An Improved Design of Program Representation for Genetic Programming

Feng-Cheng Chang[1] and Hsiang-Cheh Huang[2]

[1] Dept. of Innovative Information and Technology, Tamkang University, TAIWAN
135170@mail.tku.edu.tw
[2] Dept. of Electrical Engineering, National University of Kaohsiung, TAIWAN
hch.nuk@gmail.com

Abstract. Although genetic programming (GP) is derived from genetic algorithm (GA), there are issues that prevent GP being as efficient as GA. The validity of a generated program, the design of ephemeral constants, and the empirical considerations of GP parameters are part of the issues affecting the convergence of a GP process. Based on many experimental results, an improved GP scheme is proposed in this paper. The design of the program representation and the storage of the computation result(s) ensure that a program is both structurally and behaviorally valid. Empirical considerations are also discussed, and a few guidelines of setting GP parameters are derived from the experimental observations.

Keywords: Genetic programming · GP

1 Introduction

Genetic algorithm (GA)[7][8] is a kind of Evolutionary computation (EC)[2]. The process searches for a sub-optimal solution with the affordable resources by randomly exploring the solution space. With a properly defined solution structure and the fitness function, the mechanism of GA, including the selection strategy, the crossover operator, and the mutation operator, would evolve the population to converge to the sub-optimal solution.

Genetic programming (GP)[9][11], which inherits the concepts from GA, is a method for finding a *program*. It has been applied to various research fields, such as [1]. However, its computation complexity is generally much higher than GA's. To speed up the GP process, the parallelism and locality properties of the evolution mechanism can be exploited[4]. The issue of producing invalid programs is also addressed in the literature[6][10]. By combining the concepts of Linear GP (LGP)[3] and VLIW, we proposed a program representation scheme to reduce the probability of producing invalid programs during the evolution[5].

Based on our previous work, an improved design is described in this paper. The improvements are developed by analyzing the run-time behavior of the programs in many experiments. In Sec. 2, the GP issues that affects the design are discussed. In Sec. 3, the guidelines for designing the program representation are derived by analyzing the run-time behavior of programs. In Sec. 4, the experimental results are discussed and the guidelines for setting the GP parameters are proposed. Finally, we conclude this work and describe the future directions in Sec. 5.

2 Issues Affecting the Design

Although the concepts of GP were derived from those of GA, the properties are different in many aspects. Three issues that affects our work are described in this section. Firstly, the genetic operations (crossover and mutation) in GP are more likely to produce invalid offspring than those in GA. There are a few researches that address this issue and try to tackle it. In our previous work, we proposed a program representation that is fixed-length and consists only operands. Thus, the genetic operations always produce a *structurally valid* program.

Another issue is related to the coefficients (literal values or constants) in a program. Since coefficients are constant values in an expression, they are not obtained from the input values. The solution for this issue is to design a mechanism that allows a program to embed constants. In a tree-based program representation, *ephemeral constants* are designed as the solution. An ephemeral constant is generated by a predefined random function and inserted into the tree as a leaf. In our previous design, a similar concept is adopted: the *LOAD operator* assigns a random number to the destination register (variable).

The other is the termination condition of a GP process. In GA, the fitness function defines the high-dimensional solution space. Therefore, the target of the optimization is the fittest solution to the fitness function. In GP, however, the target is actually the fittest *program behavior* that consumes the input values and produce the output values. The problems are: (1) there are multiple fittest programs and (2) the training set is much smaller than the problem/solution space. If the size of the training set is quite small to the size of the problem space, the *over-fitting* problem may affect the quality of the searched program. To prevent the over-fitting effect, *regulation* should be designed into the fitness function. Since there is no theoretically exact match, a GP process is usually terminated by the number of generations. However, it is application-dependent. If the size of the training set is roughly the size of the problem space, targeting at the fittest program is a proper design.

3 Improvements of the Design

In this paper, we focus on the first two issues mentioned in the previous section. In our previous work, a program representation was proposed to address the two issues. After a few extended experiments, we discovered that the design only solves the issues in the structural aspect. The issues in the behavioral aspect are not covered properly. The operand-only program representation removes the problem of invalid number of operands (caused by the mutation operation) but it does not remove the problem of invalid value of operands (caused by both genetic operations). For example, an exception will be raised when the divisor for a division operator is zero. When the exception occurs, it can be handled with different approaches:

– terminate the program and mark the fitness to be the lowest value;
– terminate the program, remove the program, and create a valid program to replace it;
– set the result to infinite (IEEE754 Inf);
– set the result to not-a-number (IEEE754 NaN);
– discard the exception and continue the next computation.

According to the experimental results, the first approach tends to fill the population with invalid programs. It makes the population difficult to converge. The second approach ensures that only valid programs exist in the population. However, the invalid-program issue occurs so frequently that the GP process spends a lot of time generating programs. The third and the fourth approach share the similar problem. Once an Inf or NaN appears, the subsequent operators are likely to produce Inf or NaN. The result is that the values held in the register file are dominated by Inf and NaN. Therefore, we found that the fifth approach is more practical than the others. It ignores the exception and keeps the computation results not "polluted" by special values (Inf and NaN); and it neither leaves a lot of invalid programs in the population nor wastes time generating temporary valid programs. An alternative viewpoint of this approach is to replace the *normal* version with the *conditional* version if the operator requires an operand with restrictions.

We mentioned that the LOAD operator in our program representation is the adapted design of the ephemeral constant in the conventional GP. The LOAD operator requires two operands: the input literal value and the output register. The representation of the literal value can be implemented as an an IEEE754 64-bit value or an integer value. The former one is straightforward and allows to optimize the value via the genetic operations. However, the approach makes it possible to include Inf or NaN in a program, which reduces the behavioral validity of a program. The latter one, by using an integer, guarantees that each bit pattern corresponds to a distinct integer value. The drawback is that it may require a few arithmetic operations to obtain the desired value. A related problem is to design the size (the number of bits) of an integer literal. Theoretically it is possible to use any integer size in the program representation. Considering that the register indexes are also integers, it is better to leave no "holes" in the bit-level representation. Therefore, the following rules are added to the design of the program representation:

– the number of registers is the power of 2;
– the size of the literal value is multiple of the size of the register index.

With the rules, the result of a genetic operation is guaranteed to generate a structurally and behaviorally valid program.

A minor revision to the previous work is how the result register is determined. Previously, the program execution result is expected to be stored in a specific register (as part of the fitness function definition). Sometimes a program that never touches the result register may be generated. Although the program is both structurally and behaviorally valid, it is meaningless in that the program effectively does nothing in terms of the output. To ensure a program writing to the result register at least once, the result register is defined as one of the output registers in the program. By this definition, the output register of a LOAD operator is allowed but not generally a good choice.

4 Experiments and Discussions

Based on the improved design of the GP program representation, we performed a few experiments with different configurations. Some of the experiments worked as expected and the others did not converge to an acceptable level. It indicates that some empirical factors should be considered for tuning the GP process. A configuration that produce acceptable results is described in Sec. 4.1. The empirical considerations are described in Sec. 4.2.

4.1 An Example Configuration

The following GP configuration was used in the experiments and produced the acceptable results.

– the population size is 400;
– the selection strategy is the best 25% of the population;
– the mutation rate is 0.001;
– the single-point crossover is used;
– the maximum number of generations is 100000.

There are 16 double-precision (IEEE754) floating-point registers for storing temporary computation results and they are indexed from 0 to 15. The program representation is designed as follows. A program contains 10 *instructions* and each instruction contains a sequence of operators. For the operands of an operator, i, j, and k are the register indexes. The *value* is an unsigned integer in the range $[0, 15]$.

Operator	**Operands**	**Description**
LOAD	**value**, **i**	$reg_i \leftarrow value$
ADD	**i**, **j**, **k**	$reg_k \leftarrow reg_i + reg_j$
MUL	**i**, **j**, **k**	$reg_k \leftarrow reg_i \times reg_j$
NEG	**i**, **j**	$reg_j \leftarrow -reg_i$
CDIV	**i**, **j**, **k**	$reg_k \leftarrow reg_i \div reg_j$ if $reg_j \neq 0$
MOV	**i**, **j**	$reg_j \leftarrow reg_i$

There are two sets of experiments to examine the convergence of the GP population. The target of the first set is that the computation result depends only on the input values. At the beginning of the simulations, 100 training item are generated. Each item contains four random numbers in the range $[0.0, 1.0)$ as the input values and the output is the *sum* of the four numbers. Each time the program is executed with a training item, the registers are reset to all-zero and then reg_0 to reg_3 are initialized to the input values. At the beginning of Sec. 3, we mentioned that the over-fitting problem is not in the scope of this paper. Therefore, the target is to find the fittest program to the training set. The result is expected to be stored in the output register of the MOV operator of the last instruction. To evaluate the fitness of a program with the training set, the mean square error (MSE) of the output values to the expected values are computed, and the inverse of the MSE is returned. The simulation results showed that an exact match (the sum of the four numbers) could be found between 2500 to 7500 generations.

The target of the second set is that the computation result depends on both the input values and some coefficients. There are also 100 training items. A training item contains four random numbers in $[0.0, 1, 0)$ and the expected output is the *average* of the numbers. All the other setting are the same as the previous set of experiments. The simulation results showed that an exact match could not be found within 100000 generations. The fitness reached the range from 1005.45 (MSE=0.00099) to 1321.46 (MSE=0.00076).

4.2 Empirical Considerations

Although the rules of our GP design reduce the probability of generating improper programs, it simply makes the GP process be executed smoothly. The convergence of the population relies on some other configuration parameters. By analyzing the configurations of the poor-result experiments, some observations are listed below:

– The number of registers (N_{regs}) is correlated to the number of register operands (N_{opds}) in a program. It is likely to produce reasonable results when $N_{opds} \geq 5N_{regs}$.
– Although we need large enough program to meet the previous criterion, excessively large program tends to slow down the convergence.
– GP is quite different from GA in that the bit-pattern similarity is not generally consistent with the solution similarity. Therefore, the mutation rate should be high enough to avoid being trapped in a local minimum.
– According to the same reason of the previous point, the size of the population should be large enough to keep the diversity of the bit patterns.

5 Conclusions

In this paper, based on our previous work, an improved design that ensures generating structurally and behaviorally valid programs during the GP process was proposed. A few experiments for verifying the effectiveness of the design were performed and analyzed. Some observations on setting the GP configuration parameters were also conducted. The key points of the design are listed below:

– A program is a sequence of instructions; an instruction is a sequence of defined operators; and an instruction is encoded by using the operands.
– The LOAD operator is the adapted mechanism of the traditional ephemeral constants. The literal value is restricted to an integer.
– A conditional operation is used when one of its operands is restricted.
– The result register index is obtained from an output operand.
– The program size should be large enough. The empirical results show that the number of operands is more than 5 times of the number of registers.
– The mutation rate and the population size should be large enough.

In this paper, only the arithmetic operators were used in the simulation. We would like to incorporate logical operators into the experiments in the future.

Acknowledgements

This work was partially supported by the Ministry of Science and Technology, Taiwan, under Grant MOST 104-2221-E-032-061.

References

1. Al-Sahaf, H., Al-Sahaf, A., Xue, B., Johnston, M., Zhang, M.: Automatically evolving rotation-invariant texture image descriptors by genetic programming. IEEE Transactions on Evolutionary Computation 21(1), 83–101 (Feb 2017)
2. Back, T., Emmerich, M., Shir, O.: Evolutionary algorithms for real world applications [application notes]. Computational Intelligence Magazine, IEEE 3(1), 64–67 (Feb 2008)
3. Brameier, M.F., Banzhaf, W.: Linear Genetic Programming. Springer US (2007)
4. Chang, F.C., Huang, H.C.: A refactoring method for cache-efficient swarm intelligence algorithms. Information Sciences 192, 39–49 (Jun 2012)
5. Chang, F.C., Huang, H.C.: A design of genetic programming scheme with VLIW concepts. In: The Twelfth International Conference on Intelligent Information Hiding and Multimedia Signal Processing. pp. 307–314. Kaohsiung, Taiwan (Nov 2016)
6. Ferreira, C.: Gene expression programming: a new adaptive algorithm for solving problems. Complex Systems 13, 87–129 (2001)
7. Fogel, D.B.: Evolutionary Computation. IEEE Press, New York (1998)
8. Kantardzic, M.: Data Mining: Concepts, Models, Methods, and Algorithms. IEEE Press (2003)
9. Koza, J.R.: Genetic programming - on the programming of computers by means of natural selection. Complex adaptive systems, MIT Press (1993)
10. Oltean, M., Grosan, C.: A comparison of several linear genetic programming techniques. Complex Systems 14(4), 285–314 (2003)
11. Walker, M.: Introduction to genetic programming. Tech. rep., (Oct 2001)