American J. of Engineering and Applied Sciences 2 (4): 789-795, 2009
ISSN 1941-7020
© 2009 Science Publications

# Evolutionary Algorithm Definition

Nada M.A. AL-Salami
Department of Management Information Systems, Faculty of Economic and Business,
Al Zaytoonah University of Jordan, Amman, Jordan

**Abstract: Problem statement:** Most resent evolutionary algorithms work under weak theoretical basis and thus, they are computationally expensive. **Approach:** This study discussed the use of new evolutionary algorithm for automatic programming, based on theoretical definitions of program behaviors. Evolutionary process adapted fixed and self-organized input-output specification of the problem, to evolve good finite state machine that efficiently satisfies these specifications. **Results:** The proposed algorithm enhanced evolutionary process by simultaneously solving multi-parts from the same problem. **Conclusion:** The probability that the algorithm will converge to the optimal solution was highly enhanced when decomposing the main problem into multi-part.

**Key words:** Evolutionary computation, genetic programming, automatic programming, system design, self-organization system

## INTRODUCTION

Life on earth has evolved for some 3.5 billion years. Initially only the strongest creatures survived, but over time some creatures developed the ability to recall past series of events and apply that knowledge towards making intelligent decisions. The very existence of humans is testimony to the fact that our ancestors were able to outwit, rather than out power, those whom they were in competition with, in other words, their response to the threat of their environment was intellectual adaptation. This could be regarded as the beginning of intelligent behavior. “Intelligent behavior is a composite ability to predict one’s environment coupled with a translation of each prediction into a suitable response in light of some objective”. Evolutionary Computing is a research area within Computer Science, which draws inspiration from the process of natural evolution. Evolutionary computation, offers practical advantages to the researcher facing difficult optimization problems. These advantages are multi-fold, including the simplicity of the approach, its robust response to changing circumstance, its flexibility and many other facets. The evolutionary approach can be applied to problems where heuristic solutions are not available or generally lead to unsatisfactory results. Thus evolutionary computing is needed for Developing automated problem solvers, where the most powerful natural problem solvers are human Brain and evolutionary process (that created the human brain). Designing the problem solvers based on human brain leads to the field of “neurocomputing”. While the second one leads to evolutionary computing. The algorithms involved in Evolutionary computing are termed as Evolutionary Algorithms (EA). Application of EC may includes: Bioinformatics, numerical combinatorial optimization, system modeling and identifications, planning and control, engineering design, data mining, machine learning and artificial life. In evolutionary computation, the idea of self-modification has its origins in the ontogenetic programming system of Spector and Stoffel[1], the graph re-writing system of Gruau[2] and the developmental method of evolving graphs and circuits of Miller[3].

In this study, we propose new evolutionary algorithm, based on theoretical definition of system and it’s input-output boundaries, in contrast with traditional evolutionary methods. Then compare it to the most recently used evolutionary algorithms.

**Background:** Evolutionary algorithms are ubiquitous nowadays, having been successfully applied to numerous problems from different domains, including optimization, automatic programming, machine learning, operations research, bioinformatics and social systems. In many cases the mathematical function, which describes the problem is not known and the values at certain parameters are obtained from simulations. In contrast to many other optimization techniques an important advantage of evolutionary algorithms is they can cope with multi-modal functions[4]. Additional advantages are listed as follows:

- It is conceptually simple. The procedure may be written as difference equation:

  $$x[t + 1] = s(v (x [t])) \quad (1)$$

  Where:

  | | | |
  |---|---|---|
  | x[t] | = | The population at time t under a representation x |
  | v | = | A random variation operator |
  | s | = | The selection operator |

- It is representation independent, in contrast with other numerical techniques, which might be applicable for only continuous values or other constrained sets
- It offers a framework such that it is comparably easy to incorporate prior knowledge about the problem. Incorporating such information focuses the evolutionary search, yielding a more efficient exploration of the state space of possible solutions
- Can also be combined with more traditional optimization techniques. This may be as simple as the use of a gradient minimization used after primary search with an evolutionary algorithm, or it may involve simultaneous application of other algorithms
- The evaluation of each solution can be handled in parallel and only selection (which requires at least pair wise competition) requires some serial processing
- Traditional methods of optimization are not robust to dynamic changes in problem the environment and often require a complete restart in order to provide a solution (e.g., dynamic programming). In contrast, evolutionary algorithms can be used to adapt solutions to changing circumstance
- It has the ability to address problems for which there are no human experts. Although human expertise should be used when it is available, it often proves less than adequate for automating problem-solving routines

However there are some disadvantages of EC such as:

- There is no guarantee for optimum solution within finite time
- Works under weak theoretical basis
- May need parameter tuning
- Computationally expensive

**Resentally area in EC:** Sub-area of the term evolutionary computation or evolutionary algorithms includes:

- Evolutionary Programming (EP)
- Evolution Strategies (ES)
- Genetic Algorithm (GA)
- Genetic Programming (GP)

They all share a common conceptual base of simulating the evolution of individual structures via processes of selection, mutation and reproduction. The processes depend on the perceived performance of the individual structures as defined by the problem, Table 1. Evolutionary programming, developed by Fogel *et al*.[4] traditionally has used representations that are tailored to the problem domain. EP is often used as an optimizer, although it arose from the desire to generate machine intelligence. Rechenberg and Schwefel developed Evolutionary Strategies. The algorithm is similar to EP in many ways. In the last few years they have had something of a renaissance and have become more popular, particularly in research work. However, in practical and industrial systems, they have been eclipsed somewhat by the success of the GA. One reason behind the GA’s success is that its advocates are very good at describing the algorithm in an easy to understand and non-mathematical way.

A genotype-phenotype mapping therefore implies an algorithm that transforms an input string of numbers encoding a genotype into another string of numbers that comprises the phenotype of an individual. Both evolutionary programming and evolutionary strategies are known as phenotypic algorithms (physical characteristic of the genotype like smart, beautiful, healthy), whereas the genetic algorithm is a genotypic algorithm (Particular set of genes in a genome). Phenotypic Algorithms operate directly on the parameters of the system itself, whereas genotypic algorithms operate on strings representing the system. In other words, the analogy in biology to Phenotypic Algorithms is a direct change in an animal's behavior or body and the analogy to Genotypic is a change in the animal’s genes, which lie behind the behavior or body.GA is implemented by having arrays of bits or characters to represent the chromosomes. In EP there are no such restrictions for the representation. In most cases the representation follows from the problem. EP typically uses an adaptive mutation operator in which the severity of mutations is often reduced as the global optimum is approached while GA’s use a pre-fixed mutation operator. Among the schemes to adapt the mutation step size, the most widely studied being the “meta-evolutionary” technique in which the variance of the mutation distribution is subject to mutation by a fixed variance mutation operator that evolves along with the solution.

Table 1: Comparison between different Evolutionary Algorithms

| | Algorithm type | Developed researcher | Individual representation | Operators | Selection method |
|---|---|---|---|---|---|
| Evolutionary programming | Phenotypic | Fogel *et al.*, 1966 [4] | FSMs | Mutation only | Tournament |
| Evolutionary strategies | Phenotypic | Rechenberg, 1973 [4-8] | Real values | Mainly mutation | Ranking |
| Genetic algorithm | Genotypic | Holland, 1975 [5] | Bitstrings | Mainly crossover | Proportionate |
| Genetic programming | Phenotypic | Koza, 1992 [4][8] | Expression trees | Mainly crossover | Proportionate |

On the other hand, when comparing evolutionary programming to evolution strategies, one can identify the following differences: When implemented to solve real-valued function optimization problems, both typically operate on the real values themselves and use adaptive reproduction operators. EP typically uses stochastic tournament selection while ES typically uses deterministic selection. EP does not use crossover operators while ES uses crossover. Some specific advantages of genetic programming are that no analytical knowledge is needed and still could get accurate results. GP approach does scale with the problem size. GP does impose restrictions on how the structure of solutions should be formulated. There are several variants of GP, some of them are: Linear Genetic Programming (LGP), Gene Expression Programming (GEP), Multi Expression Programming (MEP), Cartesian Genetic Programming (CGP), Traceless Genetic Programming (TGP) and Genetic Algorithm for Deriving Software (GADS). Following we shall concentrate on CGP, since it is the most near to our proposed method[5-8].

**Cartesian genetic programming:** Cartesian genetic programming was originally developed by Miller and Thomson[9] for the purpose of evolving digital circuits and represents a program as a directed graph. One of the benefits of this type of representation is the implicit re-use of nodes in the directed graph. Originally CGP used a program topology defined by a rectangular grid of nodes with a user defined number of rows and columns. In CGP, the genotype is a fixed-length representation and consists of a list of integers which encode the function and connections of each node in the directed graph. The genotype is then mapped to an indexed graph that can be executed as a program. In CGP there are very large numbers of genotypes that map to identical genotypes due to the presence of a large amount of redundancy. Firstly there is node redundancy that is caused by genes associated with nodes that are not part of the connected graph representing the program. Another form of redundancy in CGP, also present in all other forms of GP is, functional redundancy. Simon Harding and Ltd introduce computational development using a form of Cartesian Genetic Programming that includes self-modification operations. One advantage of this approach is that the system can be used to solve computational problems[10]. The interesting characteristic of CGP are:

- More powerful program encoding using graphs, than using conventional GP tree-like representations, the population of strings are of fixed length, whereas their corresponding graphs are of variable length depending on the number of genes in use
- Efficient evaluation derived from the intrinsic feature of subgraph-reuse exhibited by graphs
- Less complicated graph recombination via the crossover and mutation genetic operators

## MATERIALS AND METHODS

Proposed method is based on theoretical system definitions discussed in[11], thus it overcomes the difficulties of traditional method, in addition it has all attractive characteristic of CGP. Our evolutionary algorithm evolves FSA that achieve input-output specification of the problem. FSA transit from state to state according to trajectory data sets, which either fixed, or Self-Organized during evolutionary process. Trajectory data are stored as a string of numbers (the genotype) and evolved to achieve the optimum mapping. The theory is based on McCarthy's formalism of the theory of computer science[12-13]: There is a set of base function F and a set of strategies C for building new function out of old, the closure C (F) comprises all computable functions. For any language L it may be possible to isolate a set ($F_L$.) of base functions to express the meaning of identifiers and statements and a set ($C_L$) of strategies to express the meaning of the linguistic structure and data structures of L. Then the meaning of P in L would be computable function in $C_L(F_L)$:

$$\text{Meaning (P): } L \rightarrow C_L(F_L)$$

So, P effects a transformation:

$$(P)\ X_{initial} \rightarrow X_{final}$$

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)[11]:

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

Where:

| | | |
|---|---|---|
| x | = | The set of system variables |
| X | = | The set of system states, $X = \{X_{initial}, \ldots.., X_{final}\}$ |
| T | = | The time scale, $T = [0, \infty)$ |
| F | = | The set of primitive functions |
| Z | = | The state transition function, $Z = \{(f, X, t): (f, X, t) \in F \times X \times T, z(f, X, t) = (^{\bullet}X, ^{\bullet}t)\}$ |
| I | = | The set of inputs |
| O | = | The set of outputs |
| $\gamma$ | = | The readout function |
| $X_{initial}$ | = | The initial state of the system, $X_{initial} \in X$ |

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

$$z ( -, -, t) = (X_{initial}, 1) \text{ if } t = 0 \quad (2)$$

$$z(f, X, t) = z (f, z( f(t-1), X, t-1)) \text{ if } t \neq 0 \quad (3)$$

The concepts of reusable parameterized subsystems 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 $^{\bullet}P$ and main-system P, then they can be defined by the following 9-tuples:

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

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

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

$$h = {}^{\bullet}z (^{\bullet}f, ^{\bullet}X_{initial}, 1) = X_{h,} t_i \quad (4)$$

$$z(*f, X, t) = z (h, X, t) = X_{h,}t \quad (5)$$

*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 $^{\bullet}S$, sub-SFSA, to distinguish it from the main SFSA. Formally, a system $^{\bullet}S$ is a sub-system of a system S, iff: $^{\bullet}x \subseteq x$, $^{\bullet}T \subseteq T$, $^{\bullet}I \subseteq I$, $^{\bullet}O \subseteq O$, $^{\bullet}\gamma$ must be the restriction of $\gamma$ to $^{\bullet}O$ and $^{\bullet}F \subseteq N$, where N is the set of restrictions of F to $^{\bullet}T$. If $(^{\bullet}f, ^{\bullet}X, ^{\bullet}t)$ is an element of $^{\bullet}F \times {}^{\bullet}X \times {}^{\bullet}T$, then there exists $f \in F$, such that the restriction of f to $^{\bullet}T$ is $^{\bullet}f$ and $^{\bullet}z (^{\bullet}f, ^{\bullet}X, ^{\bullet}t)$ is z (f, X, t).

The idea of recursive function could be simply applied with the proposed method using mathematical induction. The principle of mathematical induction can be used to construct system as well as proofs. Consider the following definition of the recursion function $f_r$, which is highly reminiscent of proofs by mathematical induction:

$f_r (X) = X, t = t_{max} +1$ if $X = 0$ (base of induction)

$f_r (X) = X_{initial} = X, t = 0$ otherwise (induction step)

where, $T = [0, t_{max}]$.

**Input-Output Specification (IOS):** An IOS is a modification for input-output specification used with ant colony optimization algorithm given in[14]. 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 not a system, but it determines the set of all systems that satisfy the IOS. It is a 6-tuples:

$$IOS = (T, I, O, T_i, T_o, \eta)$$

Where:

| | | |
|---|---|---|
| T | = | The time scale of IOS |
| I | = | The set of inputs |
| O | = | A set of outputs |
| $T_i$ | = | A set of input trajectories defined over T, with values in I |
| T | = | A set of output trajectories defined over T, with values in O |
| H | = | A function defined over $T_i$ whose values are subset of $T_o$; that is, $\eta$ matches with each given input trajectories $T_i$ 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 $T_i$. |

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 $\eta$ such that the output trajectory generated by S, started in the state X is:

$$\gamma (Z (f (g), X, t) = \eta(h(t)) \text{ For every } t \in U \quad (6)$$

## RESULTS

The search space in genetic program generation algorithm is the set 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), whereas, β, is the time required by the machine to complete system execution, hence it is high sensitive to the machine type.. The fitness value of individual is computed by the following equation:

$$\text{fitness}(i) = \delta\left(\alpha_i - \sum_{j=0}^{T_x}\left|\eta\left(T_i\,(j)\right) - \eta(R_i\,(j))\right|\right) + (T_{max} - T_i\,) + (\beta - \beta_i\,) \qquad (7)$$

Where:

| | | |
|---|---|---|
| $\delta$ | = | The weight parameter, $\delta >= 2$ |
| $\beta_i$ | = | The run time of individual i |
| $T_x$ | = | The time scale of the individual i |
| $R_i$ | = | The actual calculated input trajectory of individual i |

Three types of points are defined in each individual: Transition z€Z, function f€F 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. When sub-SFSA functions are being used, the initial random generation of the population must be created so that each individual has the intended constrained structure, that is one main-SFSA and zero or more sub-SFSA defined under the condition of transition function restriction. The population at generation 0 is architecturally diverse, the architecture of the participating individuals are changing during a run of GPG and hence determine the architecture of a multi-part system dynamically during the run.

Sub-systems can be reused to solve multiple problems. They provide rational way to reduce software cost and increase software quality. 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. From Fig. 1, its clear that using high number of sub-program may lead to speed up algorithm convergences). The operation of "sub-SFSA creation", creates new sub-system within an overall system:

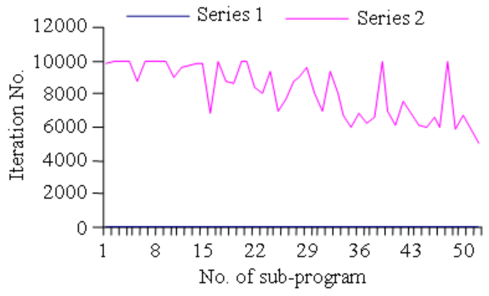

Fig. 1: Convergance time with respect to no. of sub-program

**Creating sub-SFSA algorithm:**

- An individual is selected from the population, based on I's fitness value
- Randomly create sub-SFSA defined by a 9-tuples P ($^\bullet$x, $^\bullet$X, $^\bullet$T, $^\bullet$F, $^\bullet$Z, $^\bullet$I, $^\bullet$O, $^{\bullet X}$ initial, $^\bullet\gamma$), where $^\bullet$x is a subset of the corresponding term x in the main-FSA and $^\bullet X_{initial}$ gets its value from the state of the calling transition function
- A uniquely-named sub-SFSA function $^*$f is added to the set F of the main-SFSA such that each occurrence of $^*$f in the transition function set Z will be replaced by the transition function $^\bullet$z ($^\bullet$f, $^\bullet X_{initial}$, 1) of the newly created sub-SFSA
- Randomly choose a point in the main-SFSA transition function and mutate it with *

The last step is optional, since it just ensures the existence of at least one reference to the newly created sub-SFSA function *f.

## DISSCUSION

**Fixed versus self-organized data trajectory sets:** During evolutionary process, trajectory information play the primary role. States transformations are done according to them values, which either fixed, or self-organized during evolutionary process. Trajectory data are stored as a string of numbers (the genotype) and evolved to achieve the optimum mapping.

**Example:** Assume we try to solve a search problem to find an occurrence of element e in a list L of i integer number. At least i+2 inputs are needed (two inputs to read the values of e and i and i input to read i element of L).

Table 2: Fixed input-output trajectory sets

| | | | | | | |
|---|---|---|---|---|---|---|
| I = {e,i, L[1], L[2], ….., L[i]} | | | | | | |
| 0 = {0, 1) | | | | | | |
| $T_x$ = | 1 | 2 | 3 | 4 | 5 | 6 |
| $T_i$ = | I[1] | I[2] | I[j] | -1 | -1 | -1 |
| $T_o$ = | -1 | -1 | -1 | -1 | O1 | O2 |
| Then η: | | | | | | |
| η (Ti(t)) = | O1 if t = 5 | | | | | |
| | O2 if t = 6 , and -1 otherwise | | | | | |

Table 3: Self-Organized Input and Output Trajectory sets.

| | |
|---|---|
| Ti(at iteration: 0) = -1, -1, I[1] | To(at iteration: 0) = O1, -1, -1 |
| Ti(at iteration:10) = I[1], -1, -1, I[2] | To(at iteration: 10) = -1, -1, O1, -1 |
| Ti(at iteration: n) = I[1], I[2], -1, -1, -1, -1, I[3] | To(at iteration: n) = -1, -1, -1, -1, O1, -1, -1 |

Accordingly, at least two different output may be produced by the program to indicate search result (found and not found, or, 1 and 0). Obviously no such outputs are produced unless at least four operations are executed that are: input e, input i, input L[1] and check its equality with e. The time scale of IOS must be defined under the worst case. i.e., L[i] = e, or no occurrence of e is found at all. Now, we can pre-specified $T_x$, $T_i$ and $T_o$, as fixed set as given in Table 2. In this case, Evolutionary process computes the fitness value for each individual based on applying fitness function only. While in case of self-organized case, trajectory sets are randomly built according to currently available information about system input-output boundaries, as seen in Table 3. At the end of i generations, these sets are modified according to the input-output specification of the best individuals, obviously, such modifications are vary continually until the required results are produced. Although trajectory data are changed over time, but by experiment, it still sensitive to initial configuration of SFSA. This is one of the most important characteristic of a chaotic system (butterfly effect sensitivity to the initial conditions)[15].

## CONCLUSION

- Proposed method is based on theoretical system definitions, thus it overcomes the difficulties of traditional method, in addition it have all attractive characteristic of CGP
- Sub-systems can be reused to solve multiple problems. They provide rational way to reduce software cost and increase software quality. 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
- Trajectory information play important role in the Evolutionary Process. Fixed specification of trajectory sets, speed-up convergence time of the algorithm. Although self-organized trajectory sets are useful tools in chaotic behavior, they take more time to converge to the fine solution

## REFERENCES

1. Spector, L. and K. Stoffel, 1996. Onto genetic programming. Proceedings of the 1st Annual Conference, MIT Press, Stanford University, CA., USA., pp: 394-399.
2. Gruau, F., 1994. Neural network synthesis using cellular encoding and the genetic algorithm. PhD Thesis, Laboratoire de l'Informatique du Parallelisme, Ecole Normale Superieure de Lyon, France. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.29.5939
3. Miller, J.F. and P. Thomson, 2003. A developmental method for growing graphs and circuits. Lecture Notes Comput. Sci., 2606: 93-104. http://cat.inist.fr/?aModele=afficheN&cpsidt=15672446
4. Abraham, A., N. Nedjah and L.D.M. Mourelle, 2006. Evolutionary computation: from genetic algorithms to genetic programming. Stud. Comput. Intel., 13: 1-20. http://www.springerlink.com/content/l646l37m787g121t/
5. Grosan, C. and A. Abraham, 2007. Hybrid evolutionary algorithms: Methodologies, architectures and reviews. Stud. Comput. Intell., 75: 1-17. http://www.softcomputing.net/hea1.pdf
6. Montes, H.A. and J.L. Wyatt, 2003. Cartesian genetic programming for image processing tasks. Proceedings of the IASTED International Conference on Neural Networks and Computational Intelligence, May 19-21, Cancun, Mexico, pp: 185-190. http://md1.csa.com/partners/viewrecord.php?requester=gs&collection=TRD&recid=200311420157CI&q=&uid=788263103&setcookie=yes
7. Koza, J.R., 1995. Survey of genetic algorithms and genetic programming. Proceeding of the Conference on Microelectronics Communications Technology Producing Quality Products Mobile and Portable Power Emerging Technologies, Nov. 7-9, IEEE Xplore Press, San Francisco, CA, USA., pp: 589. http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=485447

8. Poli, R., W.B. Langdon, N.F. McPhee and J.R. Koza, 2007. Genetic programming: An introductory tutorial and a survey of techniques and applications. Technical report CES-475. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.126.3889
9. Miller, J.F. and P. Thomson, 2000. Cartesian genetic programming. Proceedings of European Conference on Genetic Programming, Apr. 15-16, ACM Press, Springer-Verlag, London, UK., pp: 121-132. http://portal.acm.org/citation.cfm?id=704075
10. Harding, S.L., J.F. Miller and W. Banzhaf, 2007. Self-modifying Cartesian genetic programming. Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation, July 7-11, ACM Press, New York, USA., pp: 1021-1028. http://portal.acm.org/citation.cfm?id=1277161&dl=GUIDE&coll=GUIDE&CFID=51767811&CFTOKEN=87184646
11. Nada Al Salami, 2009. System evolving using ant colony optimization algorithm. J. Comput. Sci., 5: 380-387.
12. Hoperoft, J.E. and J.D. Ullman, 1979. Introduction to Automata Theory: Languages and Computation. Addison Wesley Publishing Company, USA., ISBN: 10: 020102988X, pp: 418.
13. Wymore, 1986. Theory of System. Handbook of Software Engineering, CBS Publishers, pp: 119-133.
14. AL-Salami, N.M.A., 2009. System evolving using ant colony optimization algorithm. J. Comput. Sci., 5: 380-387. http://www.scipub.org/fulltext/jcs/jcs55380-387.pdf
15. Smith, L., 2007. Chaos: A Very Short Introduction. Illustrated Edn., Oxford University Press, USA., ISBN: 10: 0192853783, pp: 176.