sexp> + <Sexp>	6
<Application> : = <Sexp> - <Sexp>	7
<Application> : = <Sexp> * <Sexp>	8
<Application> : = <Sexp> % <Sexp>	9
<Application> : = <Sexp> function <Sexp>	10

In this technique, the modifications are to identify the parameters of the ADF which have numbers 4 and 5. The productions rule which points to ADF has the number 10.

From Table II we can see that there is a facility in the representation, such that when Koza separates the specified function set of value-returning branch from the specified function set of function-defining branch, we can define the same syntax rules.

As we have mentioned before, in our work, ADF consists of two parameters. Therefore, during the interpretation operation, the AST of the value-returning branch takes the two branches which are below considered as "function" when it shows the symbol "function". Control is then transferred to the root tree to point to the left branch (which is the function-defining branch). Then we compute the value, and return it. The interpretation operation is then continued.

Each chromosome in the population consists of just two parts: a main body and one ADF, see Figure 2. The generation of both the main body and ADF of any individual depends on some syntactic constraints.

In addition to the syntactic constraints presented in Fig. 2, there are other constraints which are:

1. Each individual in the population consists of the main body of the program and its subroutines. So, in this case the genotype is divided into two parts, one part specialized for ADF and the second part specialized for the main body.

2. The part of the ADF must consists (must consist) of at least 3 genes. Also, the part of main body must consist of at least 5 genes.

The size, and the content of each ADF and main body in the population can be dynamically changed during the genetic iterations due to its operations.

The process of generating each genotype in the initial population is done as in the following steps:

A. Generate ADF as follows:

   1. The length of AFD part is generated randomly, which is in the range 3-ADFMax-length specified beforehand.

   2. The first gene is generated randomly, so that it respects the constraint specified beforehand.

   3. Until the ADF part is fully created, repeat the following steps.

      3.1 Generate the next gene randomly such that it respects the neighbor gene created beforehand and the syntactic constraints presented in Fig. 2.

      3.2: Repeat the process of step 3.1 until all genes are created.

B. Generate the second part of the genotype, which is the main body by doing the following steps:

1. The length of the main body is generated randomly which is in range 5-Max-length of main body specified.
2. The first gene is generated randomly using the constraints satisfied beforehand.
3. Until the second part is fully created, repeat the following steps :

   3.1: Generate the next gene randomly such that it respects the neighbor gene created beforehand and the syntactic constraints presend in Fig. 2.

   3.2: Repeat the process of step 3.1 until all genes are created.

C. If all individuals (genotypes) in the initial population were created, then continue to perform other operations, otherwise, repeat steps from A-C.

Each individual created in the population can be defined using the C language structure as follows.

```
Struct Population
            {
                  int individual [ ] ;
                        int length of main body ;
                  int length of ADF ;
                  float fitness ;
            } ;
```

With ADGADS, the crossover is as described for CGADS, but with little modifications. That is, the crossover points should be chosen from the same part type of the parents, and only the genes of that part are exchanged.

Example

Consider the following genotype:

80404161705000000000 | 81818160302021611002030303

ADF **Main Body**

The corresponding program is as follows, and Fig. 3 shows its parse tree.

$(((1+X)*X)*(\text{function}(X,1)+1)*1) = (((1+X)*X)*(X^2+1)*1) = \mathbf{X^4+X^3+X^2+X^4}$

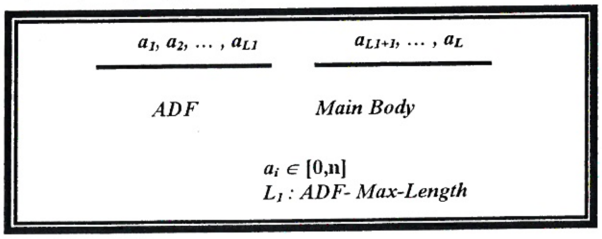

Figure 2: Individual structure in ADGADS

Conclusion

As in GA, the structure under adaptation in GADS is a set of fixed length strings, while in GP, the structure is a set of programs. Thus, GADS uses the GA engine and works on a simpler structure. The reason for using the same structure used in GA is that we expect some changes in the performance, and in addition to simplify the implementation of genetic operations such as crossover and mutation.

As was pointed out previously, there are some problems associated with TGADS, such as:

1. The chromosomes are represented as fixed length strings. Thus there is a big loss in the genes, which causes wastage in the storage space.
2. Reusable functions cannot be used.

This suggested us that TGADS needed to be investigated further to overcome these problems. Our work involved developing TGADS with the aim of overcoming these problems. Two new techniques evolved: CGADS and ADGADS. These two techniques have the following features:

1. Variable length strings may be used to represent chromosomes.
2. In CGADS the wasted genes are reduced greatly in numbers, thus reducing storage space.
3. In ADGADS reusable functions can be used. This is very beneficial when dealing with problems involving reusable subsystems.

In addition to above we also used more than one initial value in the phenotype generation which resulted in a great reduction in the number of wasted genes. This improvement was implemented for all types of GADS, even TGADS. Also the ephemeral terminal $\Re$ is added in the representation of the three GADS techniques, which is very important for many applications.

As for symbolic regression problem the following conclusions may be observed.

1. All GADS techniques TGADS, CGADS, and ADGADS gave correct solution after small number of generations comparable with Koza's work.
2. By using CGADS and ADGADS, all problems associated with TGADS have been overcome.
3. There are some differences in the performance of CGADS, ADGADS, and TGADS. E.g., the performance of TGADS and ADGADS is better than the performance of CGADS in term of number of generations; that is because GA in CGADS should learn the length and the contents of the chromosomes. But CGADS is better in terms of required storage space. However, this efficiency will be clearer when it is used to solve more complex problems. In addition, when a problem consists of reusable functions, ADGADS is the best.
4. The chromosome length affect the performance as follows: when the maximum length of the chromosomes of CGADS is increased, the required number of generations to find the correct solution is decreased. Also, when the chromosome length in TGADS is increased its efficiency will be less, because the search space size is increased.
5. The genotype to phenotype mapping is simpler in CGADS and ADGADS comparing to TGADS. That is because there is no loss of genes in CGADS and ADGADS which results in a faster implementation as all genes are of value.
6. The use of two initial values in the phenotype generation reduces the number of wasted genes, since there is no need to skip some genes at the beginning of the chromosome.
7. Duplicating some production rules in the syntax of the phenotype language increases the probability of using these productions. By this operation, a number of basic functions will appear in the programs with higher probabilities, and thus the performance is increased.

REFERENCES

1. W. BANZHAF, et al., **Genetic Programming: An Introduction**, Morgan Kaufmann and dpunkt-Verlag, 1998.
2. D.E. GOLDBERG, **Genetic algorithms in search, optimization, and machine learning**, Addison-Wesley, New York, 1989.
3. T. HAYNES, et al., **Strongly typed GP in evolving cooperation strategies**, in Eshelman, L.J. (Ed.), Proc. of the Sixth Int. Conf. on GA, Morgan Kaufmann, July 1995, pp. 271-278.
4. J.R. KOZA, **Genetic Programming: On the programming of computer by means of natural selection**, MIT press, 1992.
5. J.R. KOZA, **Genetic Programming II: Automatic discovery of reusable programs**, MIT press, 1995.
6. C. RYAN, et al., **Grammatical Evolution: Evolving Programs for an Arbitrary Language**, Lecture Notes in Computer Science, Paris, France, April 1998 Proceeding, pp. 83-96.
7. N. PATERSON & M. LIVESEY, **Evolving Caching Algorithms** in C by GP. In Genetic Programming 1997, MIT Press, pp. 262-267.