Genetic System Generation

Nada M. A. Al Salami

Abstract— **This research attempt to evolve computer programs, to solve a problem by using the input-output specifications of this problem. The evolutionary process adapts Genetic Programming to search for a good Finite State Automata that efficiently satisfies these specifications. It has been presented that for large and complex problems, it is necessary to divide them into sub problem(s) and simultaneously breed both sub- program(s) and a calling program.**

Index Terms— **Evolutionary Algorithm, Genetic Algorithm, Genetic Programming, Finite State Machine.**

I. Introduction

The input to an **Automatic Programming System (APS)** is a programming problem expressed in the specification language defined by that APS. The specification may state tasks to be performed within the world domain or restriction to be enforced, or both, the job of an APS is to translate that specification into a program in some target language using it's knowledge bases and problem solving strategies [1][2][3]. The recent resurgence of interest in AP with Genetic Algorithm has been spurred by the work on **Genetic Programming (GP).** GP paradigm provides a way to do program induction by searching the space of possible computer programs for an individual computer program that is highly fit in solving or approximately solving the problem at hand. [4] [5] [6] [7][8]. Unfortunately, since every real life problem are dynamic problem, thus their behaviors are much complex, GP suffers from serious weaknesses. Complex systems often include chaotic behavior(the classic example of chaos theory is "the Butterfly effect"), which is to say that the dynamics of these systems are nonlinear and difficult to predict over time, even while the systems themselves are deterministic machines following a strict sequence of cause and effect. Natural chaotic systems may be difficult to predict but they will still exhibit structure that is different than purely random systems. [9][10]. GP weakness and the chaotic behavior of real live problem is reduced if induction process based on the meaning of the low-level primitives rather than their structure. In this paper we attempted to scale-up GP application to real live problems, by focusing on the meaning rather than the structure of a program to overcome the representation problem. Abstract machine, like Fixit State Machine, is used to specify the meaning of a programming language mathematically [11] [12], as we shall explain in detailed in the rest of this paper.

Nada M. A. Al Salami (1971) Author is with the Manegement Information System Department, University of Al Zaytoonah, Amman , Jordan, (her e-mail: nada.alsalami@yahoo.com).

II. Theoretical Definition

The meaning of a program P can be specified by set of function transformation from states to states; hence P effects a transformation:

$$(P)\ ^{X}\mathbf{initial} \longrightarrow\ ^{X}\mathbf{final}$$

on a state vector X, which consists of an association of the variable manipulated by the program and their values. A Program P can be defined as 9- tuples, called Semantic Finite State Automata **(SFSA)**: $\mathbf{P=(\ x, X, T, F,\ Z, I, O, \gamma, X_{initial})}$, where: x is the set of system variables, X is the set of system states, $\mathbf{X= \{\ X_{initial}, ----, X_{final}\ \}}$, **T** is the time scale: T =[0, ∞), F is the set of primitive functions, Z is the state transition function, $\mathbf{Z = \{(f, X, t): (f, X, t) \in F \times X \times T\ , z(f, X, t) = (\bullet X, \bullet t)\}}$, **I** is the set of inputs, **O** is the set of outputs, **γ** is the readout function, and $\mathbf{X_{initial}}$ is the initial state of the system: $\mathbf{X_{initial} \in X}$.

All sets involved in the definition of P are arbitrary, except **T**, and **F**. Time scale **T** must be some subset of the set **[0, ∞)** of nonnegative integer numbers, while the set of primitive function **F** must be a subset of the set $\mathbf{C_L(F_L)}$ of all computable functions in the language **L** and sufficient to generate the remainder functions. Two features characterize state transition function:

$$\mathbf{1-\ Z\ (\ -,\ -,\ t)\ =\ (X_{initial},\ 1)\ \ if\ t = 0} \quad \text{... Eq 1}$$

$$\mathbf{2-\ Z\ (\ f,\ X,\ t)\ =\ z\ (\ f,\ z\ (\ f(t\text{-}1),\ X,\ t\text{-}1))\ \ if\ t \neq 0} \quad \text{... Eq. 2}$$

The concepts of reusable parameterized sub-systems can be implemented by restricting the transition functions of the main system, so that it has the ability to call and pass parameters to one or more such sub-systems. Suppose we have sub-system 'P, and main-system P, then they can be defined by the following 9-tuples:

$$\mathbf{P\ (x, X, T, F, Z, 1, 0, X_{initial},\ \gamma)}$$

$$\mathbf{P\ (\bullet x, \dot{}X, \dot{}T, \dot{}F, \dot{}Z,\ \dot{}I,\ \dot{}O, \dot{}X_{initial},\ \dot{}\gamma)}$$

where:

$\mathbf{\bullet x \subseteq x}$, $\mathbf{\dot{}X_{initial} \in X}$, then there exit $\mathbf{*f \in F, z \in Z, \dot{}f, \in F}$, and $\mathbf{\bullet z \in \dot{}Z}$, and ***h*** is a function defined over $\mathbf{\dot{}Z}$ with value in $\mathbf{\dot{}X}$ is defined as follows:

$$\boldsymbol{h} = \mathbf{\dot{}z\ (\dot{}f,\ \dot{}X_{initial}\ , 1) = X_{\boldsymbol{h}}\ , t_i,} \quad \text{.. Eq 3}$$

$$\mathbf{z\ (*f,\ X,\ t) = z\ (}\boldsymbol{h}\mathbf{\ ,\ X\ ,\ t) = X_{\boldsymbol{h}}\ , t,} \quad \text{..Eq 4}$$

*f is a special function we call it sub-SFSA function to distinguish it from other primitive functions in the set F. Also, we call the sub-system $\mathbf{\dot{}P}$, sub-SFSA, to distinguish it from the main SFSA. Formally, a system $\mathbf{\dot{}P}$ is a sub-system of a system **P**, iff: $\mathbf{\bullet x \subseteq x}$, $\mathbf{\dot{}T \subseteq T, \dot{}I \subseteq I, \dot{}O \subseteq O,}$ $\mathbf{\dot{}\gamma}$ must be the restriction of **γ** to $\mathbf{\dot{}O}$, and $\mathbf{\dot{}F \subseteq N}$, where **N** is the set of restrictions of **F to $\dot{}$T. If ($\dot{}$f, $\dot{}$X, $\dot{}$t)** is an element of $\mathbf{\dot{}F \times \dot{}X \times \dot{}T}$, then there exists $\mathbf{f \in F}$, such that the restriction of f to $\mathbf{\dot{}T}$ is $\mathbf{\dot{}f}$, and $\mathbf{\dot{}z\ (\dot{}f, \dot{}X, \dot{}t)}$ **is z (f, X, t)**.

III. System Induction

On the basis of the theoretical approach sketch in section 2, we shall define and explain another theoretical approach for system induction, it is modification of system theory given in reference[13]. The newly defined approach highly depend on input-output behavior of the problem, it is expressed as 7-tuples: **(IOS, S, F, a_1, T_{max}, β, υ):**

1. Input-Output Specification (IOS):
IOS is establishing the input-output boundaries of the system. It describes the inputs that the system is designed to handle and the outputs that the system is designed to produce. An IOS is a 6-tuples: **IOS = (T, I, O, T_i, T_O, η)**. Where **T**, is the time scale of IOS, **I** is the set of inputs, **O** is a set of outputs, T_i is a set of input trajectories defined over **T**, with values in **I**, T_O, is a set of output trajectories defined over **T**, with values in **O**, and **η** is a function defined over T_i whose values are subset of T_O; that is, **η** matches with each given input trajectories the set of all output trajectories that might, or could be, or eligible to be produced by some systems as output, experiencing the given input trajectory. A system **P** satisfies IOS if there is a state **X** of **P**, and some subset **U** not empty of the time scale **T** of **P**, such that for every input trajectory **g** in T_i, there is an output trajectory **h** in T_O matched with **g** by **η** such that the output trajectory generated by **P**, started in the state **X** is:

γ (Z (f (g), X, t) = η(h(t)), For every **t Є U** ……. Eg.(5)

2. Syntax Term (S):
Refers to the written form of a program as far as possible independently of its meaning. In particular it concerns the legality or well-formed ness of a program relative to a set of grammatical rules, and parsing algorithms for the discovery of the grammatical structure of such well-formed programs. S is a set of rules governing the construction of allowed or legal system forms.

3. Primitive Function (F):
Each f_i must be coupled with its effect on both the state vector X, and the time scale T of the system. Some primitive functions may serve as primitive building blocks for more complex functions or even sub-systems.

4. Learning Parameter (a_1):
is a positive real number specifying the minimum accepted degree of matching between an IOS, and the real observed behavior of the system over the time scale, T_x, of IOS only.

5.Complexity Para(T_{max}, β):
T_{max} and β parameters are merits of system complexity: size and time, respectively. It is important to note that there is a fundamental difference between a time scale **T** and an execution time of a system. **T** represents system size, it defines points within the overall system, whereas, **β**, is the time required by the machine to complete system execution, hence it is high sensitive to the machine type.

6. System Proof Plan (υ):
Prove process should be a part of the statement of system induction problem especially when the **IOS** is imprecise or inadequate to generate an accurate system. We say **P** is correct iff it computes a certain function **f** from $X_{initial}$ Є **X** properly, that is if for each X_i Є **X**, **P** (X_i) is defined, i.e. **P** does not loop for X_i, and is equal to $f(X_i)$. Broadly speaking, there have been two main approaches to the problem of developing methods for making programs more reliable [11]: **Systematized testing, and Mathematical proof.**

Our works use systematized testing approach as a proof plane. The usual method for verifying that a program is correct by testing is by choosing a finite sample of states $X_1, X_2, ..., X_n$ and running **P** on each of them to verify that: $P(X_1) = f(X_1), P(X_2) = f(X_2), \ldots\ldots P(X_n) = f(X_n)$. Formally, if testing approach is used for system verification, a system proof is denoted $\upsilon=(\alpha_2, \mathbf{d})$, where α_2 is a positive real parameter defining the maximum accepted error from testing process. α_2 focus on the degree of generality, so that α_1 ,and α_2 , parameters suggest a fundamental tradeoff between training and generality. On the other hand, **d** represents a set of test cases pairs (O_i, K_i), where K_i is a sequence of initial state $X_{initial}$ and input I_i.

In addition to using the idea of sub-system functions, i.e., **sub-FSA**, for complex software it is better to divide the process of system induction into sub-system(s) and main-system induction. Sub-system induction must be accomplished with several objectives; first one is that a suitable solution to sub-problem must determine a solution to the next higher level problem. Second is to ensure that the figure of merit of the sub-system has relationships to the figure of merit of the top-level problem. The third objective is to ensure specific functional relationships of sub-system proof plans to the system proof plan of the top-level problems.

IV. Genetic Generation Process

Within the context of the suggested mathematical approaches, to automatically generate a system means search to find an appropriate SFSA satisfying IOS efficiently, with regard to learning and complexity parameters. Then, proof plan υ must be applied to that SFSA for further assessing its performance and correctness. If that SFSA behaves well with υ , it may be considered as a solution or approximate solution to the problem, else, some or all terms in the statement of the problem of system induction must be modified, and the process is repeated until a good SFSA is found, or no further revisions can be made. The search space in Genetic Program Generation algorithm is the space of all possible computer programs described as an 9-tuples SFSA. Multi-objective fitness measure is adopted to incorporate a combination of correctness (satisfy **IOS**), parsimony (smallness **T**), and efficiency (smallness **β**).The fitness value of individual is computed by the following equation:

$$fitness(i) = \delta\left(\alpha_1 - \sum_{j=0}^{Tx} \left|\eta(T_i(j)) - \eta(R_i(j))\right|\right) + (T_{max} - T_i) + (\beta - \beta_i)$$

……..Eq. 6

where: δ is the weight parameter, δ >=2, βi the run time of individual i, Ti is the time scale of the individual iI, Ri is the actual calculated input trajectory of individual i. Since the goal is to satisfy the IOS, first term in equation (1) is multiplied by the weight parameter **δ**. Learning parameter $\boldsymbol{\alpha_1}$, is used ultimately to guide the search process. Values for **δ** are selected experimentally forms substantial number of runs. To give rise to the fitness variation in the overall population from one generation to the next, the fitness of each individual is computed proportional to the fitness summation of all individuals in the population as follows:

$$"fitness(i) = \frac{'fitness(i)}{\sum_{j=0}^{M} 'fitness(j)} \quad \text{..Eq. 7}$$

where: M: is the population size. Equation (7) is also adjusted so that the adjusted fitness lies between 0 and 1:

$$fitness(i) = \frac{1}{"fitness(i)} \quad \text{..Eq. 8}$$

Three types of points are defined in each individual: transition, function, and function arguments. When structure-preserving crossover is performed, any point type anywhere in the first selected individuals may be chosen as the crossover point of the first parent. The crossover point of the second parent must be chosen only from among points of this type. The restriction in the choice of the second crossover points ensures the syntactic validity of the offspring. The proposed APS breeds SFSA to solve problems by executing the following algorithm:

Genetic System Generation Algorithm

♦ Initialize the following: variable terms, and(learning, complexity, generalization, and δ) parameters.

♦ Generate an initial population of random SFSA represented as a composition of the constant and variable terms, which are consistent with S.

♦ Iteratively perform the following subsets until the termination criterion has been satisfied

A. Run each individual in the current population over all fitness cases and assign it a fitness value using equation (8).

B. Create a new population based on operation probability: Darwinian Reproduction, Structure-Preserving Crossover, and Structure-Preserving Mutation.

C. Apply test plans υ to the best-of-generation individual, and compute the error returned from testing e.

♦The best of generation individual with small error; $e=<\alpha_2$, is designated as the result for the run.

V. Result and Discussion

A. Input-Output Specification

Unfortunately, when we deal with complex systems and real live problem, strong feedback (positive as well as negative) and many interactions exist: i.e. chaotic behavior, as we explain in part I. Thus, we need to find a way to control chaos, to understand, and predict what may happen long term. In these cases input and output specifications are self organized, which mean that trajectory data are collected and enhanced over time, when genetic generation process runs again and again. Figure1, specify clearly that SFSA populations, with high trajectory information converge to the solution in less time than these populations with little trajectory information. Although trajectory data are changed over time, but by experiment, it still sensitive to initial configuration of SFSA (sensitivity to the initial conditions). In figure 2, for the same problem we change in initial configurations of state set X, and data trajectory sets. The behaviors of the resulting SFSAs are completely different. There is a fundamental difference between a crossovers occurring in a sub-SFSA versus one occurring in the main-SFSA. Since the later usually contains multiple references to the sub-SFSA(s), a crossover occurring in the sub-SFSA is usually leveraged in the sense that it simultaneously affects the main-SFSA in several places. In contrast, a crossover occurring in the main-SFSA provides no such leverage. In addition, because the population is architecturally diverse, parents selected to participate in the crossover operation will usually possess different numbers of sub-SFSA(s). The proposed architecture-altering operations are:

Creating sub-SFSA,
Deleting sub-SFSA,
Adding Variables, and
Deleting Variables

.

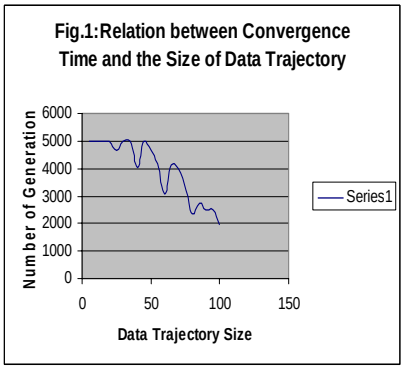

figure 1: Relation between Convergence Time and the size of data trajectory

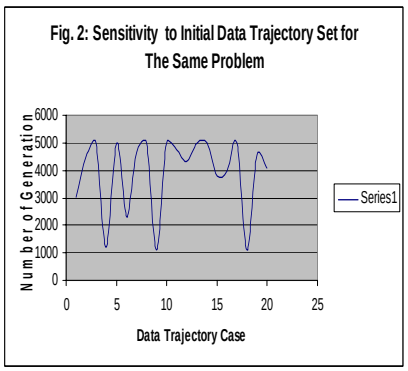

Figure 2: Sensitivity to initial Data Trajectory set for the same Problem

B. Performance

Because of its probabilistic steps, non-convergence and premature convergence problems become inherent features of genetic generation process. To minimize the effect of these problems, multiple independent runs of a problem must be made. Best-of-run individual from all such multiple independent runs can then be designated as the result of the group of runs. If every run of GPG were successful in yielding a solution, the computational effort required to get the solution would depends primarily on four factors: population size, M, number of generation that are run, g, (g must be less than or equal to the maximum number of generation G) the amount of processing required for fitness measure over all fitness cases, and the amount of processing required for test phase e, we assume that the processing time to measure the fitness of an individual is its run time, P. If success occurs on the same generation of every run, then the computational effort E would be computed as follows [8][15]:

$$E = M \bullet g \bullet \beta \bullet e \qquad \text{.....E q. 9}$$

Since the value of e is too small with respect to other factors, we shall not consider it. However, in most cases, success occurs on different generations in different runs, then the computational effort E would be computed as follows:

$$E = M \bullet g_{avr} \bullet \beta \qquad \text{.....Eq.10}$$

where: g_{avr} is the average number of executed generations. Since GPG is a probabilistic algorithm, not all runs are successful at yielding a solution to the problem by generation G. Thus, the computational effort is computed in this way, first determining the number of independent runs R needed to yield a success with a certain probability. Second, multiply R by the amount of processing required for each run, that is. The number of independent runs R required to satisfy the success predicate by generation i with a probability z which depends on both z and P (M, i), where z is the probability of satisfying the success predicate by generation i at least once in R runs defined by:

$$z = 1-[1-P(M, i)]R \qquad \text{.....Eq.11}$$

P (M, i) is the cumulative probability of success for all the generations between generation 0 and generation i. P (M, i) is computed after experimentally obtaining an estimate for the instantaneous probability Y (M, i) that a particular run with a population size M yields, for the first time, on a specified generation i, an individual is satisfying the success predicate for the problem [8]. This experimental measurement of Y(M, i) usually requires a substantial number of runs. After taking logarithms for equation 4, we find:

$$R = \left| \frac{\log(1-z)}{\log(1-\rho(M,i))} \right| \qquad \text{.....Eq.12}$$

The computational effort E, is the minimal value of the total number of individuals that must be processed to yield a solution for the problem with z probability (ex: z = 99%):

$$E = M \bullet (\dot{g} + 1) \bullet \beta \bullet R \qquad \text{....Eq.13}$$

Where $\dot{g}$ is the first generation The computational effort ratio, R_E, is the ratio of the computational effort without sub-SFSA to the computational effort with sub-SFSA:

$$R_E = \frac{Ewithoutsub-SFSA}{Ewithsub-SFSA} \qquad \text{.....Eq. 14}$$

The fitness ratio, $R_{fitness}$, is the ratio of the average fitness without sub-SFSA to the average fitness, with sub-SFSA [8], for a problem.

$$R_{fitness} = \frac{fitnessaveragewithoutsub-SFSA}{fitnessaveragewithsub-SFSA} \qquad \text{.. ..Eq. 15}$$

VI. Conclusion

1. The proposed APS can be changed to produce software in a different implementation language without significantly affecting existing problem specification, leading to an increase in the system productivity. Trajectory data of the system are Self-organization to reflect the chaotic behavior in real live applications. Convergences time is highly sensitive to the initial input-output specification of the problem.

2-Programs with less sub-programs tend to disappear because they accrue fitness from generation to generation, more slowly than those programs with sub-programs. The proposed APS gain leverage in simultaneously solving the problems of system induction and evolving the architecture of a single or multi-part system, with the aid of four new architecture-altering operations. Sub-systems can be reused to solve multiple problems. They provide rational way to reduce software cost and increase software quality.

3-Although complexity parameters provide a way to bound the search space of the proposed method, their effect in the induction process is relatively less than the effect of learning and generalization parameters. System analyst must select a value for α_2, so that missing or incorrect specifications of some input-output relationships of the problem don't lead to un convergence situation. Real payoff will come when α_2 value is too small (near zero) due to the poor behavior of the system during testing phase.

REFERENCES

[1] E. Rich, "Artificial Intelligence", McGraw-Hill, Inc, 1983.W.-K. Chen, *Linear Networks and Systems* (Book style). Belmont, CA: Wadsworth, 1993, pp. 123–135.

[2] S. K. Misra, and, P.J. Jalles, "Third-Generation versus Fourth-Generation Software Development", IEEE software, July, pp. 8-14, 1998.

[3] J. Verner, and, G. Tate, "Estimating Size and Effort in Fourth-Generation Development", IEEE software, July, pp. 15-22, 1988.

[4] A. Schafer, "Graphical Interactions with an Automatic Programming System", IEEE Transactions on systems, Man, and cybernetics, vol. 18, No. 4, July/August, pp. 575-591, 1988.

[5] J. G. Cleaveland, "Building Application Generators", IEEE software, July, pp. 25-33, 1988.

[6] J. R. Koza, "Genetic Programming: on the Programming of Computer by means of Natural Selection", Massachusetts Institute of technology, 2004.

[7] D. E. Golberg, "Genetic Algorithm in Search, Optimization, and Machine Learning", Addison-Wesley, 1989.

[8] M. Mitchell, "An Introduction to Genetic Algorithm", Massachusetts Institute of tech., 1996.

[9] Leonard Smith, "Chaos: Avery Short Introduction", OXFORD university press, 2007.

[10] George Rzevski, Petr Skobelev, "Emergent Intelligence in Large Scale Multi- Systems", Journal or Education and Information Technologies Issue 2, Volume 1, 2007 64, http://www.naun.org/journals/educationinformation/eit-11.pdf

[11] J. M. Brady, "The Theory of Computer Science: A Programming Approach", Chapman, and Hall ltd., 1977.

[12] J. E. Hoperoft, and J. D. Ullman, "Introduction to Automata Theory: Languages and Computation", Addison Wesley, Reading, Mass., 1979.

[13] A. W. Wymore, "Theory of System", Handbook of Software Engineering, CBS Publishers, pp. 119133, 1986.

[14] P. Berlioux, and P. Bizard, "Algorithms: The Construction, Proof, and Analysis of Programs", John Wiley and Son Ltd, 1986.

[15] J. P. Koza, "Two Ways of Discovering the size and shape of a computer program to solve a problem", pp. 287-294.

[16] A. Kent, J. G. Williams, C. M. Hall, "Genetic Programming", Encyclopedia of Computer Science and Technology, Marcel Dekker, pp. 29-43, 1998.

[17] R. E. Smith, B. A. Dike, and S. A. Stegmann, "Fitness Inheritance in Genetic Algorithm", In Proceedings of the 1995 ACM Symposium on Applied Computing, 1995.

[18] J. R. Koza, S. H. Al-Sakra and W. J. Lee, Automated re-invention of six patented op- 37 tica lens systems using genetic programming, Genetic and Evolutionary Computation Conference (GECCO) '05 (Washington, DC, 2005).

[19] Riccardo Poli, William B. Langdon , Nicholas F. McPhee, John R. Koza, "Genetic Programming :An Introductory Tutorial and a Survey of Techniques and pplications" , Technical Report CES-475 ISSN: 1744-8050 October 2007. essex.ac.uk/dces/research/publications/.../2007/ces475.pdf