Learning Automata-Based Co-Evolutionary Genetic Algorithms for Function Optimization

F. Abtahi*, M. R. Meybodi*, M. M. Ebadzadeh*, R. Maani**
* Amirkabir University of Technology, Department of Computer Engineering and IT, Tehran, Iran
**University of Manitoba, Department of Computer Science, Winnipeg, Canada
{abtahi, mmeybodi, ebadzadeh}@aut.ac.ir, maani@cs.umanitoba.ca

Abstract— **Co-evolutionary genetic algorithms are being used to solve the problems which are naturally distributed and need the composition of couple of elements or partial solutions to be solved. In these algorithms, the problem decomposes into several elements and for each element, a sub-population is regarded. These sub-populations evolve separately by considering the way of interactions among them. The general solution is the result of composition of some individuals from the mentioned sub-populations. These algorithms are more similar to the natural evolution and can be run in parallel and therefore, are more efficient. Function optimization problem is one of the examples of distributed problems in which co-evolutionary genetic algorithms can be used appropriately. To solve the problem, at the first step, we should know whether the variables of the problem are dependent or not. In each case, a different approach should be taken. But, sometimes the recognition of variable dependencies is too hard because of the complexity or discreteness of the functions. This paper represents a solution using a combination of co-evolutionary genetic algorithm and learning automata to address this problem. Learning automata are able to learn the dependence (or independence) of variables and choose the appropriate approach for each case. Experimental results show that using learning automata improves the efficiency of co-evolutionary algorithms and make them suitable for the optimization of any function.**

Keywords— **Co-evolutionary Genetic Algorithm, Learning Automata, Function Optimization.**

I. INTRODUCTION

Nowadays evolutionary computing techniques, in which solutions evolve gradually, are highly used in theoretical problems as well as real world issues. One of the most popular of them is Genetic Algorithm. Genetic Algorithm is proposed by *Holland* in 1970, represents an abstract model of *Darwin's Theory of Evolution.* It uses basic operators, namely crossover and mutation, and is based on the evolution of a population of similar individuals (species from the same class). Each individual can be a solution for the target problem we want to solve. Although this approach has been a good way of solving many problems and to propose desirable results, it involves some problems and shortcomings in some cases.

Many problems, such as social phenomena or economic issues are intrinsically distributed. In other words, they can be seen as several independent entities with their own goals interacting with each other. This interaction engenders a general behavior for the whole system. To address these kinds of problems, researchers have looked for new powerful methods base on natural processing.

The proposed approach to deal with modeling and solving events composed of several independent entities evolving synchronously is *Co-Evolutionary Algorithm.* The main idea of *Co-Evolutionary Algorithm* is originated form natural and biological observations. In nature, the evolution of some species is the result of evolution of a collection of phenotypically similar individuals and not a population of similar individuals of a same class or species. Hence, instead of using a population composed of similar individuals, each of which brings in a general solution, it is better to use collections of individuals, each of which represents a part of the general solution. This approach is not only more natural but also modular. Generally, two kinds of *Co-Evolutionary Algorithms* are presended [3]:

1. *CCGA: Cooperative Co-Evolutionary Genetic Algorithm*
2. *LCGA: Lousely Coupled (or competitive) Genetic Algorithm*

In this paper, we study *CCGA* and try to use it for function optimization problems. In *CCGA* individuals in each sub-population cooperate in order to find an appropriate solution.

Two different versions of *CCGA* can be used for function optimization problems. The first version, called *CCGA-1*, is suitable for the fuctions in which the variables are independent and therefore it is possible to find the optimized value of the function by optimizing each variable separately. The second version, namely *CCGA-2*, is used when the variables are dependent to each other and as a result, the optimized value of the function cannot be found by independent optimization of variables. But, the recognition of dependence (independence) of the variables is not always an easy job. In fact, problem occurs when it is impossible to decisively determine whether variables are dependent or not. This happens when the function is so complex and so the dependency of variables is unknown or when the function is discrete and the relation of variables is obscure. It seems we need a solution for these situations. In fact, instead of choosing the correct version manually, we can use an algorithm to learn the relation between variables and select the right version of *CCGA* each time.

Learning Automata technique is used as the learning tool in this paper. *Learning Automata* are machines capable of doing finite set of actions. Along with each action, *LA* is rewarded or punished according to the desirability of that action. Based on these rewards and

1-4244-2407-8/08/$20.00 ©2008 IEEE

punishments, *LA* tries to adjust the likelihood of taking actions in order that the probability of taking a desirable action increases (or similarly the probability of taking an undesirable action decreases).

The rest of the paper is organized as follows. In section 2 the definition of *CCGA*, its different versions and the way of its usage in function optimization are presented. Section 3 includes the definition of *Learning Automata* and the learning algorithm used in them. In section 4, our proposed algorithm is presented and its efficiency in optimization of different functions is studied. Finally, we bring in our conclusions in section 5 and state the advantages of the proposed approach.

II. Cooperative Co-Evolutionary Genetic Algorithm And Its Application to Fuction Optimization

Actually, *CCGA* uses the basic principles of the *Co-Evolutionary Genetic Algorithm* and extend them as follows [1]:

- Each species (or sub-population) shows an element of potential solution
- The whole solution is produced by putting individuals from all species together
- Reward or punishment for each individual of each species is computed in terms of the complete solution in which the individual participates
- The evolution of each species is based on the standard *Genetic Algorithm*

Fig. 1 shows the standard Genetic algorithm and Fig. 2 describes *CCGA*. Considering these figures, you can compare the differences between these two general solutions.

Each optimization problem includes a function (or some functions) with one or more variables and the goal is finding the best value for each variable in order that the function gets its optimized value. To deal with these kinds of problems using *CCGA*, in the first step we should find a way to decompose the problem and define sub-populations.

A natural decomposition for an optimization problem with *N* variables is defining *N* sub-population and assigning the value of each variable to one of these sub-populations. In the next step the fitness of each value (individual) of sub-populations should be evaluated. This evaluation is carried out as follows:

1. The value is put along with randomly selected values of the other sub-populations. This composition makes an *N* dimensional vector.
2. The fitness of the vector can be calculated easily by applying it to the function.
3. The calculated fitness is assigned to each element of the vector (individuals).

In fact, the fitness of each individual of a sub-population shows how it cooperates with the other individuals —which belong to the other sub-populations— in order to produce an appropriate solution, and that is why this approach is entitled "Cooperative". Fig. 3 depicts *CCGA* flowchart for solving optimization problems.

```
gen = 0
Pop(gen) = randomly initialized population
evaluate fitness of each individual in Pop(gen)
while termination condition = false do begin
        gen = gen + 1
        select Pop(gen) from Pop(gen-1) based on
        fitness
        apply genetic operators to Pop(gen)
        evaluate fitness of each individual in
        Pop(gen)
end
```

Figure 1. Pseudocode of standard Genetic Algorithm [1]

```
gen = 0
for each species s do begin
     Pop_s(gen) = randomly initialized population
     evaluate fitness of each individual in
     Pop_s(gen)
end
while termination condition = false do begin
     gen = gen + 1
     for each species s do begin
        select Pop_s(gen) from Pop_s(gen-1) based
        on fitness
        apply genetic operators to Pop_s(gen)
        evaluate fitness of each individual in
        Pop_s(gen)
     end
end
```

Figure 2. Pseudocode of CCGA [1]

The first version of *CCGA,* namely *CCGA-1*, includes the following steps:

1. A population of individuals (values) is produced for each variable of the function
2. The primary fitness of each individual is evaluated by:
   a. Each individual is integrated with random individuals from the other sub-populations to make a vector
   b. The produced vector is applied to the target function
   c. The value of the function is assigned to each individual participated in the vector
3. Sub-populations evolve using the standard *Genetic Algorithm.*
4. Fitness of each individual of the sub-populations is calculated by integrating it with the best individuals of the other sub-populations and applying the produced vector to the target function. In this step only one sub-population is active and the others are frozen.
5. Steps 3 and 4 iterate until a predefined condition is satisfied.

As you can see, *CCGA-1* uses the simplest way of fitness evaluation and it suffers from the problem of greediness. This algorithm can be used only for the optimizations of functions with independent variables in which the separate optimization of each variable leads to the optimization of the function. In functions with dependent variables a solution should be found which does not have the greediness problem.

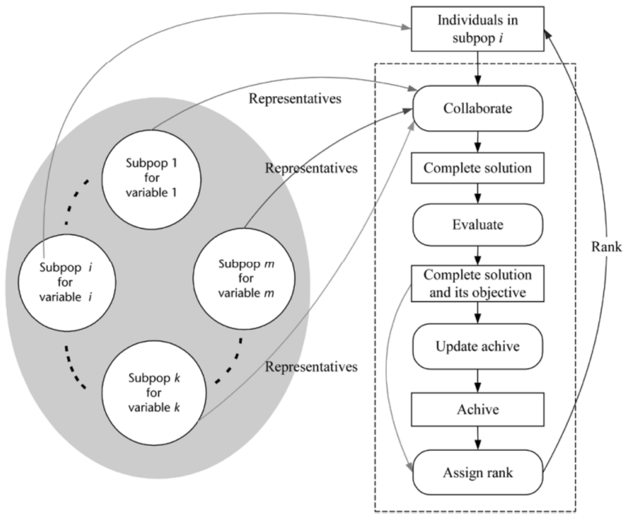

Figure 3. CCGA flowchart for solving optimization problems [2]

To address the mentioned issue, *CCGA-2* can be used. In this approach, the fitness of each individual is calculated following these steps:

1. The fitness of each individual of a sub-population is evaluated in two cases:
   a. The individual is integrated with the best individuals of the other sub-populations.
   b. The individual is integrated with random individuals of the other sub-populations.
2. For each case the produced vector is applied to the target function.
3. The better value among two cases is considered as the individual's fitness.

CCGA-2 is not greedy and for the functions with dependent variables produces better results in comparison with *CCGA-1*.

III. LEARNING AUTOMATA

Learning Automaton (LA) is a machine that can perform finite number of actions. Each selected action is evaluated by a random environment. The result is presented to the *LA* as a positive or negative signal and then the *LA* will use this response to choose its next action. The final goal of *LA* is to learn to select the best action among all its actions. The best action is an action that maximizes the probability of receiving reward from the environment. *LA* and its interaction with the environment are shown in figure 1 [4, 5].

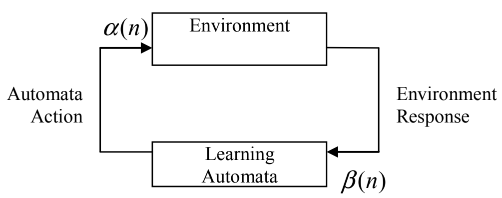

Figure 4. Relation between Learning Automata and Environment [5]

The environment is represented by triple $E \equiv \{\alpha, \beta, c\}$. Here, $\alpha \equiv \{\alpha_1, \alpha_2, ..., \alpha_r\}$ is the set of inputs, $\beta \equiv \{\beta_1, \beta_2, ..., \beta_m\}$ is the set of outputs and $c \equiv \{c_1, c_2, ..., c_r\}$ is the set of punishment probabilities. If β is a two member set, the environment is of type *P*. In such an environment, $\beta_1 = 1$ and $\beta_2 = 0$ are considered as punishment and reward respectively. In a *Q* environment, β can take a discrete value from finite values in [0,1]. In an environment of type *S*, $\beta(n)$ is a random variable in [0,1]. c_i is the probability that action α_i has an undesirable result. In a static environment, c_i values are fixed, but in a dynamic environment, they might change as time passes.

Fixed structure *LA* is represented by quintuple $\{\alpha, \beta, F, G, \phi\}$ in which $\alpha \equiv \{\alpha_1, \alpha_2, ..., \alpha_r\}$ is the set of actions, $\beta \equiv \{\beta_1, \beta_2, ..., \beta_r\}$ is the set of inputs, $\phi(n) \equiv \{\phi_1, \phi_2, ..., \phi_k\}$ is the set of internal states at time *n*, $F: \phi \times \beta \rightarrow \phi$ is next state transition function and $G: \phi \rightarrow \alpha$ is the output function that maps the current state of *LA* to next output (action).

Variable structure *LA* can be defined as $\{\alpha, \beta, p, T\}$. Here, $\alpha \equiv \{\alpha_1, \alpha_2, ..., \alpha_r\}$ is the set of actions, $\beta \equiv \{\beta_1, \beta_2, ..., \beta_r\}$ is the set of inputs, $p = \{p_1, .., p_r\}$ is the probability vector of actions and $p(n+1) = T[\alpha(n), \beta(n), p(n)]$ is the learning algorithm. The following algorithm is a sample of linear learning algorithms. We assume that action α_i is selected at time-step *n*.

In case of desirable response from the environment:

$$p_i(n+1) = p_i(n) + a[1 - p_i(n)]$$
$$p_j(n+1) = (1-a)p_j(n) \quad \forall j \ \ j \neq i$$

In case of undesirable response from the environment:

$$p_i(n+1) = (1-b)p_i(n)$$
$$p_j(n+1) = (b/r - 1) + (1-b)p_j(n) \quad \forall j \ \ j \neq i$$

In equation (1) and (2), *a* and *b* are reward and punishment parameters respectively. When *a* and *b* are equal, the algorithm is called L_{RP}, when *b* is much smaller than *a*, the algorithm is $L_{R\varepsilon P}$ and when *b* is zero, the it is called L_{RI}.

IV. INTEGRATING CCGA WITH LEARNING AUTOMATA FOR FUNCTION OPTIMIZATION

In this section, at first, an experiment is presented in order to compare *CCGA-1*, *CCGA-2* and standard *Genetic Algorithm* for function optimization applications. In fact, this experiment is performed just to show to what extent the selection of a right algorithm can be crucial. You can find a more precise and comprehensive comparison of these algorithms in [1]. For this experiment, two functions are considered. The first, called *Rastrigin,* is formulated as:

$$f(\vec{x}) = 3.0n + \sum_{n} x_i^2 - 3.0\cos(2\pi x_i)$$
$$n = 20 \quad \text{and} \quad -5.12 \le x_i \le 5.12 \qquad (3)$$

Because of the fact that variables in (3) are independent, we expect that *CCGA-1* is more appropriate and produces better results that *CCGA-2*. The second function namely *Rosenbrock* is formulated as:

$$f(\vec{x}) = 100(x_1 - x_2)^2 + (1 - x_1)^2 \\ -2.048 \leq x_i \leq 2.048 \tag{4}$$

Equation (4) includes variable multiplication and therefore variables are dependent in the function and it is not possible to optimize the value of the function by optimizing each variable distinctly. Thus, we expect *CCGA-2* has better performance than *CCGA-1*.

The performance of *CCGA-1*, *CCGA-2* and standard *Genetic Algorithm* for the mentioned functions can be compared via Fig. 5. In this figure the best value (best individual) over different times As you can see for *Rastrigin* function *CCGA-1* has a better performance than *CCGA-2* while both are better that standard *Genetic Algorithm*. For *Rosenbrock*, *CCGA-2* has the best performance. It is observable that *CCGA-1* performs even worse than standard *Genetic Algorithm*.

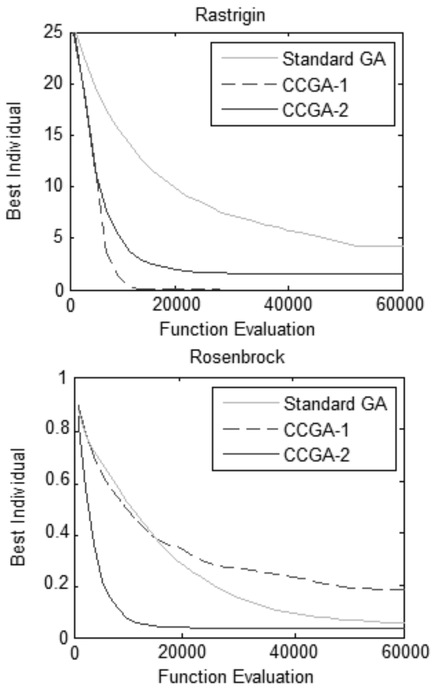

Figure 5. Comparison of CCGA-1, CCGA-2 and standard Genetic Algorithm for Rastrigin and Rosenbrock functions.

To summarize the results we can conclude that finding the right algorithm for function optimization, is a critical step, because of the fact that *CCGA-1* and *CCGA-2* have different performances depend on dependence (independence) of variables. In addition *CCGA-2* has to evaluate individuals twice for each time of fitness calculation and it raises the runtime of the algorithm. Regarding the mentioned facts, we propose a learning optimization algorithm which is not only able to find the optimized value of the variables but also has the capability to distinguish the dependence (independence) of the function variables simultaneously.

By having the mentioned ability, at first, a mixture of *CCGA-1* and *CCGA-2* is used and gradually by learning the dependence (independence) of the variables, the appropriate algorithm will be chosen automatically.

In order to have the ability of learning, we use L_{RI} *LA* and define {*CCGA-1, CCGA-2*} as the action set. Actions, here, mean to choose the appropriate algorithm namely *CCGA-1* and *CCGA-2* as the selected algorithm. The probability vector is initially set to {0.5, 0.5}.

At each iteration of the algorithm, an action is chosen based on the probability vector. The fitness (value) of the function produced by the chosen algorithm (action) is given back to the *LA* as the reward of its action. This procedure continues until the probability vector converges and the likelihood of a certain action becomes greater that a specific threshold. From this point, the procedure of learning will be stopped and the algorithm with the greater likelihood will always be chosen. We call the described approach as *CCGA+LA*. We believe that *CCGA+LA* has a good performance for many functions in general, and specially in cases where the identification of dependence (independence) of the variables is a dilemma.

CCGA+LA is not only useful in terms of produced results but also in terms of time. This is sensible by regarding the fact that *CCGA-2* needs always two evaluations, and it can be reduced to one by *CCGA+LA*. Without using *CCGA+LA* we have to run both algorithms for all values and then choose the best in order to guarantee to have the optimized value. However learning time is an overhead where the dependence (independence) of variables is known.

The coverage of the *LA* probability vector is depicted in Fig. 6. We use *Rastrigin* and *Rosenbrock* functions and set the convergence threshold to 0.9. As you can see, the *LA* converges to *CCGA-1* for *Rastrigin* function after about 360 iterations while for *Rosenbrock* function it converges to *CCGA-2* and it happens after about 750 iterations.

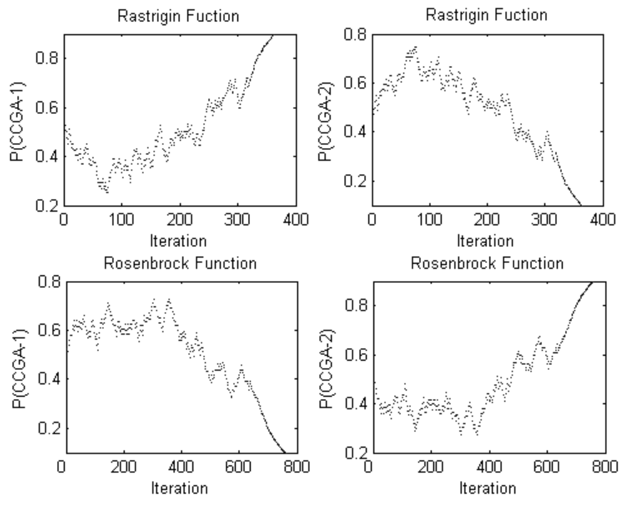

Figure 6. Coverage of Learning Automata for Rastrigin and Rosenbrock functions.

V. CONCLUSION

In this paper, an approach is presented to optimize functions by using *Learning Automata* and *Co-Evolutionary Genetic Algorithm*. In the proposed technique, two versions of *CCGA* are used to optimize the target function and *LA* is employed in order to find the better version of mentioned optimization algorithms. Experimental results show the correctness of learning process. Using this approach, the appropriate optimization algorithm is selected automatically and thereby the need of manual selection can be removed.

REFERENCES

[1] Potter, M. A., De Jong, K. A., "A Cooperative Coevolutionary Approach to Function Optimization". Proceedings of the International Conference on Evolutionary Computation, The 3rd Conference on Parallel Problem Solving from Nature, Springer Verlag, pp.249-257, 1994.

[2] Tan, K. C., Yang, Y. J., Lee, T. H., "A Distributed Cooperative Coevolutionary Algorithm for Multiobjective Optimization". The 2003 Congress on Evolutionary Computation, vol. 4, pp. 2513 – 2520, 2003.

[3] Seredynski, F., Zomaya, A. Y., Bouvry, P., "Function Optimization with Coevolutionary Algorithms". The International Intelligent Information Processing and Web Mining Conference, published by Springer Verlag, 2003.

[4] Nowe, A., Verbeeck, K., Peeters, M., "Learning Automata as a Basis for Multi-Agent Reinforcement Learning", Proceedings of First International Workshop on Learning and Adaptation in Multi-Agent Systems (LAMAS), Utrecht, the Netherlands, 2005, pp. 71-85, 2006.

[5] Shirazi, M.R., Meybodi, M.R. "Application of Learning Automata to Cooperation in Multi-Agent Systems", Proceedings of First International Conference on Information and Knowledge Technology (IKT2003), pp. 338-349, 2003.