Updating Strategy in Compact Genetic Algorithm Using Moving Average Approach

Sunisa Rimcharoen, Daricha Sutivong and Prabhas Chongstitvatana
Department of Computer Engineering
Chulalongkorn University
Bangkok Thailand
suni16@hotmail.com, daricha.s@chula.ac.th, prabhas@chula.ac.th

Abstract— **The Compact Genetic Algorithm (cGA) has a distinct characteristic that it requires almost minimal memory to store candidate solutions. It represents a population structure as a probability distribution over the set of solutions. Although cGA offers many advantages, it has a limitation that hinges on an assumption of the independency between each individual bit. For example, cGA fails to solve a deceptive function or the so called trap function, which is a standard difficult test problem for genetic algorithm. This paper proposes applying a moving average technique to update a probability vector in the compact genetic algorithm. This method requires fewer evaluations and achieves a higher solution quality. The results are compared with the original cGA, sGA, persistent elitist cGA (pe-cGA) and nonpersistent elitist cGA (ne-cGA). The compared results illustrate that the proposed methodology can successfully improve the solution quality by modifying the updating strategy of cGA.**

Keywords—**compact genetic algorithm, moving average, updating strategy**

I. INTRODUCTION

The genetic algorithm (GA) [1, 2] is an optimization algorithm inspired by natural evolution [3]. The GA is performed by creating a population of solutions and uses genetic operators, e.g. reproduction, crossover and mutation to produce offsprings. The solutions are gradually improved by a selection scheme which selects the survivors by their fitness values defined by users. Contrary to the GA, the compact genetic algorithm (cGA) proposed by Harik, Lobo and Goldberg [4] represents the population as a probability distribution over the set of solutions; thus, the whole population needs not to be stored. At each generation, cGA samples individuals according to the probabilities specified in the probability vector. The individuals are evaluated and the probability vector is updated towards the better individual. The cGA mimics the order-one behavior of simple genetic algorithm (sGA) with uniform crossover using a small amount of memory and achieves comparable quality with approximately the same number of fitness evaluations as the sGA. The cGA reduces the size and power requirements of the system by representing the population as a probability vector rather than a collection of bitstrings. Thus, these advantages translate into a flexible hardware implementation. There are several research works that apply cGA to hardware implementation [5-7] because it is easy to implement cGA using common VLSI techniques. Although it has many advantages, the cGA does not provide acceptable solutions to difficult problems such as a deceptive problem or so called trap function, which is a standard difficult test problem for GA. To improve the cGA, Zhou, Meng and Qiu [8] presented an improved cGA using mutation, named mutated by bit compact genetic algorithm (MBBCGA). At each generation, MBBCGA generates only one individual and then mutates this individual bit by bit. Ahn and Ramakrishna [9] proposed persistent elitist compact genetic algorithm (pe-cGA) and nonpersistent elitist compact genetic algorithm (ne-cGA) for solving difficult optimization problems. The pe-cGA keeps the current best solution until a better solution is found. The ne-cGA relaxes selection pressure of the pe-cGA by restricting the length of elite chromosome's inheritance, thereby mitigating the possibility of premature convergence. The ne-cGA further improves the performance of the pe-cGA by avoiding strong elitism that may lead to premature convergence.

In order to improve the ability of cGA, we propose using a moving average technique to modify the updating strategy of cGA. This technique is simple to understand and implement. The concept behind this method is to wait for more information in order to reduce of incorrect decision. The moving average approach can help slowing down the increasing or decreasing of the probability vector. The algorithm implicitly waits to see the trend, which could lead to a better decision strategy.

The paper is organized as follows: Section II reviews the compact genetic algorithm. Section III describes the solution technique using a moving average approach. Section IV contains the test problem and the experiment setup. Experiment results and analysis are provided in Section V. A conclusion is drawn in Section VI.

II. THE COMPACT GENETIC ALGORITHM

The Compact Genetic Algorithm (cGA), proposed by Harik, Lobo and Goldberg [4], is a special class of genetic algorithms. It represents the population as a probability distribution over the set of solutions; thus, the whole population needs not to be stored. At each generation, cGA samples individuals according to the probabilities specified in the probability vector. The individuals are evaluated and the probability vector is updated towards the better individual. Hence, its limitation hinges on the assumption of the independency between each individual bit.

The cGA has an advantage of using a small amount of memory and achieves comparable quality with approximately the same number of fitness evaluations as sGA. The pseudocode of cGA is shown in Fig. 1. The parameters are a population size(n) and a chromosome length(l).

```
1) initialize probability vector
   for i := 1 to l do p[i] := 0.5;
2) generate two individuals from the vector
   a := generate(p);
   b := generate(p);
3) let them compete
   winner, loser := compete(a, b);
4) update the probability vector towards
   the better one
   for i := 1 to l do
      if winner[i] ≠ loser[i] then
         if winner[i] = 1 then p[i] := p[i] + 1/n
                          else p[i] := p[i] – 1/n;
5) check if the vector has converged
   for i := 1 to l do
      if p[i] > 0 and p[i] < 1 then
         return to step 2;
```

Figure 1. Pseudocode of the cGA

First, the probability vector p is initialized to 0.5. Next, the individuals a and b are generated from p. The fitness values are then assigned to a and b. The probability vector is updated towards the better individual. In the population of size n, the updating step size is $1/n$; the probability vector is increased or decreased by this size. The loop is repeated until the vector converges.

Harik, Lobo and Goldberg [4] also propose a modification of the compact genetic algorithm with a higher selection pressure. It simulates a tournament size s. Fig. 2 shows the modification of cGA.

```
1) generate s individuals from the vector
   and store them in S
   for i := 1 to s do S[i] := generate(p);
2) rearrange S so that S[1] is the individual
   with the highest fitness
3) Compare S[1] with the other individuals
   for i := 2 to s do
   begin
         winner, loser := compete(S[1], S[i]);
         update probability vector
         (step 4 of cGA code)
   end
```

Figure 2. Pseudocode of a tournament cGA

III. MODIFYING UPDATING METHOD

This paper proposes applying a moving average technique to update the probability vector in compact genetic algorithm.

A moving average approach is one of the oldest and most popular technical analysis tools for trend identification in financial application. A simple moving average is calculated by adding together the closing prices of a financial instrument over a certain number of days and then dividing the sum by the number of days involved. For example, the five-day average for a stock price would be calculated by taking five days' worth of data, adding them together, and dividing by five. Assume that the following table is the closing prices for the last seven days of market.

TABLE I. EXAMPLE OF STOCK PRICES

day1	day2	day3	day4	day5	day6	day7
1311	1284	1271	1307	1388	1304	1368

To calculate the moving average: take the first five days worth of data and calculate the average value. Then add the prices for day 2-6 together and divide by five. Continue doing this for day 3-7 and so on. From table I, the moving average for day 1-5 is (1311 + 1284 + 1271 + 1307 + 1388) / 5 and the moving average for day 2-6 is (1284 + 1271 + 1307 + 1388 + 1304) / 5 respectively.

We apply the moving average approach to update the probability vector by adding the circular array size M that is a window size of the moving average. This modification replaces a step 4) of the standard cGA shown in Fig. 1. The new step 4) is described in the following pseudocode.

```
4.1) calculate the updating rate (q[i])
     for i := 1 to l do
        if winner[i] ≠ loser[i] then
           if winner[i] = 1 then q[i] := q[i] + 1/n
                            else q[i] := q[i] – 1/n;

4.2) calculate the moving average (movavg)
     for i := 1 to l do
        for m := 1 to M do
           movavg = movavg + q[i][m]
        movavg = movavg / M;
     p[i] = movavg;
```

Figure 3. Pseudocode of modification of cGA

IV. TESTING PROBLEMS

In the experiments, we test the algorithms using two test problems: 100 bit one-max problem and 3x10-bit trap problem.

The data are averaged over 50 runs. All runs end when the vector fully converges, that is all positions are zero or one.

One-max problem is a simple test problem for GA. This problem finds a maximum value in which all bits are one. The fitness value is assigned according to the number of bits that are one in the chromosome. Thus, the maximum value is equal to the chromosome length.

The trap function [10] is a difficult test problem for GA. The general *k*-bit trap function is defined as:

$$F_k(b_0 \ldots b_{k-1}) = \begin{cases} f_{\text{high}} & ; \quad \text{if } u = k \\ f_{\text{low}} - u\dfrac{f_{\text{low}}}{k-1} & ; \quad \text{otherwise} \end{cases} \tag{1}$$

where $b_i \in \{0, 1\}$, $u = \sum_{i=0}^{k-1} b_i$, and $f_{\text{high}} > f_{\text{low}}$. Usually, f_{high} is set at k and f_{low} is set at k-1. The test function $F_{k \times m}$ is defined as:

$$F_{k \times m}(B_0 \ldots B_{m-1}) = \sum_{i=0}^{m-1} F_k(B_i),\ B_i \in \{0,1\}^k \tag{2}$$

This function fools gradient-based optimizers to favor zeroes, but the optimal solution is composed of all ones. The *k* and *m* may vary to produce a number of test functions. For example, 3x5 bit trap function is shown in table II.

We test the algorithm on these two problems using a moving average window size of 2, 3, 4, 5, 10 and 15. The results are compared with the original cGA, sGA, pe-cGA and ne-cGA.

TABLE II. EXAMPLE OF 3X5 BIT TRAP FUNCTION

Ind.	$b_0b_1b_2$	$b_3b_4b_5$	$b_6b_7b_8$	$b_9b_{10}b_{11}$	$b_{12}b_{13}b_{14}$	Fit.
1	111	111	000	111	000	13.0
2	000	000	111	000	111	12.0
3	111	111	011	111	111	12.0
4	111	000	000	111	000	12.0
5	111	001	010	111	111	11.0
6	000	000	000	000	111	11.0
7	111	001	110	111	111	10.0
8	000	000	000	000	000	10.0

V. EXPERIMENT RESULTS AND ANALYSIS

In the experiments, we test the algorithms using two problems: one-max and trap problems. This section presents the experiment results and compares the proposed technique with the cGA, sGA, pe-cGA and ne-cGA in terms of the solution quality and the number of function evaluations. The results of cGA and sGA are from the original paper of cGA [4], and the pe-cGA and ne-cGA results are from [9].

First, the results from the one-max problem are shown. Fig. 4 shows the solution quality, and Fig. 5 shows the number of function evaluations needed to converge. The results from the moving-average cGA (mcGA) are comparable to sGA and cGA in terms of performance and solution quality. From Fig. 4 and Fig. 5, it can be seen that using the same number of function evaluations, the moving-average cGA obtains higher solution quality than the cGA and sGA.

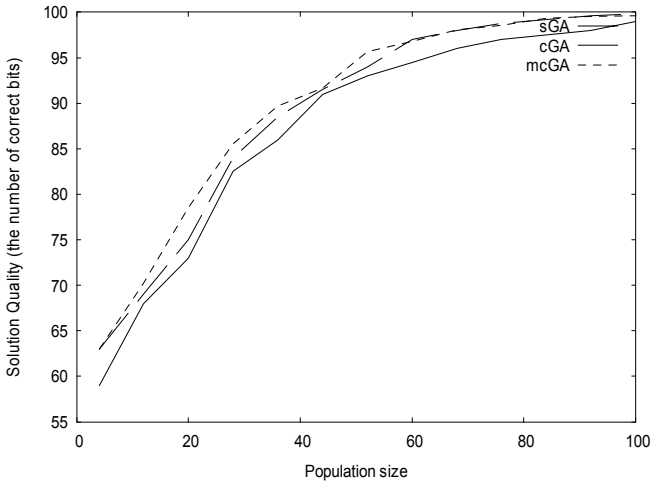

Figure 4. Comparison of solution quality on one-max problem.

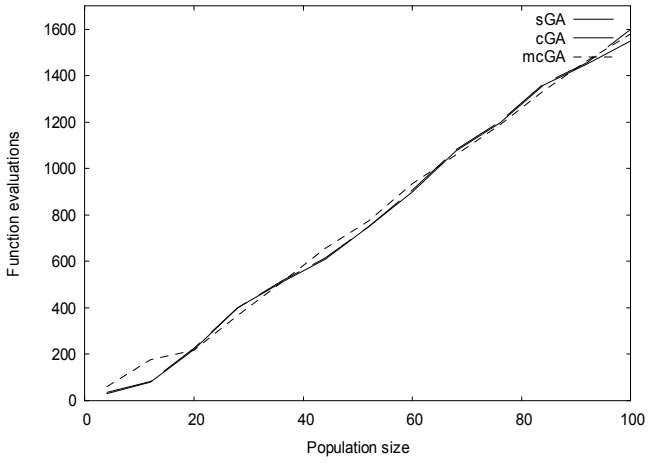

Figure 5. Comparison of the number of function evaluations on one-max problem.

Second, the results from the trap problem are illustrated. Fig. 6 shows the comparison of sGA, cGA, pe-cGA, ne-cGA and cGA with a moving average (mcGA). The graphs illustrate that the moving average cGA outperforms the original cGA in all cases in terms of solution quality and the number of function evaluations. Table III shows the details of window size variations versus the average number of correct building blocks and the average number of function evaluations taken to converge.

TABLE III. PERFORMANCE COMPARISON AT VARIOUS WINDOW SIZES

Window Size	Population Size	Tournament Size 2		Tournament Size 4		Tournament Size 8	
		BBs	Evaluation	BBs	Evaluation	BBs	Evaluation
2	8	3.84	141.20	3.42	70.88	3.00	44.96
	500	4.68	17699.68	6.88	10531.76	7.98	7875.52
	1000	4.22	38320.96	7.16	22483.84	8.16	16952.48
	1500	4.00	56551.88	7.72	32065.92	8.38	23490.56
	2000	3.36	76136.00	8.54	43296.88	9.40	30356.16
	2500	3.80	96455.92	8.28	54014.88	8.88	37940.00
	3000	3.56	118202.44	8.56	62631.84	8.92	46039.68
3	8	3.68	150.36	3.46	69.92	3.26	44.48
	500	4.60	18163.12	7.48	10437.36	7.90	7874.08
	1000	3.90	37549.80	7.48	22438.00	8.02	17321.60
	1500	3.50	56066.40	8.40	32198.48	9.28	22958.08
	2000	3.26	76706.76	8.76	42448.08	9.52	30449.76
	2500	3.72	95621.80	8.98	54092.80	9.58	38102.40
	3000	3.84	116655.84	8.52	63708.40	8.92	44619.68
4	8	3.46	147.48	3.68	73.12	2.92	44.00
	500	4.62	18006.48	7.30	10377.04	8.00	7973.76
	1000	4.34	38481.28	7.26	22858.16	8.06	16927.36
	1500	3.88	57742.28	8.24	32326.72	9.02	23488.16
	2000	4.00	77834.80	8.84	42976.80	9.34	30811.04
	2500	3.80	96903.68	8.88	53437.20	9.68	38340.96
	3000	3.54	116630.28	8.50	64264.80	8.96	44970.56
5	8	3.88	159.00	3.98	76.08	3.34	44.00
	500	4.42	18210.36	7.12	10329.12	7.60	7868.32
	1000	3.84	38444.56	7.42	22876.32	8.14	16613.76
	1500	3.70	56212.72	8.04	32073.76	9.00	23500.96
	2000	3.56	76595.96	8.56	42433.44	9.48	30300.00
	2500	3.60	96629.32	8.98	53151.68	9.74	37524.32
	3000	3.24	113416.52	8.56	64491.20	8.94	45147.84
10	8	3.72	199.84	4.34	97.36	3.50	66.40
	500	4.38	17622.40	7.18	10512.56	7.50	8039.36
	1000	3.80	37699.12	7.58	22382.56	7.98	16888.32
	1500	4.10	57466.28	8.18	32280.96	8.92	23189.12
	2000	3.88	78326.04	8.48	42414.80	9.26	31106.88
	2500	3.64	95888.12	8.84	51769.04	9.70	37622.24
	3000	3.80	119054.36	9.18	63173.76	9.72	44631.84
15	8	4.30	253.76	4.08	125.20	4.06	85.12
	500	4.50	17724.96	6.76	10398.24	7.70	7800.64
	1000	3.82	38106.60	7.30	22432.40	7.94	16826.88
	1500	3.62	57436.36	8.34	32679.28	8.96	23796.32
	2000	3.66	77055.96	8.72	43499.44	9.42	31139.36
	2500	3.94	99288.96	8.86	53999.28	9.80	37801.76
	3000	3.30	117041.40	9.12	63836.88	9.68	45330.24

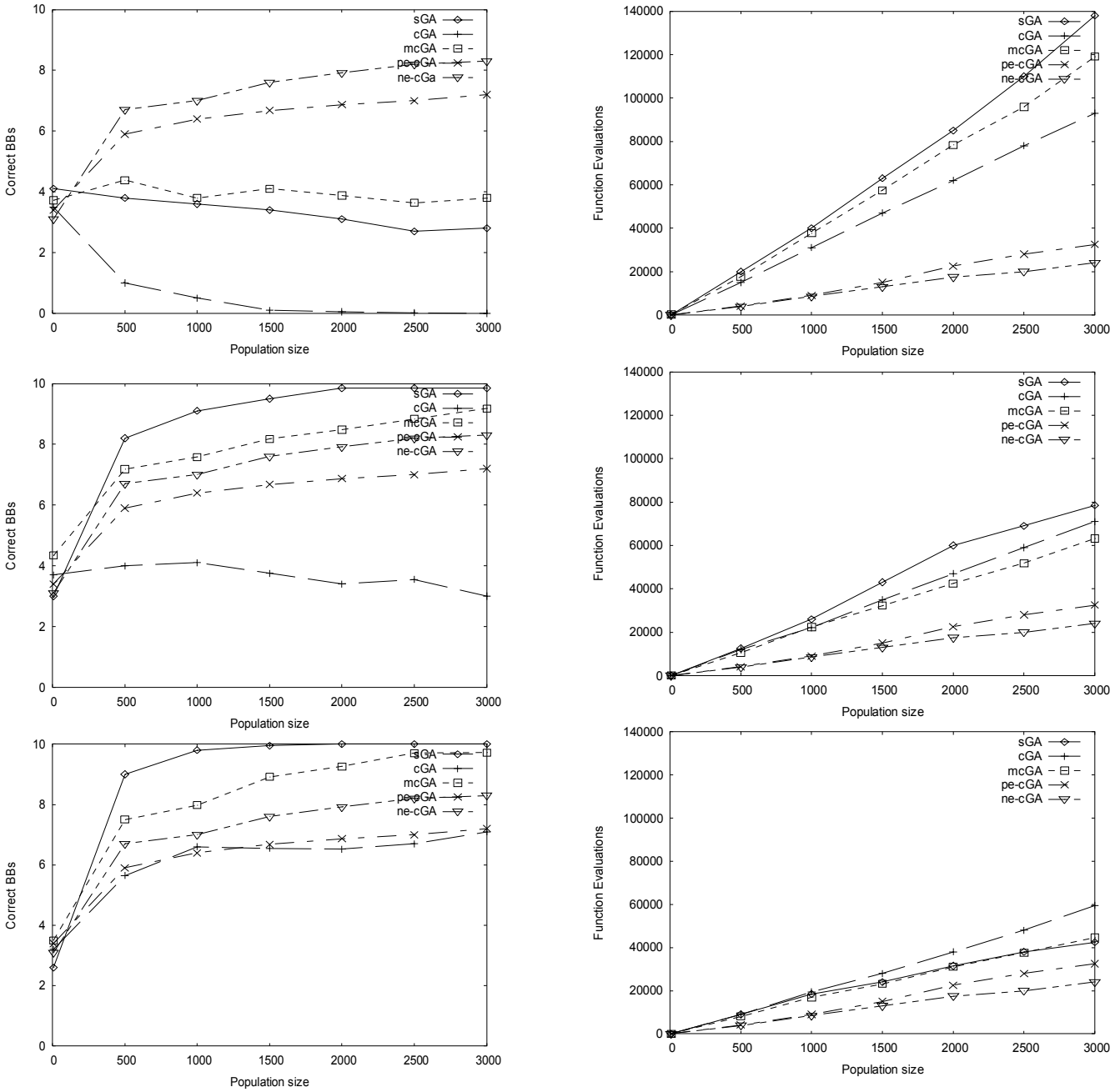

Figure 6. The plots illustrate the comparison of sGA, cGA, pe-cGA, ne-cGA and mcGA. The test problem is ten copies of 3-bit trap function, using selection rate of two (top), four (middle) and eight (bottom). The algorithms were run for population sizes of 8, 500, 1000, 1500, 2000, 2500 and 3000. On the left side, the graphs show the number of correct building blocks (Correct BBs). On the right side, the graphs show the number of function evaluations taken to converge.

We also compare the results with the pe-cGA and ne-cGA. From Fig. 6, the pe-cGA and ne-cGA outperform the others methods in terms of the number of function evaluations (the right side of Fig. 6). Considering the solution quality (the left side of Fig. 6), pe-cGA and ne-cGA also achieve a higher solution quality than the others in case of the tournament size of two (the top left). For more selection pressure, when tournament size are four and eight, (the middle and the bottom graphs on the left side of Fig. 6), the proposed method achieves a higher solution quality than the pe-cGA and ne-cGA, although it requires a higher number of function evaluations.

Finally, we analyze the results form the two test problems. In the one-max problem, the moving average method obtains the result that is comparable to the sGA and the original cGA in terms of solution quality and the number of function evaluations. In the trap problem, the moving average method outperforms the original cGA in all cases. The results emphasize the merit of updating the probability vector at a slower pace, namely using a moving average which normally produces a more moderate update than the original cGA. A minor adjustment to vector updating technique allows the algorithm to achieve a higher solution quality.

The moving average window size (M) has minimal effect on the solution quality and the number of function evaluations. Nonetheless, we observe that a larger window size generally leads to a slightly higher solution quality, except when the population size is large. Overall, the performance will not be improved significantly by varying the window size. However, choosing a suitable window size can lead to a smooth increase in solution quality, as the probability vector is updated with a smooth sequence of values. From the experiments, it can be observed that a suitable moving average window size for the 3x10 trap problem is 10. As may be expected, the solution quality increases when the tournament size increases. We also notice that the number of function evaluations needed to converge decreases as the tournament size increases regardless of the window size.

VI. CONCLUSIONS

This paper proposes using a moving average approach to modify the updating strategy of cGA. The technique is simple to understand and implement. A minor adjustment to the vector updating process allows the algorithm to achieve a higher solution quality. The experiment results show that the proposed method can improve the solution quality with a smaller number of function evaluations than the original cGA. The study yields an insight that waiting for more information in order to better capture the probability trend can help the algorithm make a better decision. For future extension, we will model the look-ahead decision strategy that can assist the algorithm in deciding whether to increase or decrease the probability vector under the uncertainty of possible trap.

REFERENCES

[1] D. E. Goldberg, Genetic algorithms in search, optimization and machine learning, Addison-Wesley, 1989.

[2] J. H. Holland, Adaptation in natural and artificial systems, University of Michigan Press, 1975.

[3] C. Darwin, On the Origin of Species by means of Natural Selection, John Murray, 1859.

[4] G. R. Harik, F. G. Lobo and D. E. Goldberg, The compact genetic algorithm, in IEEE Transactions on Evolutionary Computation, 1999, Vol. 3, No. 4, 287-297.

[5] C. Aporntewan and P. Chongstitvatana, A hardware implementation of the compact genetic algorithm, in Proceedings of the 2001 IEEE Congress on Evolutionary Computation, 2001.

[6] J. Gallagher and S. Vigraham, A modified compact genetic algorithm for the intrinsic evolution of continuous time recurrent neural networks, in Proceeding of the 2002 Genetic and Evolutionary Computation Conference, 2002.

[7] J. Gallagher, S. Vigraham and G. Kramer, A family of compact genetic algorithms for intrinsic evolvable hardware, in IEEE Transactions on Evolutionary Computation, 2004.

[8] C. Zhou, K. Meng and Z. Qiu, Compact genetic algorithm mutated by bit, in Proceeding of 4th World Congress on Intelligent Control and Automation, 2002.

[9] C. W. Ahn and R. S. Ramakrishna, Elitism-based compact genetic algorithms, in IEEE Transactions on Evolutionary Computation, 2003.

[10] D. H. Ackley, A connectionist machine for genetic hillclimbing, Kluwer Academic Publishers, Boston, MA, 1987.