Mutation in Compressed Encoding in Estimation of Distribution Algorithm

Orawan Watchanupaporn, Worasait Suwannik
Department of Computer Science
Kasetsart University
Bangkok, Thailand
orawan.liu@gmail.com, worasait.suwannik@gmail.com

Prabhas Chongstitvatana
Department of Computer Engineering
Chulalongkorn University
Bangkok, Thailand
prabhas@chula.ac.th

***Abstract*—Estimation of Distribution Algorithm (EDA) is a new kind of evolutionary algorithm. However, it does not use evolutionary operators such as crossover and mutation. In this paper, we investigate how mutation has an effect on the performance of EDA, more specifically, compact genetic algorithm (cGA) and LZWcGA. The result shows that cGA performs poorly with mutation while LZWcGA's performance is improved by mutation. We also present an analysis of mutation in both algorithms.**

Mutation; LZW; Estimation of Distribution Algorithm

I. INTRODUCTION

Genetic Algorithm (GA) is an algorithm that solves problems by imitating a process of natural evolution [1]. In GA, a candidate solution is encoded in a binary string called an individual or a chromosome. A collection of individuals in one generation is called a population. The algorithm selects highly fit individuals from the population. The new generation is created by reproducing, recombining and mutating the selected individuals. The process is repeated until the solution is found.

Estimation of Distribution Algorithm (EDA) is an improvement over GA. Many steps in GA contain unprincipled randomness such as mutation and crossover. For example, the crossing site of crossover is selected randomly. Therefore, EDA replaces those processes by probabilistic modeling of highly fit individuals and generating the next generation from the model [2]. Various types of EDA assume different dependencies between different positions in the binary string. The univariate EDA assumes all bits are independent. Univariate EDA includes compact GA (cGA), PBIL, and UMDA. Bivariate EDA assumes dependency among pairs of bits. Bivariate EDA includes MIMIC and BMDA. Multivariate EDA assumes multiple dependencies between bits. Multivariate EDA includes BOA and ECGA.

Compressed chromosome encoding is proposed to enable evolutionary algorithm to solve large scale problems [3][4]. For example, LZW encoding in Genetic Algorithm can solve one-million-bit problem. The individual is in the compressed form and has to be decompressed before the fitness evaluation. The advantage of this approach is low memory requirement.

In GA, mutation helps maintaining diversity. However, mutation has fewer, if not none, roles in EDA. Handa includes mutation in EDA and show the effectiveness of his method [5]. Zhang et al. used guided mutation which generates offspring using information from both probabilistic model and a group of best individuals that have been found during the search [6]. This paper studies and analyzes mutation in compressed encoding in estimation of distribution algorithm.

II. COMPACT GENETIC ALGORITHM

Harik et al. [7] introduced a compact genetic algorithm (cGA). The advantage of cGA is low memory consumption. cGA consumes less memory than traditional Genetic Algorithm because the algorithm uses a single probability vector to represent the whole GA population. The probability vector requires only $l\times(\log_2 n)$ bits to represent a population of size n, where l is the length of each bit string. On the other hand, a standard GA requires $l\times n$ bits to store a population.

Figure 1 shows the cGA algorithm. The first step is to initialize each item in the probability vector to 0.5. The value 0.5 means that each bit in the chromosome has equal chance to be 1 or 0. Then the algorithm randomly generates two individuals from the probability vector. Next, both individuals are evaluated for the fitness value. The individual with higher fitness score is called the winner, whereas the one with the lower score is called the loser. For each bit, the probability vector is updated by the following rules.

• Increase the probability value by $1/n$, if the winner bit = 1 and the loser bit = 0.

• Decrease the probability value by $1/n$, if the winner bit = 0 and the loser bit = 1.

The probability update step imitates the uniform crossover in the standard GA. The update rule of cGA assumes no dependency between any bits. Thus, cGA is classified as univariate EDA.

The last step of cGA is to check whether the probability vector has been converged. If not, the evolutionary process is repeated starting from step 2 through step 5. Notice that there is no crossover and mutation in cGA.

cGA is applied to solve large-scale problems. Watchanupaporn et al. used compressed encoding with cGA to solve 128, 256 and 512-bit One-Max problem and against 60, 120 and 240-bit Royal Road problem [3]. Sastry et al. ran cGA

on a cluster of computers to solve a billion-bit noisy OneMax problem [8]. The problem is more difficult than OneMax problem because noise disrupts the evolutionary search.

```
Parameters
    n : population size
    l : chromosome length
1) Initialize probability vector p.
    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 := evaluate(a, b)
4) Update the probability vector towards the better
 individual.
    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
6) p represents the final solution.
```

Figure 1. Compact Genetic Algorithm (cGA) pseudo code

III. LZW Compact Genetic Algorithm

Lempel-Ziv-Welch Algorithm (LZW) is a lossless dictionary-based data compression/decompression algorithm [9]. The input of the compression algorithm is a character string. The output of the compression algorithm (also the input of the decompression algorithm) is an array of integer codes. The output of the decompression algorithm is the original character string.

The compression/decompression algorithms start with a dictionary which the number of entries is equal to the number of characters. Each entry contains one character. For example, when using LZW to compress/decompress an English text, the dictionary is initialized with all English characters and symbols. However, when LZW is used to compress or decompress a binary chromosome in GA, the dictionary is initialized with the number 0 and 1. During the compression, the algorithm dynamically expands the dictionary and outputs codes that refer to strings in the dictionary. Normally, the number of bits of the code is less than that of the variable length string in the dictionary. Data is compressed when the algorithm replaces the whole string with its code.

To use LZW compressed encoding with cGA, we add a decoding and decompressing step after step 2. The binary chromosome is decoded to an array of integers. After that, the array is decompressed to a binary string, which might be longer than the original binary chromosome. Figure 2 shows where the new step D) is inserted. Please note that LZWcGA evolves a direct representation of an individual as a "compressed" string. There is no compression step involved in LZWcGA. The pseudo code in step 2 (generate) of LZWcGA and cGA is different. In cGA, an invidual is created as a binary string. In LZWcGA, an individual is a binary string in "compressed" form. The mutation in LZWcGA is applied directly to this representation.

```
...
2) Generate two individuals from the vector
    a := generate(p)
    b := generate(p)
D) Decode and decompress both individuals
    a := decompress(decode(a))
    b := decompress(decode(b))
3) Let them compete.
    winner, loser := evaluate(a, b);
...
```

Figure 2. LZWcGA pseudo code

LZW chromosome encoding can be applied to various EDAs such as cGA, MIMIC, and BOA. Adding LZW encoding to existing EDA is easy. EDA algorithm does not have to be modified. Rather, the fitness evaluation has to be modified by adding decoding and decompressing at the beginning. A binary string is decoded to an array of integers using Gray decoding. Then, the integer array is decompressed to a binary string using LZW decompression algorithm. Finally, the binary string from the previous step is evaluated and its fitness is returned to EDA. To EDA's point of view, it evolves a binary string. It does not know that it is evolving a compressed encoding chromosome.

IV. Mutation

In LZWcGA, mutation occurs after individuals are generated. Each bit has a probability to be mutated equals to the mutation rate. If the bit is mutated, then its value is flipped from 0 to 1 or from 1 to 0. An LZW binary chromosome is mutated before it is decoded and decompressed. Figure 3 shows where the new step M) is inserted.

```
...
2) Generate two individuals from the vector
    a := generate(p)
    b := generate(p)
M) Mutate both individuals
    a := mutate(a)
    b := mutate(b)
D) Decode and decompress both individuals
    a := decompress(decode(a))
    b := decompress(decode(b))

...
```

Figure 3. LZWcGA with mutation pseudo code

V. BENCHMARK PROBLEMS

We use the synthetic problems to assess the strengths and weaknesses of LZW encoding. The advantage of using a synthetic problem is that its structures (i.e., relationship between variables) are known. Thus, we can assume that if an algorithm can solve the problem, it can also solve a class of problems that have the same structure. Moreover, an algorithm that can solve problems with more complex structures is more sophisticated and is likely to solve a problem with simpler structure.

A. Trap Problem

In Trap problem, an individual composes of several blocks. Each of the blocks is evaluated by the trap functions. The Trap function can fool the gradient-based optimizers to favor zeros, but the optimal solution is composed of all ones. It is a fundamental unit for designing test functions that resist hill-climbing algorithms.

A k-bit trap function is defined as:

$$F(\vec{x}) = \begin{cases} f_{high} & ;if\ u = k \\ f_{low} - (u \times f_{low})/(k-1) & ;otherwise \end{cases} \quad (1)$$

where $\vec{x} = \{0,1\}, u = \sum_{i=1}^{k} x_i$ and $f_{high} > f_{low}$. Usually, f_{high} is set at k and f_{low} is set at k-1.

The Trap problem can be decomposed to several Trap functions. The problem, denoted by $F_{m \times k}$, is defined as:

$$f_{m \times k}(K_1 \cdots K_m) = \sum_{i=1}^{m} F_k(K_i), K_i \in \{0,1\}^k \quad (2)$$

The m and k are varied to produce a number of test functions.

B. Four-Peak problem

The Four-Peak problem [10] has two global maxima and two suboptimal local optima. The problem is defined as follows.

$$f(\vec{X}, T) = \max[tail(0, \vec{X})] + R(\vec{X}, T) \quad (3)$$

where

$$tail(b, \vec{X}) = number\ of\ trailing\ b's\ in\ \vec{X} \quad (4)$$

$$head(b, \vec{X}) = number\ of\ leading\ b's\ in\ \vec{X} \quad (5)$$

$$R(\vec{X}, T) = \begin{cases} N & if\ tail(0, \vec{X}) > T \text{ and } head(1, \vec{X}) > T \\ 0 & otherwise \end{cases} \quad (6)$$

For a 10-bit problem, the global optimums are 1100000000 and 111111100. Their fitness values are 18. The local optimums are chromosomes with all 1's and all 0's. For a 800-bit problem, the optimum fitness value is 1519.

C. Six-Peak problem

The Six-Peak problem [10] is harder than the Four-Peak problem even it has two more global maxima than the Four-Peak problem. The definition of the problem is similar to that of the Four-Peak problem but the definition of $R(X, T)$ is changed as follows.

$$R(\vec{X}, T) = \begin{cases} N & if\ tail(0, \vec{X}) > T \text{ and } head(1, \vec{X}) > T\ tail(1, x) > T \text{ and } head(0, x) > T \\ 0 & otherwise \end{cases} \quad (7)$$

The optimal solutions of this problem are the same Four-Peak problem and there are two additional global maxima. For a 10-bit problem. The two additional global optimums are chromosome 0000000011 and 0011111111. For the same problem size, the optimum fitness value equal Four-Peak problem.

VI. RESULTS

We compare the performance of cGA and LZWcGA at different mutation rates. Table I shows the experimental parameters. The length of binary LZW chromosome is equal to the problem size. Before a fitness evaluation, the compressed chromosome is decoded and decompressed with LZW decompression algorithm. The length of the decompressed chromosome is varied. If the length is more than the size of the problem size, the excess bits are discarded. If the length is less than the problem size, LZWCGA will evaluate the fitness of available bits. All experimental results are the average performance obtained from 30 runs. Table II shows the average best fitness when using the algorithms to solve Trap, 4-Peak, and 6-Peak problems respectively.

LZWcGA outperforms cGA for all problems that we tested. Mutation deteriorates the performance of cGA while it can improve the performance of LZWcGA. The higher the mutation rate results in the poorer the performance of cGA. However, for LZWcGA, mutation can improve its performance. Among the mutation rates that we experiment, the rate 0.05 gives the best performance.

TABLE I. EXPERIMENTAL PARAMETERS

Parameter	Value
Population size	1000
Problem size	800
LZW chromosome length	800
Maximum evaluations	50,000
Mutation rate	0.00, 0.05, 0.10, 0.15

TABLE II. AVERAGE BEST FITNESS

Problem	Algorithm	Mutation Rate			
		0.00	0.05	0.10	0.15
Trap	cGA	575	553	526	498
	LZWcGA	768	795	791	787
4-Peak	cGA	50	31	26	23
	LZWcGA	1387	1449	1420	1395
6-Peak	cGA	47	28	25	22
	LZWcGA	1463	1498	1488	1488

VII. ANALYSIS

From the experimental result, mutation deteriorates the performance of cGA but improves the performance of LZWcGA. We hypothesize that the reason that cGA does not work well is because the mutation destroys more building block than creating a building block. We test our hypothesis with Trap problem because its building blocks are known. The building blocks in Trap problem have the same size and are lined-up consecutively.

To prove our hypothesis, for every mutation occurs during the evolution, we count the number of times that a new building block is created and compare it to the number of times that a build block is destroyed by mutation. If the mutation has a detrimental effect then the first number should be lower than the second number. However, the result shows that mutation constructs more building blocks than destroys them.

To observe the effect, the ratio of construction and destruction is defined. An about-to-be building blocks (BB2B) is defined as a part of string that is different from a true building block by one bit. In Trap problem, BB2B is a block with one 0's. A construction ration is the number of block that becomes the building block divided by the total number of about-to-be building block (BB2B). A destruction ratio is the number of building block that was destroyed divided by total number of building block. From the experiment, there are a lot of about-to-be building blocks (BB2B) than the building block. It is likely that mutation creates more building blocks than destroying it because there is a higher chance that a new building block will be created.

Table III shows the analysis result. In cGA, the number of building block per BB2B is very low. Even worse, it has very low construction ratio compares to the destruction ratio. However, in LZWcGA, the construction ratio is much higher than the destruction ratio. Note that in the case of LZWcGA, we obtained the ratio by counting the building blocks in a decompressed chromosome. The analysis is performed on an 800-bit problem. The mutation rate is 0.05. The result is an average over 30 runs.

At the mutation rate 0.05, the ratio of building block per about-to-be building block for cGA and LZWcGA is 0.148 and 1.228. However, when no mutation is used the ratio for cGA and LZWcGA is 0.148 and 0.921. Notice that mutation in LZWcGA helps increase the BB to BB2B ratio.

In Table III, the construction ratio is 0.048 while the mutation rate is 0.050. If we increased the maximum number of evaluation to a very large number, the construction ratio of cGA will be equal to the mutation rate. This is because in the ratio is equal to the probability that the only 0 in the block will be changed to 1, which is the mutation rate.

TABLE III. ANALYSIS RESULTS

Ratio	cGA	LZWcGA
Construction	0.048	0.787
Destruction	0.226	0.188
BB per BB2B	0.148	1.228

VIII. CONCLUSIONS

This paper investigates the impact of mutation to cGA and LZWcGA. Both algorithms are univariate EDA. However, mutation affects them differently. cGA's performance is worsened by mutation while mutation can improve the performance of LZWcGA. The analysis shows that, in the case of cGA with Trap problem, mutation has higher building block destruction ratio than the construction ratio. However, in LZWcGA, the same mutation method gives higher building block construction ratio and the destruction ratio.

The future work might incorporate mutation to various EDA such as MIMIC and BOA to see how mutation effect the performance of compressed encoding.

REFERENCES

[1] D.E. Goldberg, Genetic Algorithms in Search, Optimization, and Machine Learning, Addison- Wesley, 1989.

[2] T.K. Paul and H. Iba, "Linear and Combinatorial Optimizations by Estimation of Distribution Algorithms," Proceedings of 9th MPS Symposium on Evolutionary Computation, IPSJ, 2002.

[3] O. Watchanupaporn, N. Soonthornphisaj, and W. Suwannik, "A Performance Analysis of Compressed Compact Genetic Algorithm," ECTI Transactions on Computer and Information Technology, vol. 2, no. 1, 2006, pp. 16-24.

[4] N. Kunasol, W. Suwannik, and P. Chongstitvatana, "Solving One-Million-Bit Problems Using LZWGA," Proceedings of International Symposium on Communications and Information Technologies (ISCIT), 2006, pp. 32-36.

[5] H. Handa, "Estimation of Distribution Algorithms with Mutation," Evolutinary Computation in Combinatorial Optimization, 2005.

[6] Q. Zhang, J. Sun, and E. Tsang, "Combinations of Estimation of Distribution Algorithms and Other Techniques," International Journal of Automation and Computing, 2007, pp. 273-280.

[7] G.R. Harik, F.G. Lobo, and D.E. Goldberg, "The Compact Genetic Algorithm," IEEE Transaction on Evolutionary Computation, vol. 3, no. 4, 1999, pp. 287-297.

[8] K. Sastry, D.E. Goldberg, and X. Llorà, "Towards billion bit optimization via parallel estimation of distribution algorithm," Genetic and Evolutionary Computation Conference, 2007, pp. 577-584.

[9] T.A. Welch, "A Technique for High-Performance Data Compression," IEEE Computer, vol. 17, no. 6, 1984, pp. 8-19.

[10] J.S. De Bonet, C.L. Isbell, and P. Viola, "MIMIC: Finding Optima by Estimating Probability Densities," Advances in Neural Information Processing Systems, vol. 9, MIT Press, Cambridge, 1997, pp. 424-430