THESIS APPROVAL

GRADUATE SCHOOL, KASETSART UNIVERSITY

Doctor of Philosophy (Computer Science)

DEGREE

Computer Science	Computer Science
FIELD	**DEPARTMENT**

TITLE: Compressed Encoding in Differential Evolution

NAME: Miss Orawan Watchanupaporn

THIS THESIS HAS BEEN ACCEPTED BY

THESIS ADVISOR

(Assistant Professor Worasait Suwannik, Ph.D.)

DEPARTMENT HEAD

(Assistant Professor Sirikorn Channual, M.S.)

APPROVED BY THE GRADUATE SCHOOL ON _____

DEAN

(Associate Professor Gunjana Theeragool, D.Agr.)

THESIS

COMPRESSED ENCODING IN DIFFERENTIAL EVOLUTION

ORAWAN WATCHANUPAPORN

A Thesis Submitted in Partial Fulfillment of
the Requirements for the Degree of
Doctor of Philosophy (Computer Science)
Graduate School, Kasetsart University
2013

Orawan Watchanupaporn 2013: Compressed Encoding in Differential Evolution. Doctor of Philosophy (Computer Science), Major Field: Computer Science, Department of Computer Science. Thesis Advisor: Assistant Professor Worasait Suwannik, Ph.D. 134 pages.

Differential Evolution (DE) is an effective continuous real value optimizer. In this study, we apply DE to discrete optimization problems by using Lempel-Ziv-Welch (LZW) and Arithmetic Coding (AC) compression. The resulting algorithms are called LZWDE and ACDE. LZWDE applies DE to discrete problems by converting a real chromosome to an integer chromosome and then decompressing it to a binary chromosome using LZW algorithm. Experimental result shows that this approach is better than the previous binary DE scheme and the evolution time is very fast. Analysis result shows that the fitness landscape of LZW encoding is less complex than the original encoding for each test problem. ACDE applies DE to discrete optimization problems by using Arithmetic Coding compression and local search algorithm, which are Hill-climbing and Simulated Annealing. The test functions include the random-version of widely used benchmark problems. We compare the performance of each algorithm based on the quality of solution and the running time. Experimental results indicate that ACDE algorithm outperforms Bayesian Optimization Algorithm (BOA), which is a sophisticated discrete optimizer, in both solution quality and running time.

_____ _____ __ / __ / __

Student's signature Thesis Advisor's signature

ACKNOWLEDGEMENTS

First of all, I would like to gratefully thank my advisor, Asst. Prof. Dr. Worasait Suwannik, for advice, encouragement, devotion, and valuable suggestions for conducting this research until I completed this thesis, without which I would never have succeeded. He has been supportive since the days of my master studies.

I would like to express my gratitude to my thesis committee, Assoc. Prof. Dr. Nuanwan Soonthornphisaj, who is also my advisor during master studies, for her kind advice, dedication, and support. I also would like to express my deep appreciation to another thesis committee, Prof. Dr. Prabhas Chongstitvatana, for his kindness and valuable suggestions. I would like to extend my grateful thank to Asst. Prof. Dr. Arnon Rungsawang for his kind guidance.

I am especially grateful to my parents and brothers for always supporting and being there for me. I am blessed to have the love and caring of my family.

Orawan Watchanupaporn

May 2013

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF TABLES (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF FIGURES (Continued)

COMPRESSED ENCODING IN DIFFERENTIAL EVOLUTION

INTRODUCTION

Evolutionary algorithms (EAs) are probabilistic optimization methods based on the model of natural evolution. A common characteristic of this type of algorithms is population based. An individual in a population is evaluated and selected based on its fitness value. The fitness value indicates how well an individual is for a specific optimization problem. The selected individuals will be likely to survive to the next generation or they can generate offspring that will be put into the next generation. The offspring can be generated using a nature-inspired mechanism such as reproduction or crossover. The process of fitness evaluation, selection, and creating a next generation is repeated until the solution is found or another termination criterion is met.

EA differs by the representation of an individual. For example, Genetic algorithm (GA), proposed by Holland, uses a binary string to represent an individual (Holland, 1975; Goldberg, 1989) an individual in Genetic Programming is a program tree (Koza, 1990, 1992); an individual in Evolutionary Strategy (ES) is a real value vector (Beyer and Schwefel, 2002).

Different representations have direct affects on the genetic operators. For example, mutation in GA is simple bit inversion while in GP a new subtree is grown. In addition, the representation also affects the foundation of theoretical establishment. For instance, a schema theorem that proves the effectiveness of GA cannot be applied to GP. Finally, and most importantly, the representation affects the class of the problems that the algorithm can solve efficiently. GA and its descendant, Estimation of Distribution Algorithm (EDA), are doing well in binary optimization problems. Various benchmarks have been developed to show its strength over other approaches. For example, Trap problem (Ackley, 1987) and Long Path problem (Horn, 1994),

which are designed to resist a hill climbing search algorithm, can be solved by several EDAs.

While GA and EDA are normally used for binary optimization, ES and Differential Evolution (DE), are popular real value optimizers. ES was proposed by Rechenberg in 1960s. DE was introduced by Price and Storn in 1990s. DE is very compact. The core of the algorithm can be implemented in less than 20 lines of C code, which is available on-line (Price and Storn, 2012). DE is implemented in Mathematica as NMinimize (Wolfram MathWorld, 2012) and also in Matlab. DE is very fast and efficient. It was ranked the third in the First International Contest on Evolutionary Optimization in 1996. However, it is more robust than the first and the second place optimizers (Price *et al*., 2005).

The speed and effectiveness of DE in continuous value optimization appeals many researchers (Vesterstrøm and Thomsen, 2004; Storn and Price, 1997; Muelas *et al*., 2009; Qin *et al*., 2009). However, for discrete optimization, there are a few works that investigate DE's effectiveness (Gong and Tuson, 2007; Lichtblau, 2012). This study presents several methods for applying DE to discrete optimization problems. The first approach is simple real to binary conversion scheme, which we will refer as SDE. This scheme is straightforward but quite wasteful because one variable in DE, which is normally 64-bit long, represents only one bit. Thus, the second approach combines conversion and decompression to create a binary string. We used Lempel-Ziv-Welch (LZW) compression algorithm and Arithmetic Coding (AC) compression algorithm. For example, in LZW approach, a real vector is converted to an integer array. After that, the array is sent to LZW decompression algorithm. LZW will output a binary string which will be used in a discrete problem fitness evaluation.

Compressed encoding is compact and enables evolutionary algorithm to solve very large problems. For example, LZW encoding in Genetic Algorithm can solve one-million-bit problems (Kunasol *et al.*, 2006; Suwannik and Chongstitvatana, 2008). Another advantage of this approach is low memory requirement which leads to faster running time per generation due to less data transfer. However, the

disadvantage of compressed encoding is a bias toward high regularity solution. To show the existence of the bias, Chamlamai and Suwannik (2007) created a random version of an existing benchmark. The experimental result shows that the performance of LZW encoding is decreased in the random version while the performance of uncompressed encoding remains the same. To solve the random-version of a benchmark problem, we combine Arithmetic Coding, which is another compression algorithm, with DE. Moreover, we add two local search algorithms which are simple Hill Climbing (HC) and Simulated Annealing (SA) (Kirkpatrick *et al.*, 1983) to our search method. The result is compared against the Bayesian Optimization Algorithm (BOA) (Pelikan *et al*., 1999), which is one of the best EDA.

Compressed encoding not only is a bridge between continuous optimization and discrete optimization but it also transforms a fitness landscape. Analysis is conducted to understand the transformation. For binary problems, we can visualize a fitness landscape by enumerating all possible binary chromosomes and calculate its fitness and distance to the solution. Enumerating all possible chromosomes is possible for a binary optimization problem because there are finite amount of chromosomes given a fixed length binary string (e.g., a problem size n bit has 2^n possible chromosomes). In the contrast, a single real-value in a DE vector, in theory, can have infinitely uncountable possible values. Therefore enumeration is not possible in DE. Instead, we explore the fitness landscape using random walk (Uludag and Uyar, 2009). While an analysis procedure performs random walk, a fitness and distance to a solution is recorded. Each step of random walk imitates a trial vector generation process in DE.

OBJECTIVES

1. To apply Differential Evolution to discrete optimization problems. DE is a fast and efficient real value optimizer. An equivalent discrete optimizer might be obtained by adapting DE to discrete optimization problem.

2. To solve random problems using compressed encoding. Even with an extra processing step, the running time per generation of compressed encoding can be shorter than that of an original encoding. However, compressed encoding does not perform well on the random version of existing benchmark. An ability to solve such problem together with faster processing would make compressed encoding more interesting.

3. To analyze compressed encoding in Differential Evolution. The original motivation for using compressed encoding is to reduce search space. However, compressed encoding is complex. It transforms the problem in some obscure way. This work attempts at a better understanding of the compressed encoding.

LITERATURE REVIEW

Genetic Algorithm

Genetic Algorithm (GA) is an algorithm inspired by natural evolution (Holland, 1975; Mitchell, 1998). The flowchart of GA is shown in Figure 1. In order to solve a problem, a candidate solution is encoded as a binary string chromosome. A group of chromosomes is called a population. GA creates the first generation of population by randomly generating binary chromosomes. Then, it evolves binary chromosomes using the process of selection, reproduction, mutation, and recombination. The algorithm randomly selects highly fit chromosomes for creating the next generation. The selected chromosomes appear in the next generation by the process of reproduction. Mutation randomly changes the value of chromosome. Recombination or crossover creates new chromosome by combining, at a random point, two highly fit chromosomes together. These processes are repeated until the optimal solution is found or the maximum generation is reached. Notice that most processes in the algorithm contain randomness.

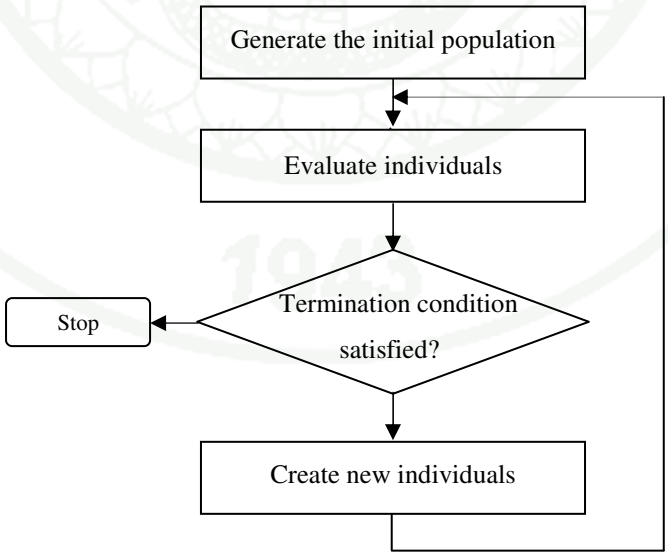


Figure 1 GA flowchart

Estimation of Distribution Algorithm

Estimation of Distribution Algorithm (EDA) is a new approach in the field of evolutionary computation (Mühlenbein and Paaß, 1996; Larrañaga and Lozano, 2002). EDA solves problems using a population of chromosomes similar to GA. However, unlike GA, which contains randomness in the most of steps, EDA evolves chromosomes in a more principled manner. As a result, EDA can solve more complex problems and scales better than GA.

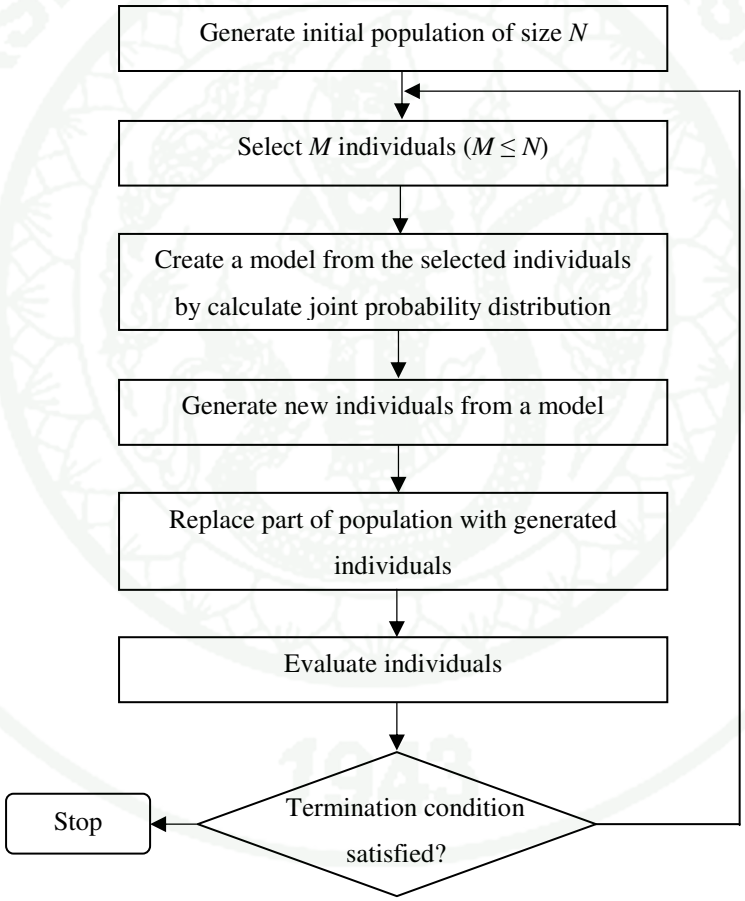


Figure 2 EDA flowchart

The flowchart of EDA is shown in Figure 2. The first step is to randomly create a population. After that, EDA randomly selects highly fit individuals from the population. Then, it models highly-fit individuals in each generation by assuming a particular probability distribution. Next, it generates new individuals from the model and puts them to the population. Modeling and generating can avoid the disruption of partial solution resulted from genetic operations such as crossover and mutation. The last step is to check whether the solution is found. If not, the evolutionary process is continued from step 2 to step 7. Figure 3 shows the EDA process.

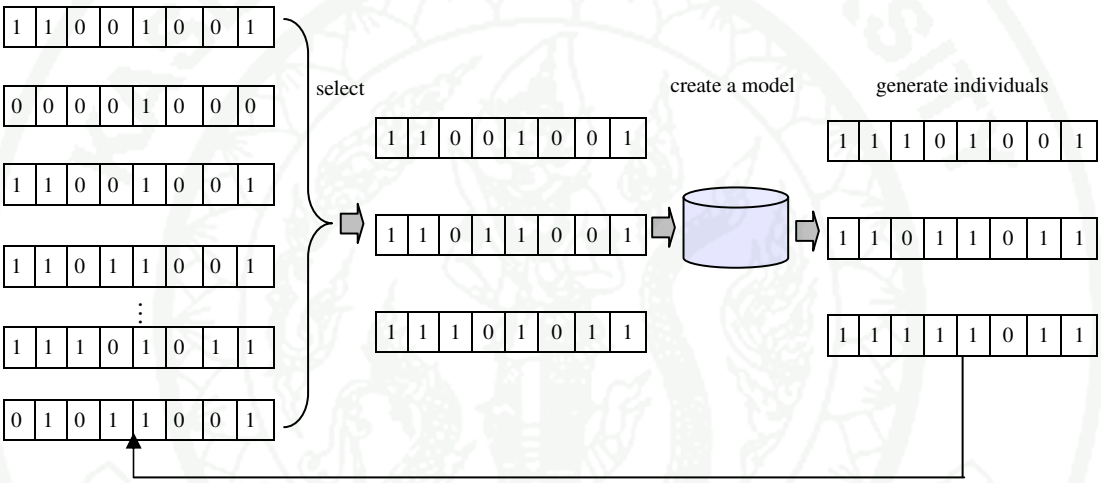


Figure 3 EDA process

Varieties of EDA can be classified based on an assumption about relationship among variables in the modeling step (Paul and Iba, 2002) (see Figure 4). A variable is a position in a chromosome. The simplest EDA is Univariate EDA, which assumes no relationship among variables. Univariate EDAs include Compact Genetic Algorithm (cGA) (Harik *et al.*, 1999), Univariate marginal distribution algorithm (UMDA) (Mühlenbein and Paaß, 1996), and Population-based incremental learning (PBIL) (Baluja, 1994). A more complex EDA is Bivariate EDA, which assumes relationship between two variables. Examples are Mutual Information Maximization for Input Clustering (MIMIC) (De Bonet *et al.*, 1997), Bivariate Marginal Distribution Algorithm (BMDA) (Pelikan and Mühlenbein, 1999) and Combining Optimizers with Mutual Information Trees (COMIT) (Baluja and Davies, 1997). The most complex

EDA is Multivariate EDA, which assumes relationship between any numbers of variables. For example, Bayesian Optimization Algorithm (BOA) (Pelikan *et al.*, 1999), Extended Compact Genetic Algorithm (ECGA) (Harik *at al.*, 2006), and Factorized Distribution Algorithm (FDA) (Mühlenbein and Mahnig, 1999) are multivariate EDA.

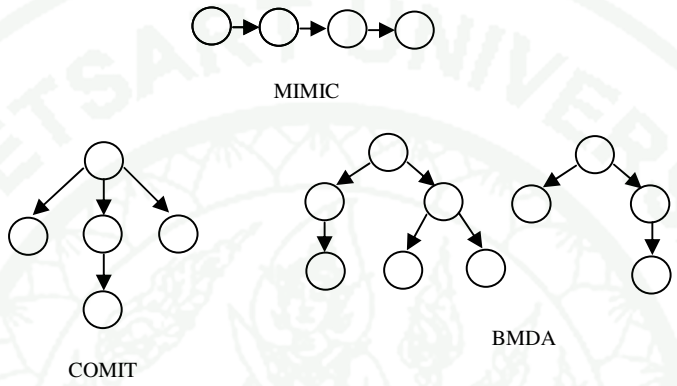


Figure 4 Relationship graph of EDA

Compact Genetic Algorithm

Harik *et al.* (1999) introduced a Compact Genetic Algorithm (cGA). The advantage of cGA is low memory consumption. cGA consumes less memory than traditional GA because the algorithm uses a single probability vector to represent the whole GA population. The probability vector requires only $l \times \lceil \log_2 n \rceil$ 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 5 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 generates two individuals from the probability vector. Next, both individuals are evaluated. 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 of the corresponding index by 1/n, if the winner bit is 1 and the loser bit is 0 in that index.
- Decrease the probability value of the corresponding index by 1/n, if the winner bit is 0 and the loser bit is 1 in that index.

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 to step 5. Notice that there is no crossover and mutation in cGA.

```
Parameters
        n : population size
        l : chromosome length
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 := 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
                go to step 2;
6)  p represents the final solution
```

Figure 5 Compact Genetic Algorithm pseudo code

cGA is applied to solve large-scale problems. Watchanupaporn *et al.*(2006) 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. The algorithm is easy to parallelize. Sataporn and Suwannik (2010) implemented the algorithm to run on a many-core GPU using CUDA. Sastry *et al.* (2007) ran cGA on a cluster of computers to solve a billion-bit noisy OneMax problem. The problem is more difficult than OneMax problem because noise disrupts the evolutionary search.

Mutual-Information-Maximizing Input Clustering Algorithm (MIMIC)

Each iteration in EDA consists of selecting highly fit chromosomes, modeling them, and using the model to generate the next generation of chromosomes. Modeling highly fit individuals is the core of every EDA. Different algorithms model has different ways to model the real joint probability distribution $p(X)$ of highly fit chromosomes, which is:

$$p(X) = p(X_1 \mid X_2 \ldots X_n) p(X_2 \mid X_3 \ldots X_n) \ldots p(X_{n-1} \mid X_n) p(X_n) \quad (1)$$

The performance of EDAs depends on the quality of the estimation of the real probability distribution. MIMIC estimates the real joint distribution using a pairwise conditional distribution, which is in the following form:

$$p'(X) = p(X_{i_1} \mid X_{i_2}) p(X_{i_2} \mid X_{i_3}) \ldots p(X_{i_{n-1}} \mid X_{i_n}) p(X_{i_n}) \quad (2)$$

where $(i_1, i_2, i_3, ..., i_n)$ denote the permutation order of chromosome positions. MIMIC uses a greedy method to determine the permutation. The time complexity per iteration of MIMIC is $O(n^2)$.

MIMIC uses a greedy method to find the value the permutation starting from i_n down to i_1 that makes $p'(X)$ closest match the real distribution $p(X)$. i_n is the position of a chromosome with the lowest entropy. i_{n-1} is the position of a chromosome with the lowest conditional entropy $h(X_{i_{n-1}} \mid X_{i_n})$ among all positions excluding X_{i_n}. i_{n-2} is

the position of a chromosome with the lowest conditional entropy $h(X_{i_{n-2}} \mid X_{i_{n-1}})$ among all positions excluding X_{i_n} and $X_{i_{n-1}}$. And so on. By finding conditional entropy between two variables, MIMIC can be classified as a bivariate EDA.

The next generation is generated from the model. First, the i_n^{th} bit in the chromosome is generated based on the probability $p(X_{i_n})$. Then, the i_{n-1}^{th} bit in the chromosome is generated based on the value of i_n^{th} bit and the conditional probability $p(X_{i_{n-1}} \mid X_{i_n})$. The remaining bits are generated in the same manner. Both modeling and generating can be done in $O(n^2)$. MIMIC is applied to solve large-scale problems. Watchanupaporn and Suwannik (2011) used compressed encoding with MIMIC to solve several large problems (i.e., 100, 400, and 800 bits) problems.

Bayesian Optimization Algorithm (BOA)

A Bayesian network is a directed acyclic graph (DAG) that represents a set of random variables and their dependencies. In BOA, a Bayesian network can be used to model interdependencies among chromosome positions. A Bayesian network is constructed as a model of selected individuals. Any search method and any metric can be used to construct the network. In BOA, the construction uses a search algorithm which greedily adding edges to the empty graph. Networks are scored using Bayesian-Dirichlet Equivalence metric.

BOA can solve difficult problems such 3-deceptive, trap-5, and 6-bipolar. However, it is very time consuming. Its time complexity for each iteration is $O(n^2N+n^3)$, where n is the length of a chromosome and N is the size of selected individuals. A pseudo code for BOA is shown in Figure 6.

Step 1 : set $t \leftarrow 0$
randomly generate initial population $P(0)$
Step 2 : select a set of promising strings $S(t)$ from $P(t)$
Step 3 : construct the network B using a chosen metric and constraints
Step 4 : generate a set of new strings $O(t)$ according to the joint distribution encoded by B
Step 5 : create a new population $P(t+1)$ by replacing some strings from $P(t)$ with $O(t)$
set $t \leftarrow t + 1$
Step 6 : if the termination criteria are not met, go to Step 2

Figure 6 BOA pseudo code

Differential Evolution

Differential Evolution (DE) is an evolutionary optimization method. The first generation of real vectors is created randomly. Each vector has *D* values (*D* stands for dimension). A population consists of *NP* vectors. There are two schemes (i.e., DE1 and DE2) presented in Storn and Price (1995). In this study, DE1 is used.

A new generation is created by the following method. Each vector competes with its trial vector. The one with less cost is selected to the next generation. A trial vector is created by combining the vector with a mutant vector. The combination is similar to crossover in Genetic Algorithm. A mutant vector $X_c^{'}$ is created by adding a random vector X_c with a weight difference of other two random vectors $F(X_a - X_b)$ (hence the name Differential Evolution). The mathematical formula for creating a mutant vector is as follows:

$$X_c^{'} = X_c + F(X_a - X_b) \tag{3}$$

DE's parameters and their suggested settings (by its inventor) (Price and Storn, 1995) are listed below.

NP (or population size) should be 5-10 times the number of parameters *D*.

F (or the weight) should start with 0.5. *F* and *NP* should be increased if the algorithm converges prematurely.

CR (or the crossover rate) should be 0.1. However, one might try CR = 0.9, 0.1, or 0.0.

Compressed Encoding

1. Lempel-Ziv-Welch Algorithm

Lempel-Ziv-Welch Algorithm (LZW) is a lossless dictionary-based data compression/decompression algorithm (Welch, 1984). 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 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. Initially, 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/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.

```
Algorithm LZW_Decompress
  input : array of integers
  output : binary strings

      add entries 0,1 to the dictionary
      read one code from input to c
      output str(c)
      p = c
      while input are still left
            read one code from input to c
            if the code c is not in the dictionary
                  add str(p) + fc(str(p)) to the dictionary
                  output str(p) + fc(str(p))
            else
                  add str(p)+fc(str(c)) to the dictionary
                  output str(c)
            end if
            p = c
      end while
The variable c is used to store a code read from input
The variable p is the previous value of c
The function str(code) returns a string associated with code
The function fc(string) returns the first character in string
```

Figure 7 LZW Decompression pseudo code

A nice property of LZW is that the dictionary does not have to be packed with a compressed data. LZW decompression does not require a dictionary because the algorithm can reconstruct the dictionary while processing the compressed data. A pseudo code for LZW Decompression algorithm can be seen in Figure 7. The reason that we show only the decompression algorithm is because LZWGA uses only LZW decompression algorithm.

Figure 8 shows LZW decompression example. The leftmost column is the input, which is an array of integers. The second column is the output, which is a binary string. During the decompression, new entries are added to the dictionary.

Decode		**Dictionary**	
Input	**Output**	**Index (c)** *Initial table*	**Full string**
		0	0
		1	1
0	0	-	-
2	00	2	00
1	1	3	001
3	001	4	10
1	1	5	0011
1	1	6	11

Figure 8 Example for LZW decompression

The LZWGA chromosome is an array of integers which will be decompressed to a binary chromosome before the fitness evaluation. LZWGA requires less memory than GA. LZWGA spends less time per generation than GA even though there is an additional decompression step before fitness evaluation. This is because LZWGA chromosome is smaller than GA chromosome. Therefore, genetic operations (e.g., crossover, mutation, and reproduction) in LZWGA require less time than GA. For example, to solve one-million-bit problem, each chromosome in LZWGA have only 16,000 genes or 207,631 bits but GA have to used 10^6 bits for each chromosome (Kunasol *et al.*, 2005, 2006).

LZW compressed encoding can be used in evolutionary computation. LZWGA combines LZW with GA. The algorithm can solve very large problems that original GA cannot solve. Moreover even with an additional decompression step, if a shorter chromosome is used, one iteration of LZWGA is much faster. Additionally, the algorithm is more compact (i.e., requires less memory) than GA. LZWEDA combines LZW with EDA. It is a class of algorithms including LZWCGA, LZWMIMIC, LZWBOA (see the results in Appendix).

2. Arithmetic Coding

Arithmetic coding compression algorithm compresses a binary string into two real numbers p and c ranged between [0, 1). The first number is the probability that zero will occurs in the binary string. The second number is the compressed message. The first number is denoted by p and the second number is denoted by c.

The coding is best explained by an illustration. The following example demonstrates a decompression of (p, c) = (0.4, 0.6) to a 4-bit binary string. As shown in Figure 9, p divides the interval [0, 1) into 2 sub-intervals: [0, 0.4) and [0.4, 1). Since the compressed message c is in the second sub-interval, the algorithm outputs 1. Next, the algorithm partitions the second interval [0.4, 1) into two sub-intervals proportional to p. The resulting sub-intervals are [0.4, 0.64) and [0.64, 1). Since the compressed message c is in the first sub-interval, the algorithm outputs 0.
Then, the algorithm partitions the first interval [0.4, 0.64) into two sub-intervals proportional to p. The resulting sub-intervals are [0.4, 0.496) and [0.496, 0.64). Since the compressed message c is in the second sub-interval, the algorithm outputs 1.
Finally, the algorithm partitions the second interval [0.496, 0.64) into two sub-intervals proportional to p. The resulting sub-intervals are [0.496, 0.5536) and [0.5536, 0.64). Since the message c is in the second sub-interval, the algorithm outputs 1.

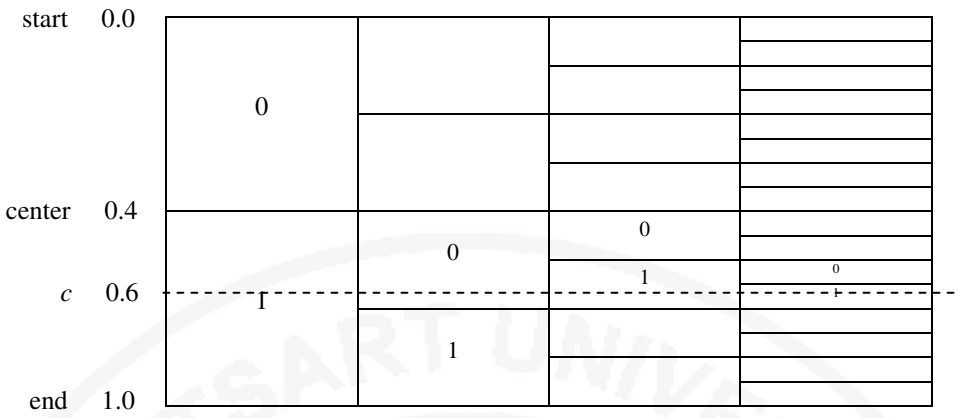


Figure 9 Using Arithmetic Coding to decompress (p,c) = (0.4, 0.6) to a 4-bit binary string. The output is 1011

Lempel-Ziv-Welch Estimation of Distribution Algorithm (LZWEDA)

LZW chromosome encoding was applied to various EDAs, which are cGA, MIMIC, and BOA. The resulting algorithms are LZWCGA, LZWMIMIC, and LZWBOA, respectively. Adding LZW encoding to an existing EDA is easy. The EDA algorithm does not have to be modified. Rather, the fitness evaluation must be modified by adding decoding and decompressing at the beginning. A binary string is decoded into an array of integers using Gray decoding (see Figure 10). Then, the integer array is decompressed into a binary string using the LZW decompression algorithm. Finally, the binary string from the previous step is evaluated and its fitness is returned to EDA.

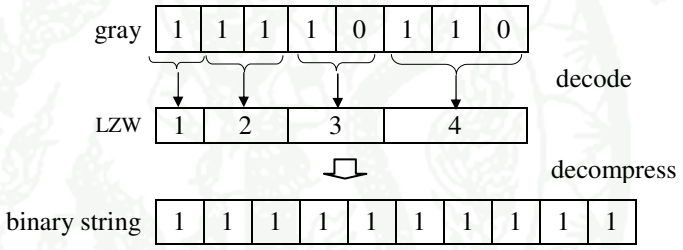


Figure 10 Gray code to binary string conversion in LZWEDA

All experimental results are the average performance obtained from 30 runs. For an experiment that involves benchmark with randomness, we use the same set of seeds. The sizes of the compressed chromosome are set to one-half of, equal to, and double that of the decompressed chromosome for these problems. We call this ratio the chromosome compression ratio. We compare the performance of cGA and LZWCGA, MIMIC and LZWMIMIC, BOA and LZWBOA for various compression ratios. These algorithms were tested against 100, 400, and 800-bit problems. The results are in Appendix.

Table 1 shows how to calculate the number of bits per chromosome. For example, in order to store an integer 0 or 1 (2 possible values), 1 bit is required. To store an integer 0, 1, or 2 (3 possible values), 2 bits are required. In general, to store n possible value, $\lceil \log_2 n \rceil$ bits are required. The i^{th} index in an LZW chromosome has i + 2 possible values.

Table 1 The number of bits per LZW chromosome is calculated by summing bits needs the store every possible value in each gene

Number of Gene	Possible Values for each Gene	Total Bit
1	(0-1)	1
2	(0-1), (0-2)	1 + 2
3	(0-1), (0-2), (0-3)	1 + 2 + 2
4	(0-1), (0-2), (0-3), (0-4)	1 + 2 + 2 + 3
5	(0-1), (0-2), (0-3), (0-4), (0-5)	1 + 2 + 2 + 3 + 3

1. Lempel-Ziv-Welch Compact Genetic Algorithm (LZWCGA)

There are two types of LZWCGA. The first one uses probability vector as in the original cGA with Gray decoding. The second one uses probability matrix to model a population of integer array.

Watchanupaporn *et al.* (2012) proposed LZWCGA by using Gray decoding. To use LZW compressed encoding with cGA, we add a decoding and decompressing after step 2 of cGA pseudo listed in Figure 5. 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 11 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. In cGA, an invidual is created as a binary string. In LZWCGA, an individual is a binary string in a compressed form.

```
...
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 11 LZWCGA pseudo code

Watchanupaporn and Suwannik (2010) proposed LZWCGA with a probability matrix. The original cGA uses a probability vector to represent the whole GA population. However, LZWCGA uses a probability matrix instead of a single probability vector because LZWGA's chromosome is an array of integer. Each column of the probability matrix is a probability that a particular integer value will occurs for each gene. We found the LZWCGA's performance is comparable to LZWGA on OneMax and Trap problem.

2. Lempel-Ziv-Welch Mutual-Information-Maximizing Input Clustering Algorithm (LZWMIMIC)

Watchanupaporn and Suwannik (2011) proposed LZWMIMIC. The proposed algorithm combines the LZW compressed chromosome encoding and Mutual-Information-Maximizing Input Clustering (MIMIC) algorithm. The performance of the original MIMIC and LZWMIMIC are compared on standard benchmark problems. Further, compressed chromosome length and problem size are varied to see their effect in the performance. The experimental results show that LZWMIMIC outperforms the original MIMIC. LZWMIMIC performs well when the problem is large. We investigated two parameters related to the compressed encoding

in LZWMIMIC and found that the length of a decompressed binary chromosome affects the performance of LZWMIMIC.

3. Lempel-Ziv-Welch Bayesian Optimization Algorithm (LZWBOA)

Watchanupaporn and Suwannik proposed the algorithm LZWBOA which combines the compress encoding and probabilistic model building. BOA is a multivariate EDA. It can model a problem with multiple dependencies between chromosome positions. We used a C++ implementation of BOA written by Martin Pelikan (1999). We conducted experiments to compare the performance of BOA and LZWBOA.

LZW encoding improves the performance of every type of EDA. For cGA, the LZW version outperforms the original version on all test problems except OneMax, for which the performance of both versions is similar. For MIMIC, LZW encoding also outperforms the binary encoding in almost every cases. Moreover, LZW encoding scales much better than the original encoding. Finally, among the tested EDAs, LZWBOA gains the most performance when LZW encoding is used.

Adding LZW decompression to each fitness evaluation would seem to increase the overall evolution time. However, this is not true if LZWEDA can determine a solution in fewer generations. From our experimental results, for the same problem size (i.e., the original encoding and LZW encoding with a 1x compression ratio), LZWEDA can determine the optimal solution faster than the original EDA (with only 2 exceptions: cGA's 800-bit OneMax and MIMIC's 100-bit Four-Peak). LZWBOA is superior to LZWCGA and LZWMIMIC in the sense that it can reliably find optima most often for every compression ratio. However, for the 1x compression ratio, LZWCGA can determine a solution faster than the other algorithms.

EDA with LZW scales better than the original EDA. As the problem size becomes larger, the performance gap between LZW encoding and the original encoding are also wider. The performance gaps of cGA and LZWCGA in 100, 400,

800-bit problems are 3.64%, 10.15%, and 18.00% respectively. The gaps of MIMIC are −0.38%, 0.47%, and 11.23%. The gaps of BOA are 0.67%, 10.94%, and 19.37%.

Adding LZW decompression to each fitness evaluation seem to increases the time per generation. However, this is not true when the length of the LZW-encoded chromosome is less than the original problem size. For example, in LZW encoding, we can use a 50-bit chromosome to solve a 100-bit problem. The decompression time is $O(n)$, whereas the time to probabilistically model the population is typically larger than that required by the sophisticated EDA (e.g., $O(n^2)$ for MIMIC and $O(n^2N+n^3)$ for BOA). The experiment shows that, with the compression ratio 0.5x, LZWMIMIC and LZWBOA are faster than the original versions.

Among various EDA, BOA exhibits the best performance in term of the number of fitness evaluations. However, cGA can provide competitive result faster. Additionally, the compactness of LZW encoding makes it attractive for many EDA that have large time complexity. Finally, LZW encoding can alleviate the memory shortage problem. For instance, if the program runs out of memory when solving a 1600-bit problem, then we might try using a 1000-bit LZW chromosome to solve the same problem.

4. Analysis of Fitness Landscape for LZWEDA

The difficulty of a problem depends on two factors: the size of search space and the shape of fitness landscape. A problem with a larger search space is usually more difficult to solve. LZW encoding mitigates the problem of a large search space. Moreover, a problem with a more complex fitness landscape is more difficult. For example, OneMax is an easy problem because the fitness value can be used to guide the search in the correct direction.

In OneMax problem, for an 8-bit compressed chromosome, the landscape is more complex than the landscape for a 10-bit uncompressed chromosome. However, as the number of bits of the uncompressed chromosome increases, the landscape can guide the search better. One reason is because there are more solutions in the search space.

The Trap problem is very difficult to solve because the fitness landscape deceives the search into moving away from the global optima. However, when the search space is transformed using LZW encoding, the landscape can obviously better guide the search to the global optima.

The Four-Peak problem has local optima. Moreover, many individuals have low fitness values. However, when the search space is transformed by LZW encoding, many individuals have higher fitness values (the darker areas move up). Moreover, as the size of the compressed chromosome increases, separate part of the landscape becomes wider. Therefore, it is easier to move to a separate part of the landscape with higher fitness values.

We analyzed how LZW encoding changes the fitness landscape. LZW completely transforms the fitness landscape. Moreover, different compression ratios have different fitness landscapes. Furthermore, the number of the optimal solutions in the transformed landscape is not equal to those in the original landscape.

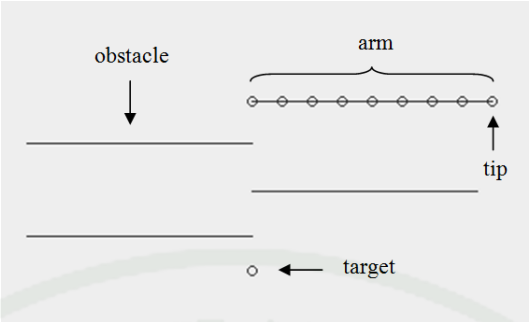
In this analysis, there are some cases where the compressed chromosome is larger than the decompressed one. However, in the actual use, which the problem size is larger, the size of the compressed chromosome is usually less than that of the uncompressed chromosome. The reason why our analysis uses only small problems is because we have measured the fitness of every chromosome and each chromosome's distance to the solution. It would take a significant amount of time to analyze longer chromosomes.

5. Generality of LZW Encoding

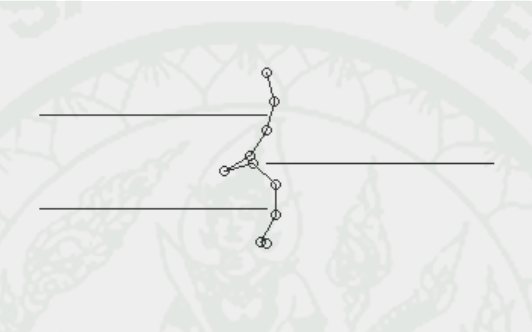
Compressed encoding tends to perform well in a class of problems with high regularity solution. Moreover as reported in (Chamlamai and Suwannik, 2007), compress encoding cannot compete with original encoding when a solution is constructed by randomness. However, due to previously mentioned benefits, there should be attempt for the further study of the compress encoding. The problem presented in this subsection is different from the benchmark problem because its structure is unknown and its solution is not regular.

To demonstrate the effectiveness of LZWEDA, we investigate the LZWCGA on Robot arm control programs (Suwannik *et al.*, 2005). This problem simulates a robot arm in a working environment. Figure 12(a) shows the initial robot arm configuration. The three lines are the obstacles. The target is located near the bottom of the figure. The objective is to moves the tip of robot arm to the target (see Figure 12(b)). The rotation step for each joint is 15 degrees clockwise and counter clockwise. The arm can sense if it hits an obstacle and knows the distance between the tip and the target. The arm cannot move any of its parts out of the boundary. A chromosome contains robot moving instructions (see Figure 13). After a robot executes all instructions in the chromosome or after it reaches the target, its fitness value will be calculated as follows.

$$f = \begin{cases} 1000 & ;if\ l < 5 \\ 1000 - l & ;otherwise \end{cases} \tag{4}$$



(a) Initial robot arm configuration



(b) The robot arm reaches the target

Figure 12 A simulated robot arm and its working environment

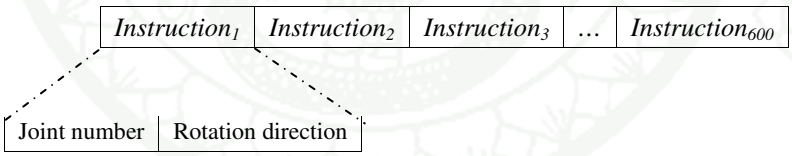


Figure 13 Robot arm control program is an array of instructions. Each instruction consists of joint number and rotation direction

We used cGA and LZWCGA to solve the robot arm problem. We varied the length of LZW chromosome by one half of, equal to, and double of the problem size. On average, the best performance is obtained when LZWCGA uses the same chromosome length as cGA. LZWCGA can find solution using about 4 times less number of fitness calculations than cGA. In summary, LZWCGA is more robust and can solve the real world problem.

Local Search Techniques

1. Hill Climbing (HC)

HC is a local search technique. It finds a better neighbor. If no such neighbor exists, HC returns the current chromosome. The neighbor of a binary individual is an individual that has one bit difference from the current individual. The search is illustrated in Figure 14 and Figure 15. The point c is a current point and n_1, n_2 are its neighbors. In Figure 14, the search will move to n_2, which will eventually move to the global optimum. However, in Figure 15, the search will select n_1 as a new current point and will be stuck in the local optimum. The major problem of HC is the search is likely to be stuck in local optima.

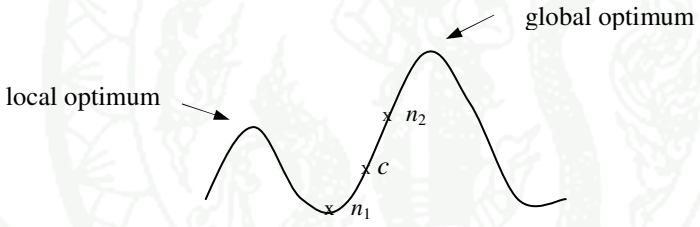


Figure 14 Hill climbing can find the global optimum

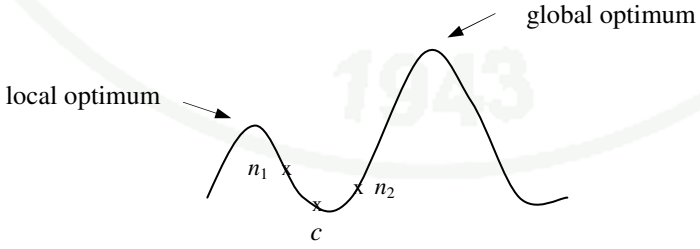


Figure 15 Hill climbing will be stuck in a local optimum

2. Simulated Annealing (SA)

SA is a probabilistic optimization method proposed in Kirkpatrick *et al.* (1983). SA is one of local search techniques. Contrary to HC, SA may accept neighbor with less fitness in order escape from local optima (see Figure 15). The plot of probability of accepting inferior solution (Equation (4)) is shown in Figure 16. This technique mimics metal cooling. At the beginning of the search, the variable T or temperature has high value. As the search progress, the temperature is decreased. According to the Equation (4), during the beginning of the search, SA is more likely to accept an inferior solution than during the end of the search.

$$P = \frac{1}{e^{\frac{\Delta f}{T}}} \tag{5}$$

where P is a probability.

Δf is fitness difference.

T is temperature.

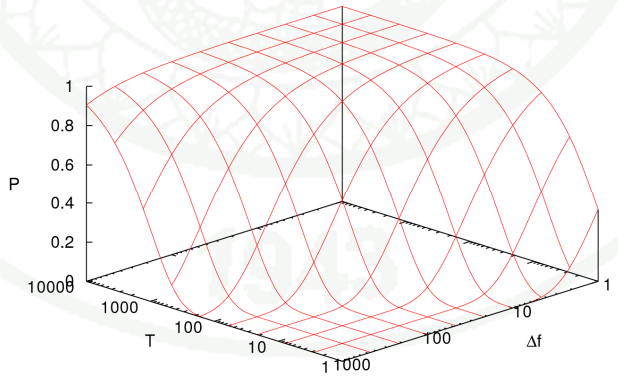


Figure 16 Probability of accepting inferior solution in Simulated Annealing

MATERIALS AND METHODS

Materials

1. Hardware : Intel(R) Core(TM) i5 CPU 650 @ 3.20 GHz, 4.00 GB of RAM (3.87 GB usable)

2. Operating System : Microsoft Windows 7 Ultimate, 64-bit

3. Software :

2.1 Java Development Kit: JDK 6.0

2.2 NetBeans IDE 7.2.1

Methods

1. Algorithms

1.1 LZW Differential Evolution

DE evolves vectors of real numbers. In this work, DE applied to solve a problem with binary solution. The idea is to convert a real vector into a binary string. After that, the binary string is evaluated according to a fitness function. The fitness will be returned to DE, which will proceed as usual. There are many ways to convert a real vector to a binary string. For example, simple real-to-binary DE (SDE) converts using the rule $X_i < 0.5$? 0 : 1, which is quite wasteful because one 64-bit double variable which represent X_i will be convert to just 1 bit. This work proposes the use of compression algorithm as more efficient real vector to binary conversion.

To use LZW compressed encoding with DE, we add a conversion and decompressing step before a fitness evaluation. As shown in Figure 17, the real value DE vector is converted to an LZW chromosome, which is an array of integers. To convert a real number to an integer, a fraction part is truncated. After that, the integer array is decompressed to a binary string. LZW decompression algorithm cannot decompress arbitrary input. Each code in an integer array must satisfy the constraint given in Equation (6) (Kunasol *et al*., 2005). Any positive integer can be changed to satisfy the constraint by modulo with *i*+2.

$$0 \le a_i \le i+1 \tag{6}$$

where *i* is a zero-based array index.

Implementing an LZW chromosome encoding in an object-oriented language is easy. The core algorithm does not have to be modified to support LZW encoding. Rather, as shown in Figure 18, our implementation of the core DE algorithm (see Figure 19) evaluates a real vector using a method double evaluate (double[ ] *vector*) in an interface Evaluator. An abstract class LzwEvaluator implements the interface by converting the variable *vector* to a variable *binaryString*, calling an abstract method double evaluate (byte[ ] *binaryString*), and returning the value obtained from the method. For each benchmark problem, we extend the class LzwEvaluator and implement the abstract method which evaluates the variable *binaryString*.

Note that even though the original DE can be instantly used with LZW encoding, we modified the vector initialization and offspring generation to satisfy the constraint given in Equation (6). LZWDE is created by modifying two steps in DE as shown in Figure 20. We also would like to point out that there is no compression step involved in LZWDE.

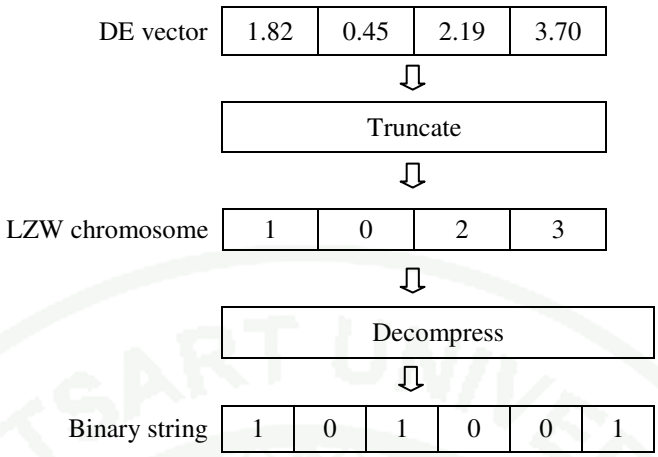


Figure 17 Real vector to binary string conversion in LZWDE

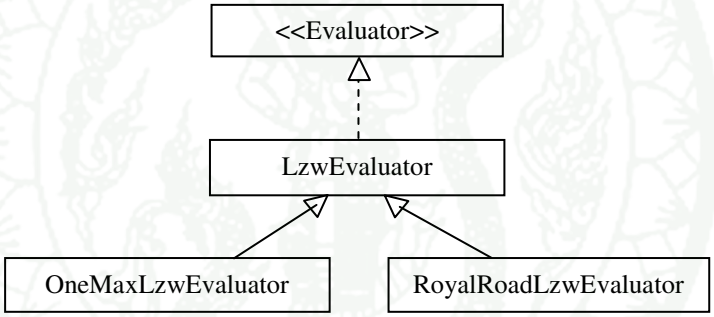


Figure 18 Implementing an LZW encoding in DE

```
Algorithm DE
  output : real vector

    initPopulation()
    evaluateAllVectors()
    while notTerminate()
        for each vector X in population
            (A, B, C) ← random 3 different vectors
            for each real value X_i in the vector X
                Trial_i = isCrossover() ? A_i + F(B_i − C_i) : X_i
            if cost(Trial) < cost(X)
                add Trial to nextPopulation
            else
                add X to nextPopulation
        population = nextPopulation
    return bestVector
```

Figure 19 Differential Evolution algorithm

```
initPopulation()

    for each vector X in population
        for each real value X_i in the vector X
            X_i ← Math.random() × (j + 2)


cost(X : real vector)

    L ← convertRealToInteger(X)
    B ← decompress(L)
    return evaluate(B)

The variable L is LZW chromosome.
The variable B is binary string.
```

Figure 20 LZWDE is created by modifying 2 steps in DE: initPopulation() and cost(X)

1.2 Arithmetic Coding Differential Evolution

ACDE (Arithmetic Coding Differential Evolution) combines Arithmetic Coding compression algorithm (AC) with Differential Evolution (DE). DE evolves vectors of real numbers. During the evolution, AC decompresses each vector of real numbers to a binary string. After that, the binary string is evaluated and its fitness is returned to DE. Therefore, ACDE is an algorithm that evolves a population of binary strings. A pseudo code for Arithmetic Coding decompression used in ACDE is shown in Figure 21. The algorithm runs in $O(l)$ time, where l is the number of bits to be produced.

```
Algorithm  Arithmetic Coding Decompression
  input:      p, c : double
  output:     data : binary string
    start ← 0
    center ← p
    end ← 1.0
    for (i ← 0; i < data.length; i++)
      if (c < center)
        data[i] ← 0
        end ← center
        center ← start + (center − start) × p
      else
        data[i] ← 1
        start ← center
        center ← start + (end − center) × p
    end for

start is the starting point of the first interval
center is the starting point of the second interval and the
       ending point of the first interval
end is the ending point of the second interval
```

Figure 21 Arithmetic Coding Decompression pseudo code

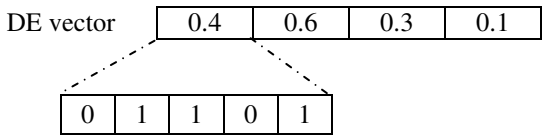


Figure 22 Decompressing a real vector to a binary string in ACDE (compression ratio = 5)

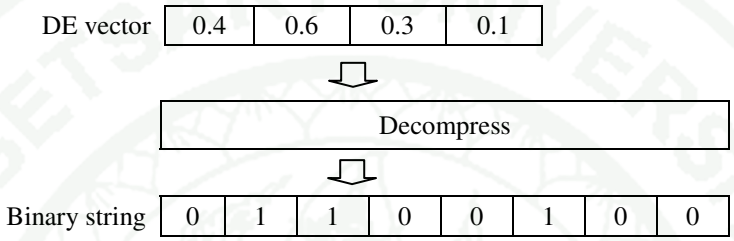


Figure 23 One real value in an ACDE vector can be decompressed to several bits (compression ratio = 2)

By applying AC to DE, we made some modifications to DE algorithm. The inputs of AC decompression algorithm are two numbers p and c (probability and code) and the output is a binary string (see Figure 22 and Figure 23). For example, the input (0.4, 0.6) will produce the output 1011 (see Figure 9). In this study, instead of evolving both p and c, we fixed the value of the probability p to 0.5. Each variable in the optimized vector is the code c. The range of the code is [0, 1). Therefore, every variable in the vector is initialized to be in the range. Moreover, the result of trial vector calculation has to be constraint to the range, there is no such constraint in the original DE.

Before the fitness evaluation, a vector of real variables X is decompressed to a binary string B. Each code $X[i]$ is decompressed to the binary string in the $(i \times r)^{th}$ to $((i+1) \times r - 1)^{th}$ positions, where r is a compression ratio or the number of bits that each code produces. After the decompression, the binary string is evaluated. The pseudo code for decompressing a DE vector X to a binary string B is as Figure 24.

```
Algorithm Decompress (X)
    B = allocate a binary string with the length r×X.length
    for i = 0 to X.length−1
        start = i×r
        end = (i+1)×r−1
        B[start..end] = AC_decompress(X[i])
    end for
    return B

X  is a real value vector.
B  is a binary string.
r  is a number of bits that each code will be decompressed.
```

Figure 24 ACDE with local search decompression pseudo code

1.3 Arithmetic Coding Differential Evolution with Local Search

To further improve the performance, we add a local search algorithm to ACDE. As shown in Figure 25, for each generation, after all trial vectors are created, the best real vector is decompressed to a binary string. Then, the local search is applied on the binary string to find a better neighbor. In this study, we use two local search techniques: hill climbing (HC) and simulated annealing (SA). After that, the output from the local search is compressed into a real value vector. Finally, the best vector in that generation is replaced by the output of compression algorithm.

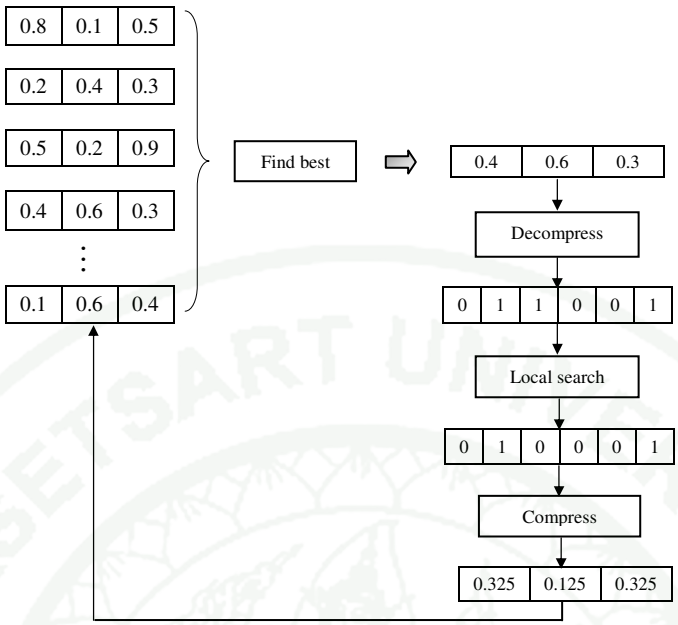


Figure 25 Local search in ACDE

1.4 LZWGA

The main difference between LZWGA and GA is that an LZWGA chromosome is in a compressed format. An LZWGA chromosome is an array of integers. It has to be decompressed to a binary string before its fitness can be evaluated. After fitness evaluation, the new population is created to replace the old population. The algorithm repeats the process of decompression, fitness evaluation, and creating a new population until a solution is found or a maximum generation is reached. The pseudo code of LZWGA is shown in Figure 26.

The algorithm begins by randomly creating the first generation of compressed chromosomes. A chromosome in the first generation are created as a random integer array with the constraint that the integer in the i^{th} index of a chromosome must not have value greater than i+1. Otherwise, the chromosome cannot be decompressed. In order to generate a valid value of the i^{th} integer, a random non-negative integer is modulo with i+2.

For example, an LZWGA chromosome that can be successfully decompressed is {1,2,3}. The decompression algorithm will output a binary string 111111. After decompression, a dictionary has the entries (0,0), (1,1), (2,11), and (3,111). Another valid chromosome is {0,1,2}. The decompression algorithm will output a binary string 0101.

If the i^{th} integer in an LZWGA chromosome is invalid, the dictionary look up will fail after the $(i+1)^{th}$ integer is read. An example of an invalid chromosome is {1,3,1}. Before entering the loop in Figure 26, the input "1" (the 0^{th} integer in the chromosome) is read and the algorithm output 1. In the first iteration, the algorithm reads "3" (the 1^{st} integer), adds to dictionary the string 11 at the entry 2, and outputs 11. In the second iteration, the algorithm reads "1" (the 2^{nd} integer), and fail when trying to execute str("3").

Before evaluating the fitness of the chromosome, the fixed length compressed chromosome is decompressed using LZW Decompression algorithm. Because the chromosome in LZWGA is compressed, it has to be decompressed before its fitness evaluation. A compressed chromosome is decompressed using LZW decompression algorithm. The result is a binary chromosome. 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. After that, the decompressed binary string is evaluated. The result becomes the fitness value for both compressed and decompressed chromosomes. If the length is less than the problem size, LZWGA will evaluate the fitness of available bits. For example, in 12-bit OneMax problem, if the length of a decompressed chromosome is 10, then we count the number of 1's in those 10 bits. As another example, in 12-bit Trap problem with 4-bit block size, if the length of a decompressed chromosome is 10, then we calculate the fitness for only available first two blocks, which is 8 bits. Finally, in 12-bit Long Path problem, if the length of a decompressed chromosome is 10, then we will fill another remaining 2 bits with 0's. Otherwise, we cannot calculate its fitness.

LZWGA creates the population of the next generation by selecting, recombining, and mutating compressed chromosomes. A highly fit chromosome is likely to be selected using any selection method such as tournament or roulette-wheel selection. Compressed chromosomes can be recombined using single-point, two-point, or uniform crossover. Because each of these crossover methods does not change the position of each integer, it automatically creates valid chromosomes that each integer satisfies the constraint. Therefore, the offspring is valid and can be decompressed. Mutation changes an integer in uncompressed chromosome to a random value that satisfies the constraint.

```
Algorithm LZWGA

  output : binary string
     Z ← create_first_generation()
     repeat
          P ← decompress(Z)
          evaluate(P)
          Z ← create_next_generation(Z)
     until is_terminate()
     return best_chromosome

The variable Z is the population of compressed chromosomes
The variable P is the population of uncompressed binary chromosomes
```

Figure 26 LZWGA pseudo code

2. Benchmark Problems

This section describes the benchmark problems used in this work. The problems include OneMax, RandomMax, Royal Road, Trap, Random Trap, Deceptive order-3, Four-Peak, Six-Peak, Long Path, NK Landscape, and Ising Spin Glass. RandomMax and Random Trap is a random-version of OneMax and Trap respectively.

We use the synthetic problems to assess the strengths and weaknesses of compressed 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 a simpler structure.

2.1 OneMax Problem

The OneMax problem is a widely used problem for testing the performance of various genetic algorithms. The problem can easily be solved even by univariate EDA. The problem can be called bit counting. Formally, this problem can be described as finding a string $\vec{x} = x_1, x_2, \ldots, x_k$, where , $x_i \in \{0,1\}$, that maximizes the following equation:

$$F(\vec{x}) = \sum_{i=1}^{k} x_i \tag{7}$$

2.2 RandomMax Problem

A random version of OneMax is called RandomMax. This problem was designed to be unfavorable against an optimizer that has a bias toward high regularity solution such as LZWGA (Chamlamai and Suwannik, 2007).

The optimum of OneMax is in the string of all ones. RandomMax problem is based on OneMax problem but an optimal solution is a random binary string. The objective is to maximize the number of bits that match that random string. Thus, instead of counting the number of 1's in the binary string, this function counts the number of matches between a binary string and a prespecified random binary string. An example of this is shown in Figure 27. The fitness of an example individual is 3 because there are 3 bits that match the solution. Although this problem is difficult for LZWGA, it can still easily be solved even by univariate EDA.

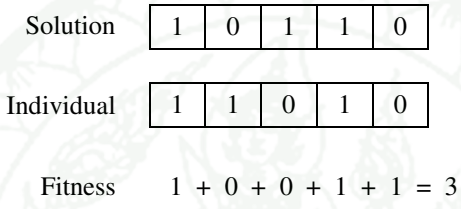


Figure 27 RandomMax fitness evaluation

2.3 Royal Road Problem

A Royal Road function has dependency among variables in the same block. The Royal Road function, denoted by *R,* is defined as:

$$R(x) = \sum_{s \in S} c_i \delta_i(x), \quad where\ \delta_i(x) = \begin{cases} 1 & ;if\ x \in s_i \\ 0 & ;otherwise \end{cases} \tag{8}$$

where c_i is a coefficient of each schema s_i . c_i is normally equal to k.

s_i is a schema (i.e., a chromosome template) that have 1 defined in the range $i \times k$ to $((i+1) \times k) - 1$, where k equals to a block size. All other positions contain a wild card '*'. Figure 28 shows a set of schemata, when $k = 5$.

s_0 = 11111**********

s_1 = *****11111*****

s_2 = **********11111

Figure 28 Schemata of Royal Road problem

2.4 Trap Problem

The Trap function can fool the gradient-based optimizers to favor zeros, but the optimal solution is composed of all ones. The Trap function is a fundamental unit for designing test functions that resist hill-climbing algorithms (see Figure 29). A *k*-bit trap function is defined as:

$$F(\vec{x}) = \begin{cases} f_{high} & ;if\ u = k \\ f_{low} - \dfrac{u \times f_{low}}{k-1} & ;otherwise \end{cases} \quad (9)$$

where $\vec{x} = \{0,1\}, u = \sum_{i=1}^{k} x_i$

$f_{high} > f_{low}$. Usually, f_{high} is set at *k* and f_{low} is set at *k*–1.

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

$$F_{m \times k}(K_1 \ldots K_m) = F_k(K_i), \quad K_i \in \{0,1\}^k \quad (10)$$

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

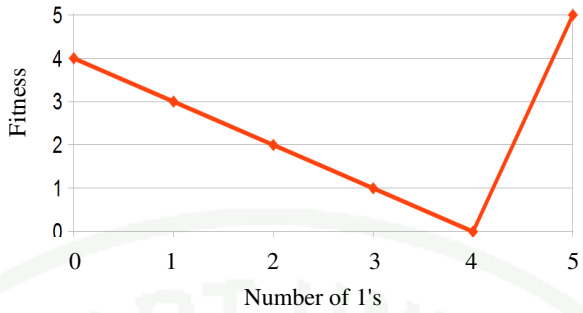


Figure 29 Trap function

2.5 Random Trap Problem

Random Trap problem (Watchanupaporn and Suwannik, 2010) is similar to Trap problem. The Random Trap problem of order *k* is defined using the same partitions as trap of order *k*, but each part is compared against a random binary string defined as a solution. The function counts the bits that are equal to the randomly preset solution. This problem is designed to reduce the effectiveness of LZW compressed encoding evolutionary algorithm in solving a problem with high regularity solutions. Figure 30 shows how to evaluate Random Trap 10-bit. The fitness value for this example is 8.

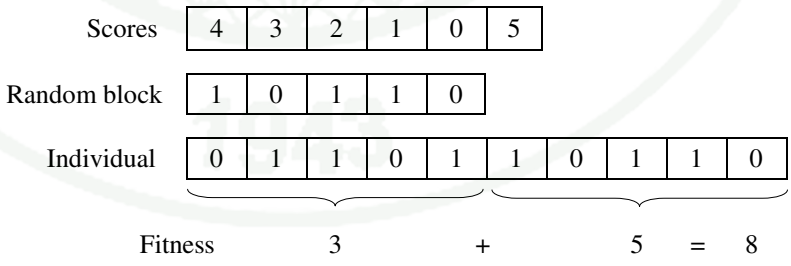


Figure 30 Random Trap fitness evaluation

2.6 Deceptive Order-3 Problem

In Deceptive problem (Goldberg, D.E., 1989), an individual composes of several blocks. Each of the blocks is evaluated by a deceptive function. The deceptive 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. The order-3 deceptive function is defined as:

$$\begin{aligned} f(000) &= 28 \\ f(001) &= 26 \\ f(010) &= 22 \\ f(100) &= 14 \\ f(011) &= 0 \\ f(101) &= 0 \\ f(110) &= 0 \\ f(111) &= 30 \end{aligned}$$

The deceptive problem can be decomposed to several deceptive functions. The problem, denoted by f_m, is defined as:

$$f_m(K1 \cdots K_m) = \sum_{i=1}^{m} f(K_i), K_i \in \{0,1\}^3 \quad (11)$$

2.7 Four-Peak Problem

The Four-Peak problem (De Bonet *et al*., 1997; Baluja and Caruana, 1995) has two global optima and two suboptimal local optima, which make a total of 4. The problem is defined as follows.

$$f(\vec{x},T) = \max(h(1,\vec{x}),t(0,\vec{x})) + r(\vec{x},T) \quad (12)$$

where

$$h(b,\vec{x}) = number \quad of \quad leading \quad b's \quad in \quad \vec{x} \quad (13)$$

$$t(b,\vec{x}) = number \quad of \quad trailing \quad b's \quad in \quad \vec{x} \tag{14}$$

$$r(\vec{x},T) = \begin{cases} N & ;if \quad t(0,\vec{x}) > T \quad and \quad h(1,\vec{x}) > T \\ 0 & ;otherwise \end{cases} \tag{15}$$

$\vec{x}$ is an input vector.
N is the length of a chromosome.
T is a threshold. In this work, T is 10% of N.
b is a value of bit.

For a 10-bit Four-Peak problem, the global optima are 1100000000 and 111111100. The local optimums are chromosomes with all 1's and all 0's. The fitness of the 10-bit optima is 18. The fitness of the 10-bit local optima is 10.

2.8 Six-Peak Problem

The Six-Peak problem (De Bonet *et al.*, 1997) 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 $r(\vec{x},T)$ is changed as follows.

$$r(\vec{x},T) = \begin{cases} N & ;if \quad (t(0,\vec{x}) > T \quad and \quad h(1,\vec{x}) > T) \quad or \\ & \quad (t(1,\vec{x}) > T \quad and \quad h(0,\vec{x}) \quad > T) \\ 0 & ;otherwise \end{cases} \tag{16}$$

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. The fitness of the optima is equal to that of the Four-Peak problem with the same problem size.

2.9 Long Path Problem

Long Path problem (Horn, J. *et al.*, 1994) is a problem that can be solved by a hill-climbing algorithm. However, it is not practical to solve this problem using a hill climbing algorithm. This is because climbing the hill (or the path) takes exponential time. Each point in the path is differed by one bit. The path is constructed such that is exponentially long. The height from the bottommost of the hill to the top is equal to:

$$HillHeight(l) = 3 \times 2^{\lfloor (l-1)/2 \rfloor} + l - 2 \qquad (17)$$

where l is a chromosome length.

Figure 31 shows chromosomes in a path and its fitness value. The chromosome that is on a path will have fitness equal to its position number plus the chromosome length. For example, the fitness of the chromosome 00111 is equal to 8. The chromosome that is not on a path will have fitness equals to the number of 0's in the chromosome.

	position
0	0
1	1

(a) 1-bit

			position
0	**0**	0	0
0	**0**	1	1
0	**1**	1	2
1	**1**	1	3
1	**1**	0	4

(b) 3-bit

					position
0	**0**	0	0	0	0
0	**0**	0	0	1	1
0	**0**	0	1	1	2
0	**0**	1	1	1	**3**
0	**0**	1	1	0	4
0	**1**	1	1	0	5
1	**1**	1	1	0	6
1	**1**	1	1	1	7
1	**1**	0	1	1	8
1	**1**	0	0	1	9
1	**1**	0	0	0	10

(c) 5-bit

Figure 31 Chromosomes in a path of a Long Path problem

2.10 NK Landscape Problem

The problem (Pelikan, M., 2010) is defined as follows:

$$F(\vec{x}) = \sum_{i=1}^{N} k(\vec{x}, i) \tag{18}$$

where $\vec{x}$ is a binary string, i is a position in a binary string, and N is the length of the binary string.

The value of $k(\vec{x}, i)$ is obtained from a table of size 2^k using the values from the $\vec{x}_i$ to $\vec{x}_{i+K-1}$ bits as indices (see Figure 32). Each entry in the table is randomly initialized. The fitness of this function depends on the value of the gene and its neighbors. The goal is to maximize the function. Unlike the previous problems, the optimal solution is unknown and depends on the seed of the random number

generator. Therefore, the performance of each algorithm is compared using the same set of seeds.

For example: $K = 2$, multiplier = 100

Binary: 11101101001, Fitness = 358

Table 0.730878 0.410081 0.207715 0.332717

[0] [1] [2] [3]

Fitness = (0.332717 + 0.2007715 + 0.410081 + … + 0.332717) × 100

For example, suppose that $K = 2$ and the table is randomly initialized with {0.730878, 0.410081, 0.207715, 0.332717}. A binary string 1101101001 would have a fitness value of 358. The values from the binary string 11, 10, 01, …, 01, 11 are used as indices for table look up. Then, the look up results 0.332717, 0.2007715, 0.410081, …, 0.332717 are summed and multiplied by 100. The reason that we multiply the result by 100 is because we will use only the integer part of the result. Such multiplying would move some significant digits in the decimal part to the integer part.

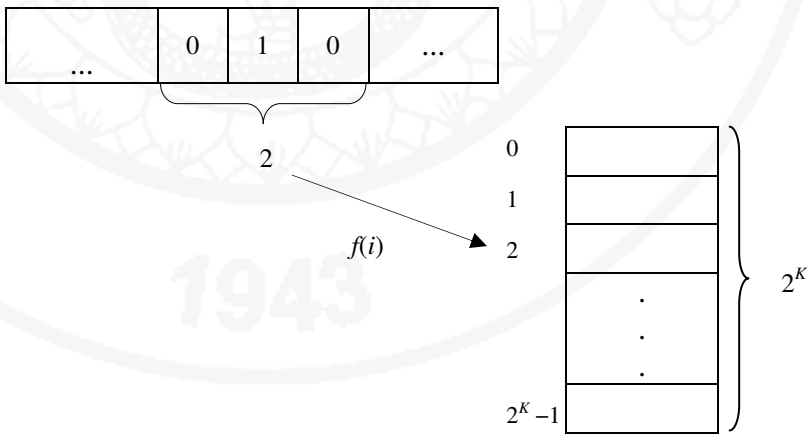


Figure 32 Addend is obtained by table look up in NK Landscape

2.11 Ising Spin Glass Problem

A 2D spin glass model is a grid of atoms (Kindermann and Snell, 1980). Each atom has an atomic spin S_i which is either +1 or −1. A coupling $J_{i,j}$ between a pair of atoms i and j is randomly initialized. The problem is to find a configuration of spins which has the lowest energy. The table of configuration is generated for each run. The fitness is calculated by adding multiplication results of all edges. An example is shown in Figure 33. The energy is obtained by the following formula:

$$energy = \sum_{<i,j>} s_i J_{i,j} s_j \tag{19}$$

Table 1

-1	**1**	-1
-1	**-1**	1
1	**-1**	-1

Table 2

-1	**-1**	**1**
1	**-1**	**1**
-1	-1	1

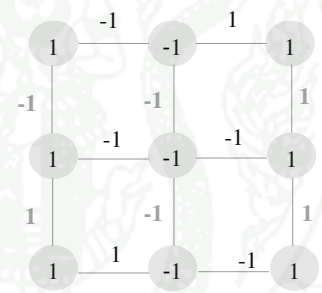


Figure 33 Example of Ising Spin Glass fitness evaluation. A chromosome 101101101 has a fitness value of 2

3. Experiments

3.1 LZWDE

In (Gong and Tuson, 2007), the authors applied DE to solve the following discrete optimization problems: OneMax, Royal Road, Order-3 Deceptive, and Long Path problems. We test the performance of our algorithm on the same benchmark. Every benchmark problem is a maximization problem. However, since DE is a global minimizer, the fitness is transformed by multiplying the cost function with −1.

We conducted the experiment to compare the performance of LZWDE with Gong and Tuson's binary adapted DE operators and with simple real to binary conversion DE (SDE). SDE, converts a real value to a binary using the rule ($X_i < 0.5$? 0 : 1)

Table 2 shows the experimental parameters. The length of an LZWDE chromosome is less than an SDE chromosome. For example, in OneMax problem, the length of LZWDE is 1/5 of SDE. Before a fitness evaluation, the compressed chromosome is decoded and decompressed with LZW decompression algorithm. The length of the decompressed binary chromosome is varied depending on the code in the integer array. If the length is more than the size of the problem size, the excess bits are discarded. However, if the length is less than the problem size, LZWDE will evaluate the fitness of available bits. All experimental results are the average performance obtained from 30 runs.

Table 2 Experimental parameters for LZWDE

Parameter	OneMax	Royal Road	Deceptive order-3	Long Path
Population size	50	30	100	30
Problem size	500	80	300	29
LZW chromosome length	100	20	25	10
Maximum generation	500	500	2000	300

3.2 ACDE

3.2.1 ACDE for Binary Encoding

We conducted an experiment to compare the performance of ACDE with SDE. Table 3 shows the experimental parameters. The compression ratio (i.e., the number of decompressed bits per code) is set to 5 because the problem size is divisible by this number. Any numbers that can divide the problem size can be used but the result may not be the same. The population size for SDE and ACDE is set to 10 times the number of variables in a vector. However, since the compression ratio of ACDE is set to 5, the population is ACDE is 5 times smaller than that of SDE. All experimental results are the average performance obtained from 30 runs.

Table 3 Experimental parameters for ACDE

Parameter	Value		
	Random Trap (Trap size: 5)	**NK Landscape**	**Ising Spin Glass**
Problem size (bits)	50	100	25, 100, 225, 400
Pop. size (SDE)	500	1000	250, 1000, 2250, 4000
Pop. size (ACDE, ACDE-local)	100	200	50, 200, 450, 800
ACDE's compression ratio	5	5	5
Maximum generations	200	200	200

3.2.2 ACDE with Local Search

We conducted an experiment to compare the performance of ACDE, ACDE with local search, and SDE. Table 4 shows the experimental parameters. We varied the problem size from 100 to 800 to see the scalability of each algorithm. The population sizes for each algorithm are varied. SDE is given largest population while ACDE and ACDE with local search are given the smallest population size. Generally, for evolutionary algorithm, larger population is likely to give higher quality solution but requires more computation time per generation. Note

that ACDE required only a decompression algorithm. However, ACDE with local search uses both decompression and compression. All experimental results are the averaged performance obtained from 30 runs.

For Ising Spin Glass, we varied the width from 10 to 30. For NK Landscape, we consider table of the size $K = 4$. When K=1, there is no dependency among variable. The larger K means more dependencies.

Table 4 Experimental parameters for ACDE with local search

Parameter	Value
Problem size (bits)	100, 400, 800/900 (for Ising Spin Glass)
Population size (SDE)	Problem size × 10
Population size (LZWDE)	(Problem size ÷ 2) × 10
Population size (ACDE, ACDE-local)	(Problem size ÷ 5) × 10
Maximum generation	200

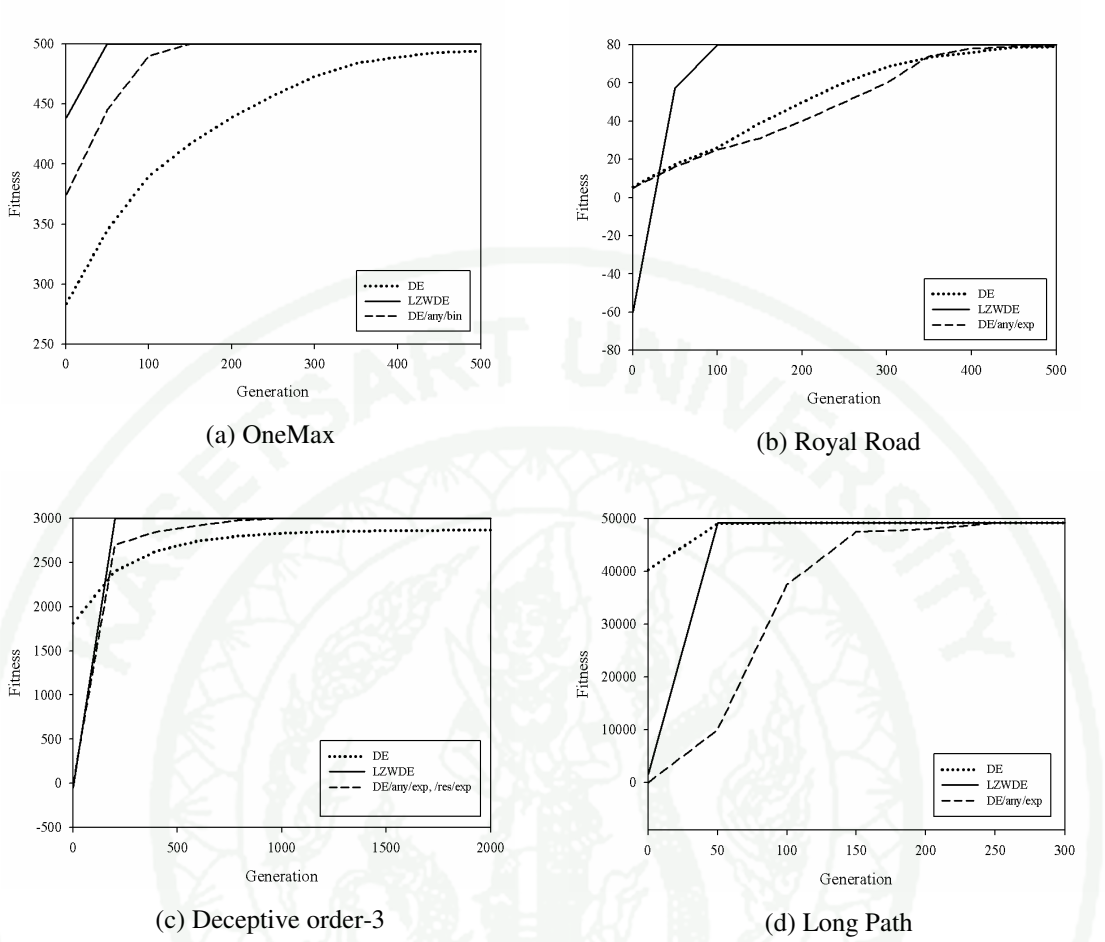
RESULTS AND DISCUSSION

Results

1. LZW Differential Evolution (LZWDE)

Gong and Tuson (2007) used different sets of parameters for OneMax, Royal Road, Deceptive Order-3, and Long Path problems. For each problem, they reported the result of 4 DE strategies which are: 1) any-change mutation and exponential crossover-DE/any/exp, 2) any-change mutation and binomial crossover-DE/any/bin, 3) restricted-change mutation and exponential crossover-DE/res/exp, and 4) restricted-change mutation and binomial crossover-DE/res/bin. We compare their best experimental results with our best parameters for each problem.
For each benchmark problem, we compare the performance of binary-adapted DE, SDE (simple real to binary conversion), and LZWDE. The result is shown in Figure 34. The X-axis shows the number of generations and the Y-axis shows the average-best fitness. In this Figure 34, the higher fitness indicates better result. LZWDE outperforms both DE and binary-adapted DE. Moreover, it is interesting to see that the performance of simple conversion is comparable to binary-adapted DE in Royal Road problem and better than binary-adapted DE in Long Path problem.

Table 5 shows the average evolution time. We ran the experiment on Intel Core i5 with 4GB of RAM. In this table, we report only the time that DE successfully finds the solution. The number in the parenthesis is the success rate. LZWDE can find the solution for every run. We do not have the data for binary-adapted DE. Therefore, we only compare the time of SDE and LZWDE. In LZWDE, there is an LZW decompression step. Even with an additional step, the algorithm can still find a solution faster than SDE. SDE cannot find a solution for Deceptive Order-3 problem.



(a) OneMax (b) Royal Road

(c) Deceptive order-3 (d) Long Path

Figure 34 Performance of DE, LZWDE, and Binary adapted DE on OneMax, Royal Road, Deceptive order-3, and Long Path problems

Table 5 Average evolution time (in milliseconds) of DE and LZWDE to solve OneMax, Royal Road, Deceptive order-3, and Long Path problems

Problem	**Algorithm**			
	SDE		**LZWDE**	
OneMax	388.43	(100)	6.50	(100)
Royal Road	36.42	(87)	29.10	(100)
Deceptive order-3	-	(0)	113.97	(100)
Long Path	13.53	(100)	45.69	(98.84)

2. Arithmetic Coding Differential Evolution (ACDE)

2.1 ACDE

We compare the performance of each optimization algorithm based on the quality of the solution. Optimization will stop when the optimum is found. However, for the case that the optimal is unknown, we are interested in the best fitness value obtained when the evolution reaches a specific number of generations. The results are presented in Table 6 to Table 8. The performance of ACDE is better than the simple real to binary conversion DE (SDE) in terms of solution quality and time. For Trap and NK Landscape problem, the higher the fitness value means the better solution. However, in these tables, the lower value is the better because we multiply the fitness function with -1. For Ising Spin Glass, the lower fitness value already means the better solution. ACDE can solve Random Trap, the solution of which as no pattern. Moreover, ACDE is better than SDE in Random Trap and NK Landscape.

Visualizations of Table 7 and Table 8 are in Figure 35. ACDE with local search gives the best result in both NK Landscape and Ising Spin Glass problems. In subfigure (a), the X-axis shows the number of K in NK Landscape problem. The larger K makes the problem more difficult. In subfigure (b), the X-axis shows the *width* of Ising Spin Glass model. The problem size is equal to $width^2$. ACDE with local search scales better than DE and ACDE.

Table 6 Performance of SDE, ACDE, and ACDE with local search on 50-bit Random Trap problem

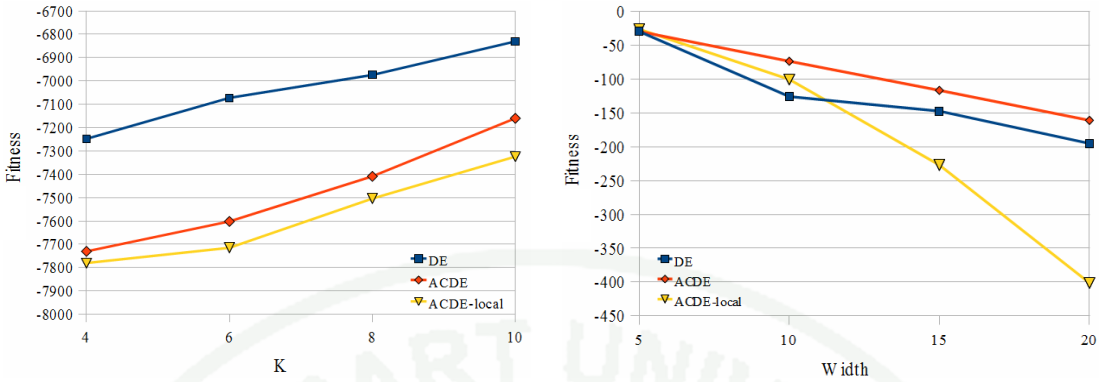
Algorithm	F	CR	Fitness	Eval.	Time (ms)	Found (%)
SDE	0.9	0.9	-48.40	99,450.00	171.50	27
ACDE	1.0	0.1	-50.00	12,107.27	30.17	100
ACDE-local	1.0	0.1	-50.00	17,912.57	31.20	100

Table 7 Performance of DE, ACDE, and ACDE with local search on 100-bit NK Landscape problem

Algorithm	Average run seed 0-29, F = 0.5, CR = 0.1					
	DE		ACDE		ACDE-local	
K	Fitness	Time (ms)	Fitness	Time (ms)	Fitness	Time (ms)
4	-7,248.08	835.67	-7,730.33	312.53	-7,780.43	325.20
6	-7,072.41	957.83	-7,602.02	340.60	-7,715.08	370.00
8	-6,974.24	1,113.83	-7,408.89	380.67	-7,504.06	428.37
10	-6,830.64	1,253.73	-7,160.05	422.23	-7,324.18	476.83

Table 8 Performance of DE, ACDE, and ACDE with local search on Ising Spin Glass problem

Algorithm	Average run seed 0-29					
	DE		ACDE		ACDE-local	
Width	Fitness	Time (ms)	Fitness	Time (ms)	Fitness	Time (ms)
5	-29.73	53.57	-29.67	10.60	-26.33	2.70
10	-125.87	787.30	-73.73	286.40	-100.52	28.83
15	-147.40	4,299.20	-116.67	2,439.27	-227.00	547.05
25	-195.2	13,503.43	-161.00	12,021.83	-401.11	4,793.59



(a) (b)

Figure 35 Performance of DE, ACDE, and ACDE-local on 100-bit NK Landscape and Ising Spin Glass problems

2.2 ACDE with Local Search

To improve the performance of ACDE even further, we add local search at the end of each iteration. The performance of ACDE with local search is compared with other DE adaptations and with BOA. Experimental results are shown in Table 9 LZWDE performs poorly but outperforms SDE only in 400- and 800-bit NK Landscape problem. Since we are interested in solving large problem, the focus would be at the result from 800-bit problems. At this problem size, adding compression can improve the effectiveness. ACDE can give solution with 7.45% better fitness value than SDE. Adding local search can improve the optimization even further. ACDE with hill climbing and ACDE with SA is 29.87% and 37.81% better than ACDE. When compared to sophisticated discrete optimization BOA, ACDE with SA can give 10.14% better quality and 28.61 times much faster than BOA.

Table 9 Comparison among ACDE with local search, other DE adaptations, and BOA. The bold numbers mean the optimal is found and the numbers in parentheses are the percentage of times that the optimal is found

Problem	Size	BOA		DE		LZWDE		ACDE		ACDE-HC		ACDE-SA	
		Fitness	**Time (ms)**	**Fitness**	**Time (ms)**	**Fitness**	**Time (ms)**	**Fitness**	**Time (ms)**	**Fitness**	**Time (ms)**	**Fitness**	**Time (ms)**
RandomMax	100	**-100** (100%)	3,049	**-100** (100%)	788	-83	300	-96	239	**-100** (100%)	39	**-100** (100%)	137
	400	-386	266,603	**-400** (100%)	12,472	-249	5,648	-304	10,790	**-400** (100%)	7,667	-331	10,989
	800	-745	762,947	-795	40,026	-471	26,631	-535	74,905	-700	75,016	-586	78,151
RandomTrap	100	-83	4,750	**-100.00** (13%)	601	-62	279	**-100** (100%)	261	**-100** (100%)	129	-96	260
	400	-306	262,649	-300	11,331	-192	5,089	-292	10,733	-345	10,648	-252	11,039
	800	-585	1,261,021	-563	40,719	-355	16,991	-479	76,910	-605	76,316	-601	80,022

Table 9 (Continued)

Problem	Size	BOA		DE		LZWDE		ACDE		ACDE-HC		ACDE-SA	
		Fitness	Time (ms)	Fitness	Time (ms)	Fitness	Time (ms)	Fitness	Time (ms)	Fitness	Time (ms)	Fitness	Time (ms)
Ising Spin Glass	100	-129	4,897	-123	677	-66	510	-128	226	-128	232	-122	189
	400	-436	564,240	-361	13,692	-214	10,643	-393	10,777	-421	10,831	-506	9,965
	900	-828	5,466,827	-354	57,572	-306	41,632	-603	110,970	-943	112,142	-1,104	107,265
NK Landscape	100	-7,681	5,141	-7,692	831	-7,402	348	-7,818	265	-7,850	257	-7,798	309
	400	-29,738	616,745	-27,343	14,928	-29,176	5,627	-29,515	11,147	-30,900	11,132	-28,371	11,353
	800	-57,021	3,119,407	-52,782	53,119	-58,227	19,562	-56,452	78,114	-59,833	81,644	-55,425	82,170

Discussion

In this section, we tried to explain why LZW encoding and Arithmetic Coding help improve the performance of DE.

1. Analysis of LZWDE

1.1 Fitness Landscape Analysis

1.1.1 Binary Fitness Landscape

The difficulty of a problem depends on two factors: the size of search space and the shape of fitness landscape. A problem with a larger search space is usually more difficult to solve. In addition, a problem with a more complex fitness landscape is more difficult. Example of complex fitness landscape is the one with many local optima or the one that leads evolutionary search away from the global optima.

To visualize the fitness landscape for a binary optimization problem, we enumerate all possible chromosomes, evaluate their fitness and measure distance from the solution, then plot the graph using the fitness and the distance. Figure 36(a) shows the fitness landscape of a 9-bit OneMax problem. The X-axis is the number of bits by which a chromosome differs from the solution. The Y-axis is the chromosome's fitness value. The darker area indicates a higher chromosome density. As shown in Figure 36(a), as the fitness increases, the chromosome is closer to the OneMax solution. Since evolutionary algorithm use fitness value to guide a search process, OneMax is an easy problem because the fitness value can guide the search to the correct direction.

On the contrary, Figure 36(c) shows the fitness landscape of a 9-bit Trap problem. The problem is more difficult to solve than OneMax because the fitness function deceives the search into moving away from the global optima. As the

fitness increase, the chromosome is more different from the solution. If we try to solve the Trap problem using a local search algorithm which produces a neighbor with 1 bit difference from the current position, the search will not be able to find the optimal solution.

Figure 36(a) and (c) visualize the fitness landscape of two extremes. We can easily tell from the graph which problem is easier. However, for a problem with difficulty in between, a subjective judgment should not be used to judge the complexity of fitness landscape. Therefore, we quantify the shape of a fitness landscape to one single number called fitness-distance correlation (*fdc*). We compute *fdc* or a correlation between fitness and distance using the formula given below.

$$fdc = \frac{\mathrm{cov}(F, D)}{\sigma(F)\sigma(D)} \tag{20}$$

where $cov(F,D)$ is a covariance of fitness F and distance D.

$\sigma(F), \sigma(D)$ is a standard deviation of F and D respectively.

From Equation (20), *fdc* value is in range [−1, 1].

- If *fdc* is 1 or a perfect positive correlation, it means that F and D always increase or decrease in the same direction. When F increases, D would also increase.
- If *fdc* is equal to 0 or no correlation, then there is no linear relationship between F and D.
- If *fdc* is −1 or a perfect negative correlation, it means that F and D increase or decrease in the opposite direction. For example, as the distance to a solution decreases, the fitness increases.

In this study, we prefer *fdc* value that is near −1 because as the chromosome is getting close to a solution (D is low), the fitness increases (F is high).

The *fdc* of the test problems is shown in Table 10. For the GA column, we enumerate all possible chromosomes, evaluate their fitness and measure distance from the solution. Note that, we do not use the GA for calculation but we represent GA for representing binary fitness landscape. OneMax's *fdc* is –1, which means that easy problem. The fitness guides to the solution. In the contrast, Deceptive problem has positive *fdc* which means that as the fitness increases, the chromosome is getting further from a solution. In this test, LZWDE is mostly better than the others.

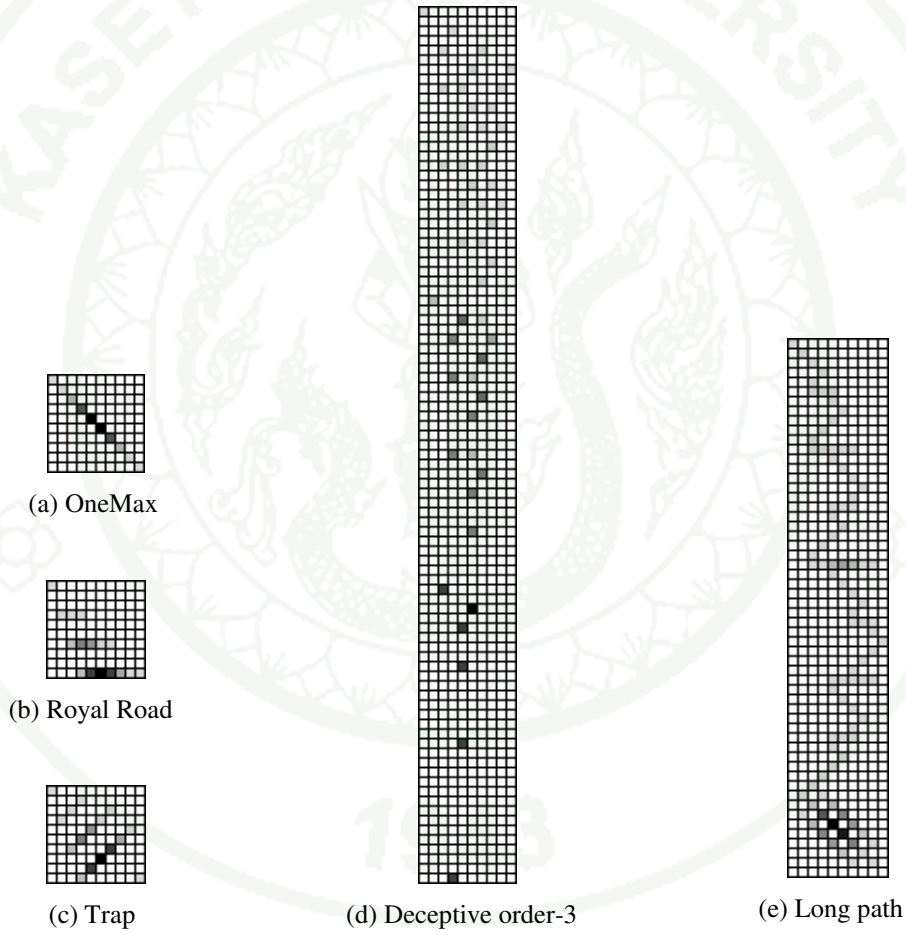


Figure 36 The fitness landscape for binary optimization problems which are (a) OneMax, (b) Royal Road, (c) Trap, (d) Deceptive-3, and (e) Long Path. The size of all problems is 9-bit

Table 10 The FDC of the test problem

Problem	Algorithm		
	GA	**SDE**	**LZWDE**
OneMax	-1.00	-0.63132	-0.78203
Royal Road	-0.65	-0.44755	-0.65961
Deceptive order-3	0.32	0.16866	-0.08296
Long Path	0.02	-0.00091	-0.07498

1.1.2 Real-value Fitness Landscape

Our study use DE to solve binary problem. DE use real value vectors. The *fdc* cannot be calculated using the same method as in the previous subsection because of we cannot enumerate all possible real-value vectors as we enumerate all possible binary chromosome. For a binary optimization problem, there are finite amount of chromosomes given a fixed length binary string. A problem size n bit has 2^n possible chromosome. However, a single real-value in a DE vector, in theory, can have infinitely uncountable possible values.

Since we cannot enumerate all possible chromosomes, we instead explore the fitness landscape using random walk. While an analysis procedure performs random walk, it records a fitness and distance to a solution. Each step of random walk imitates a trial vector generation process in DE.

$$vector_{t+1} = vector_t + F(random\_unit\_vector) \quad (21)$$

In this study, we set the value of F equals to 0.1 in order to make the step not too long. For each problem, an analysis procedure explores 100 random starting points. For each starting point, the procedure random walks for one million steps. A real value in the vector is constrained within the range [0, 1].

Another difference between binary and real value analysis is as follows. For a binary problem, we calculate a Hamming distance from a chromosome to an optimal solution. In DE, Euclidean distance is calculated. The distance calculation depends on how real-to-binary conversion is done. In SDE, the rule for converting is $X_i < 0.5$? 0 : 1. Therefore, if a one bit of binary solution is 1, and the corresponding real value is in the range [0.5, 1), the distance would be zero. Otherwise, the distance would be $0.5 - X_i$. If a binary solution is 0, the distance would be zero when the corresponding real value is in the range [0, 0.5). Otherwise, the distance would be $X_i - 0.5$.

Table 10 shows *fdc* for each problem. Real value *fdc* and binary *fdc* are different due to the way we measure the distance and perform the random walk (see Figure 37 and Figure 38).

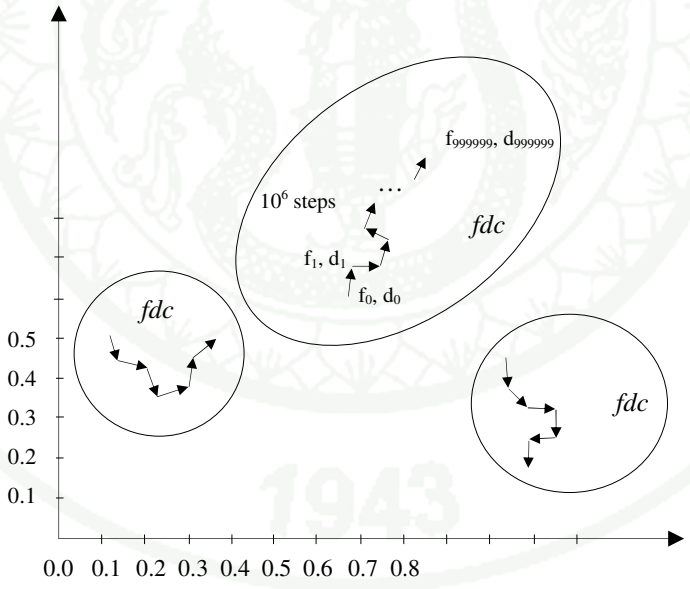


Figure 37 Visualization of random walking a fitness landscape. For each step, the fitness and distance to the solution are recorded for *fdc* calculation

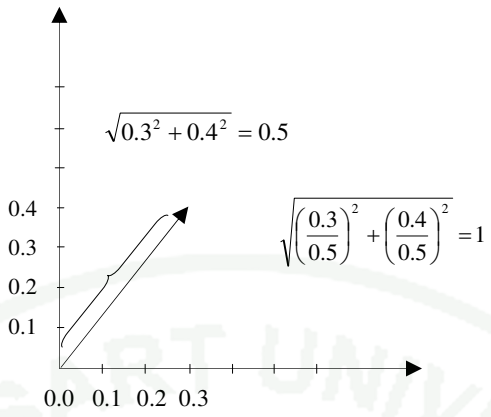


Figure 38 A random unit vector is obtained by random a vector, calculate the length, and divide every random number with the length

1.1.3 LZW Real-value Fitness Landscape

Although both SDE and LZWDE use real value vectors, the procedure to calculate the distance is different. In LZWDE, a real-value vector has to be converted to an array of integers before decompression and fitness evaluation. Thus, the distance calculation depends on how real to integer conversion is done. In this study, conversion is done simply by truncating a fraction part of a real number. An example of measuring the distance is as follows. Suppose that one integer in a solution array is 3. If the corresponding real value X_i is in the range [3, 4), the distance would be zero. If X_i is less than 3, then the distance would be $3 - X_i$. Otherwise, the distance would be $X_i - 4$. To calculate a distance of a vector to a solution vector, the Euclidean distance formula is used.

The random walk process is similar to the previous subsection. The difference is that each real value X_i is constrained to the range [0, i+2). The value within this range can be converted to a valid input for LZW decompression algorithm because it satisfies the constraint given in Equation (21).

For some problem such as OneMax, the original binary encoding has only one solution. However, when the problem is encoded with LZW, there might

be more than one solution. For example, an LZW chromosome of length 4 has 2 solutions for 9-bit OneMax problem. In that case, the minimum distance from a vector to both solutions is used to compute *fdc*.

Table 10 shows *fdc* for each problem. For each test problems, LZW encoding has lower *fdc* than the original encoding. This explains why LZWDE performs better than SDE.

2. Analysis of ACDE

An analysis procedure of ACDE is similar to LZWDE but the distance matrix is different. In both algorithms, Euclidean distance is calculated. The distance calculation depends on how real-to-binary conversion is done. A binary solution is compressed to a real vector *S*. In ACDE, a distance from a real vector *X* to *S* is measured by the following rules.

- If X_i is in the range $[S_i - half\_range, S_i + half\_range)$, then the distance would be zero. The value of *half_range* is $2^{compression\_ratio+1}$. This is because any value in the range would be decompressed to the same binary value.
- Otherwise, if X_i is less than $S_i - half\_range$, the distance would be $S_i - half\_range - X_i$.
- Otherwise, the distance would be $X_i - S_i + half\_range$.

In the case that there are many solutions, the minimum distance from X_i is used.

As shown in Table 11, for each problem, real value *fdc* and binary *fdc* are different due to the way we measure the distance and perform the random walk. The *fdc* of ACDC is lower than that of GA only in some cases. However, comparing *fdc* of different types of algorithm might not give much information about the effectiveness of each algorithm in solving a particular problem. For the same algorithm ACDE, when varying the compression ratio, higher ratio tends to have less

fdc. Moreover, higher ratio makes the vector smaller which helps the search process in two ways: the search space is smaller and the less running time per iteration.

Table 11 Fitness-distance correlation for binary landscape and compressed real vector landscapes (varied by compression ratios)

Problems	**GA (16 bits)**	**AC1 (16 doubles)**	**AC2 (8 doubles)**	**AC4 (4 doubles)**	**AC8 (2 doubles)**	**AC16 (1 double)**
RandomMax	-1.0000	-0.6248	-0.5756	-0.4359	-0.3131	-0.2805
Ising	-0.4488	-0.2069	-0.1811	-0.2094	-0.1322	-0.3048
NK	-0.2245	-0.1393	-0.1602	-0.1662	-0.1814	-0.2952
RandomTrap	0.3402	0.1940	0.1972	0.1200	0.0579	0.0821

CONCLUSION AND RECOMMENDATION

Conclusion

This study aimed to investigate compressed encoding in Evolutionary Algorithm. We applied two compression algorithms, Lempel-Ziv-Welch compression (LZW) and Arithmetic Coding (AC), to an efficient real value optimizer, Differential Evolution (DE). The results are binary optimizers called LZWDE and ACDE respectively. Therefore this work can be considered as an adaptation of DE to binary optimization. In addition to compressed encoding, we proposed simple conversion using the rule $X_i < 0.5 \ ? \ 0 : 1$ and called the scheme SDE.

LZWDE combines LZW and DE, which are two completely different types of algorithm together. LZW is a compression algorithm. DE is a real value optimizer. This combination allows DE to solve binary optimization problems. When compared to previously DE adaptation, LZWDE can solve a problem in a very short amount of time. Moreover, it scales better than SDE and binary adapted DE. This study analyzes LZWDE by exploring the fitness landscape using random walk. The random walk imitates the trial vector generation in DE. We also proposed two distance metrics, one for SDE and another for LZWDE, to analyze simple real-to-binary conversion and LZW encoding. These metrics is used with a neighborhood function to compute fitness distance correlation (*fdc*). The result shows that, for the benchmark problems, LZW encoding not only reduce the search space, but it also simplifies the fitness landscape. However, LZW encoding has a major disadvantage. Its performance of LZW encoding is decreased in the random version of a benchmark problem while the performance of uncompressed encoding remains the same.

AC compression was applied to DE to solve discrete optimization problems. The resulting algorithm is called ACDE. We applied local search to ACDE and tested them with random version of existing benchmarks. Experimental results indicate that the algorithm outperforms BOA, which is one of the best multivariate EDAs, in term of solution quality and running time. This study also analyzed ACDE by random

walking the fitness landscape. We proposed another distance metrics to measure the distance from a real vector to a solution vector. The random walk and the metric were used to compute *fdc*. The result suggested that higher compression ratio should be used in ACDE.

Recommendation

1. Generally, AC encoding should be used rather than LZW encoding.

2. The higher compression ratio is recommended in ACDE.

3. Another real value optimizer, such as ES, can be transformed to a binary optimizer using our approach.

LITERATURE CITED

Ackley, D. H. 1987. **A Connectionist Machine for Genetic Hillclimbing**. Kluwer Academic, Boston.

Ahn, C.W. and H-. TaeKim. 2009. Estimation of Particle Swarm Distribution Algorithms Bringing Together the Strengths of PSO and EDAs, pp. 1817-1818. *In* **Proceedings of the 11th Annual conference on Genetic and evolutionary computation (GECCO'09)**. 8-12 July 2009, Canada.

Alvarez, P.M. and A.H. Aguirre. 2008. The Directional EDA for Global Optimization, pp. 473-474. *In* **Proceedings of the 10th annual conference on Genetic and evolutionary computation (GECCO'08)**. 12-16 July, Atlanta, Georgia, USA.

Baluja, S. 1994. **Population-based incremental learning: A method for integrating genetic search based function optimization and competitive learning**. Technical Report CMU-CS-94-163, Carnegie Mellon University, Pittsburgh, Pennsylvania.

_____ and R. Caruana. 1995. Removing the Genetics from the Standard Genetic Algorithm. *In* **Proceedings of the 12th International Conference on Machine Learning**. Lake Tahoe, CA.

_____, S. Davies and S. 1997. **Using optimal dependency trees for combinatorial optimization: Learning the structure of search space**. Technical Report No. CMU-CS-97-107, Carnegie Mellon University, Pittsburgh, Pennsylvania.

Beyer, H. -G. and H. -P. Schwefel. 2002. Evolution Strategies: A Comprehensive Introduction. **Journal Natural Computing** 1(1): 3-52.

Chamlamai, S. and W. Suwannik. 2007. Benchmark Problems for Compressed Genetic Algorithm, pp. 653-656. *In* **Proceedings of Electrical Engineering Conference (EECON-30)** (abstract in English).

De Bonet, J.S., C.L. Isbell and P. Viola. 1997. MIMIC: Finding optima by estimating probability densities. **Advances in Neural Information Processing Systems** 9.

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

Gong, T. and A. Tuson. 2007. Differential Evolution for Binary Encoding. **Soft Computing in Industrial Applications** 39: 251-262.

Handa, H. 2006. The effectiveness of mutation operation in the case of Estimation of Distribution Algorithms. **Biosystems** 87: 243-251.

Harik, G.R., F.G. Lobo and K. Sastry. 2006. Linkage Learning via Probabilistic Modeling in the Extended Compact Genetic Algorithm (ECGA). **Studies in Computational Intelligence (SCI)** 33: 39-61.

_____, F.G. Lobo and D.E. Golberg. 1998. The compact genetic algorithm, pp. 523-528. *In* **Proceedings of the IEEE Conference on Evolutionary Computation**.

Holland, J. H. 1975. **Adaptation in Natural and Artificial Systems**. University of Michigan Press, 2nd ed., MIT Press, 1992.

Horn, J., D. E. Goldberg and K. Deb. 1994. Long Path Problems. **Lecture Notes in Computer Science** 866: 149-158.

Ingber, L. 1993. Simulated annealing: Practice versus theory. **Mathematical and Computer Modelling** 18(11): 29-57.

Kindermann, R. and J. L. Snell. 1980. **Markov Random Fields and Their Applications**. AMS.

Kirkpatrick, S., C.D. Gelett and M.P. Vecchi. 1983. Optimization by simulated annealing. **Science** 220: 621-630.

Koza, J. R. 1990. **Genetic programming: a paradigm for genetically breeding populations of computer programs to solve problems**. Technical Report STANCS-90-1314, Department of Computer Science, Stanford University.

_____. 1992. **Genetic Programming on the Programming of Computers by Means of Natural Selection**. Cambridge, MA: MIT Press.

Kunasol, N, W. Suwannik and P. Chongstitvatana. 2005. LZW-Encoding in Genetic Algorithm, pp. 861-864. *In* **Proceedings of Electrical Engineering Conference (EECON-28)**.

_____, W. Suwannik and P. Chongstitvatana. 2006. Solving One-Million-Bit Problems Using LZWGA, pp. 32-36. *In* **Proceedings of International Symposium on Communications and Information Technologies (ISCIT)**.

Li, W., X. Rong, C. Jian and L. Yong. 2008. An Estimation Distribution Algorithm of Optimum Path Planning for Mobile Robots, pp. 549-553. *In* **Proceedings of the 2008 International Conference on Cyberworlds**, Publisher IEEE Computer Society Washington, DC, USA.

Lichtblau, D. 2012. Differential Evolution in Discrete Optimization. **International Journal of Swarm Intelligence and Evolutionary Computation** 1.

Mitchell, M. 1998. **An Introduction to Genetic Algorithms**. MIT Press.

_____, J. Holland and S. Forrest. 1994. When will a genetic algorithm outperform hill climbing?. **Advances in Neural Information Processing Systems** 6: 51-58.

Muelas, S., A. LaTorre and J. Peña. 2009. A Memetic Differential Evolution Algorithm for Continuous Optimization, pp. 1080-1084. *In* **Proceedings of the 9th International Conference on Intelligent Systems Design and Applications**. IEEE.

Mühlenbein, H. 1998. The equation for response to selection and its use for prediction. **Evolutionary Computation** 5: 303-346.

_____ and G. Paaß. 1996. From recombination of genes to the estimation of distributions I. Binary parameters, pp. 178-187. *In* **Lecture Notes In Computer Science (LNCS), Proceedings of the 4th International Conference on Parallel Problem Solving from Nature**.

_____ and T. Mahnig. 1999. Convergence theory and applications of the factorized distribution algorithm. **Journal of Computing and Information Technology**. 7: 19-32.

Numnark S. and W. Suwannik. 2008. Improving the Performance of LZWGA By Using A New Mutation Method, pp. 1862-1865. *In* **Proceedings of the IEEE World Congress on Computational Intelligence (CEC 2008)**.

Paul, T. K. and H. Iba. 2002. Linear and combinatorial optimizations by estimation of distribution algorithms, pp. 99-106. *In* **Proceedings of the 9th MPS Symposium on Evolutionary Computation**, IPSJ, Japan.

Pelikan, M., D.E. Goldberg and E. Cantú-Paz. 1998. **Linkage Problem, Distribution Estimation, and Bayesian Networks**. IlliGAL Report No. 98013, November.

_____, _____, _____. 1999. BOA: The Bayesian Optimization Algorithm, pp. 525-532. *In* **Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-99)**.

_____ and H. Mühlenbein. 1999. The bivariate marginal distribution algorithm, pp. 521-535. **Advances in Soft Computing-Engineering Design and Manufacturing**.

_____. 2010. NK Landscapes, Problem Difficulty, and Hybrid Evolutionary Algorithms, pp. 665-672. *In* **Proceedings of the 12th annual conference on Genetic and evolutionary computation (GECCO'10)**. Portland, Oregon, USA.

Price, K. and R. Storn. 1997. **Differential Evolution**. Available Source: http://www.drdobbs.com/architecture-and-design/184410166, July 8, 2012.

_____, _____ and J. A. Lampinen. 2005. **Differential Evolution: A Practical Approach to Global Optimization**. Springer.

Qin, A. K., V. L. Huang and P. N. Suganthan. 2009. Differential Evolution Algorithm with Strategy Adaptation for Global Numerical Optimization. **IEEE Transactions on Evolutionary Computation** 13: 398-417.

Sastry, K., D. Goldberg and X. Llorà. 2007. Towards billion bit optimization via efficient genetic algorithms. **Complexity** 12(3): 27-29.

Sayood, K., 2006. **Introduction to Data Compression**. 3rd Edition. Morgan Kaufmann Publishers.

Shi, G. and Q. Ren. 2008. Research on Compact Genetic Algorithm in Continuous Domain, pp. 793-800. *In* **Proceedings of the IEEE Congress on Evolutionary Computation (CEC 2008)**.

Storn, R. and K. Price. 1997. Differential Evolution - A simple and efficient adaptive scheme for global optimization over continuous spaces. **Journal of Global Optimization** 11(4): 341-359.

Sun, J., Q. Zhang and E.P.K. Tsang. 2005. DE/EDA: a new evolutionary algorithm for global optimization. **Information Sciences-Informatics and Computer Science: An International Journal** 169: 249-262.

Suwannik, W., N. Kunasol and P. Chongstitvatana. 2005. Compressed Genetic Algorithm, pp.203-211. *In* **Proceedings of Northeastern Computer Science and Engineering Conference** (abstract in English).

_____ and P. Chongstitvatana. 2008. Solving One-Billion Bit Noisy OneMax Problem using Estimation Distribution Algorithm with Arithmetic Coding, pp. 1203-1206. *In* **Proceedings of IEEE Congress on Evolutionary Computation (CEC)**.

Uludag, G. and A.S. Uyar. 2009. Fitness landscape analysis of differential evolution algorithms, pp. 1-4. *In* **Proceedings of Soft Computing, Computing with Words and Perceptions in System Analysis, Decision and Control (ICSCCW 2009)**.

Vesterstrøm, J. and R. Thomsen. 2004. A Comparative Study of Differential Evolution, Particle Swarm Optimization, and Evolutionary Algorithms on Numerical Benchmark Problems, pp. 1980-1987. *In* **Proceedings of the IEEE International Congress on Evolutionary Computation**.

Watchanupaporn, O. and W. Suwannik. 2010. An Estimation of Distribution Algorithm using the LZW Compression Algorithm, pp. 97-102. *In* **Proceedings of the 2nd International Conference on Advanced Cognitive Technologies and Applications (COGNITIVE 2010)**, Lisbon, Portugal.

_____ and W. Suwannik. 2011. LZW Mutual-Information-Maximizing Input Clustering Algorithm, pp. 2273-2276 *In* **Proceddings of the International Conference on Biomedical Engineering and Informatics**, Shanghai, China.

_____, W. Suwannik and P. Chongstitvatana. 2012. Mutation in Compressed Encoding in Estimation of Distribution Algorithm, pp. 308-311. *In* **Proceedings of the International Conference on Genetic and Evolutionary Computing (ICGEC 2012)**, Japan.

_____ and W. Suwannik. 2012. LZW Differential Evolution for Binary Encoding, pp. 51-54. *In* **Proceedings of the International Conference on Advanced Topics in Artificial Intelligence (ATAI 2012)**, Singapore.

_____ and W. Suwannik. 2012. Arithmetic Coding Differential Evolution for Binary Encoding. **Advances in Information Technology and Applied Computing (AITAC)** 1: 155-158.

_____ and W. Suwannik. Arithmetic Coding Differential Evolution with Local Search. **Advanced Science Letter**. ISSN 1936-6612 (accepted for publication).

_____ and W. Suwannik. Analysis of LZW Differential Evolution for Binary Encoding. **GSTF Journal on Computing**. ISSN 2251-3043 (accepted for publication).

Welch, T. A. 1984. A Technique for High-Performance Data Compression. **IEEE Computer** 17(6): 8-19.

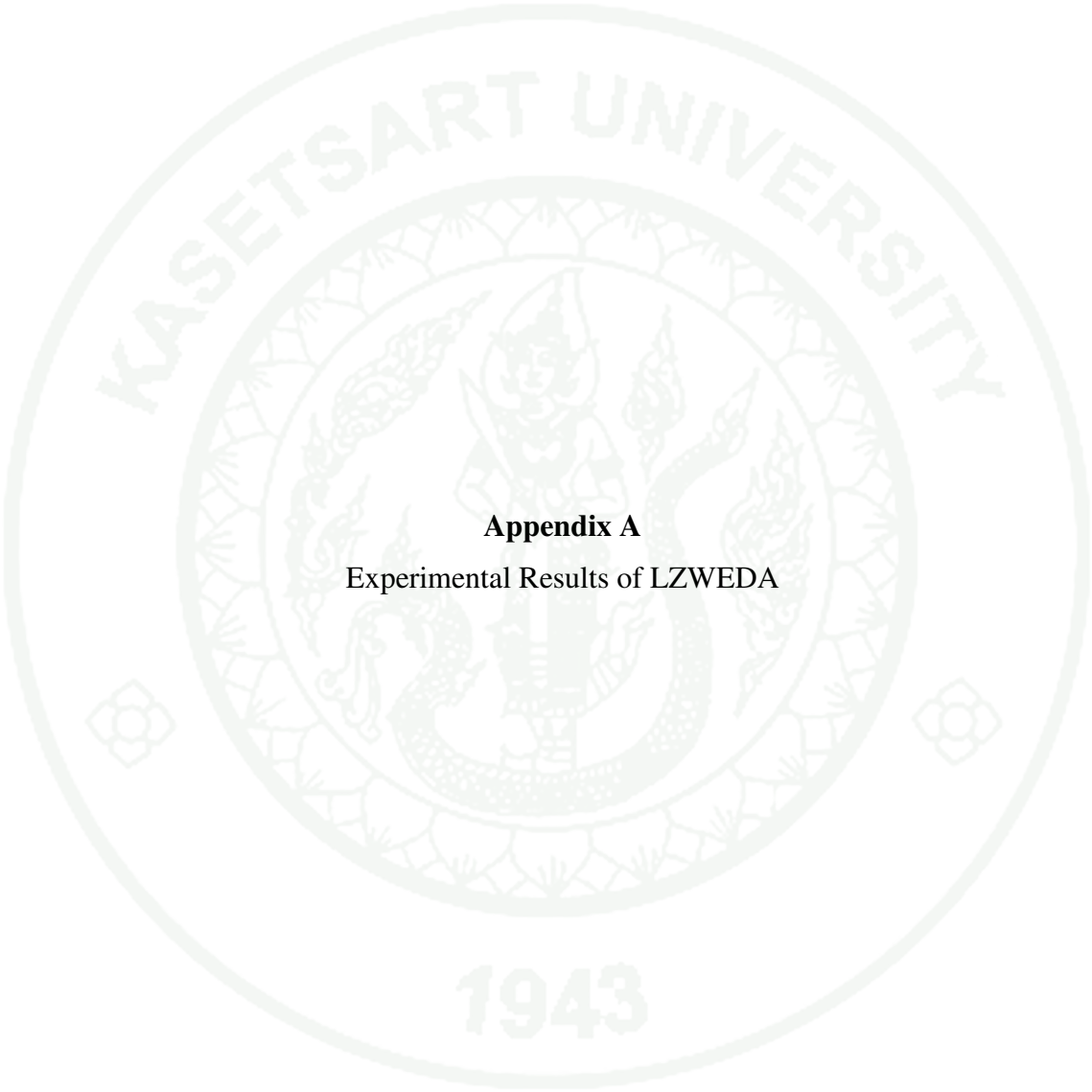
Wolfram MathWorld. 2012. **Differential Evolution**. Available Source: http://mathworld.wolfram.com/DifferentialEvolution.html, December 11, 2012.

Yokoo, H. 1992. Improved Variations Relating the Ziv-Lempel and Welch-Type Algorithm for Sequential Data Compression. **IEEE** 30(1): 73-81.

Yong, Z. and N. Sannomiya. 2001. An improvement of genetic algorithms by search space reductions in solving large-scale flowshop problems. **T.IEE Japan** 121-c(6): 1010-1015.

Zhang, Q., J. Sun, E. Tsang and J. Ford. 2004. Hybrid Estimation of Distribution Algorithm for Global Optimization. **Engineering Computations** 21: 91-107.

APPENDICES



Appendix A

Experimental Results of LZWEDA

Appendix Table A1 Experimental parameters of LZWEDA

Parameter	Value
Population size	1000
Problem size (bits)	100, 400, 800
Chromosome compression ratio (for LZWEDA)	1/2, ×1, ×2 of problem size
Tournament size (for LZWCGA)	2
Tournament size (for LZWMIMIC)	16
Tournament size (for LZWBOA)	4
Mutation rate (for LZWMIMIC)	0.5%
Maximum fitness evaluations (for LZWCGA)	100000
Max generation (for LZWMIMIC, LZWBOA)	200

Appendix Table A2 Performance of CGA and LZWCGA

Problem	Problem Size	Time (ms)			
		cGA	LZWCGA		
			0.5x	1x	2x
OneMax	100	119.60	1,826.77	50.97	7.73
	400	988.83	7,002.00	893.07	124.73
	800	2,959.50	13,859.10	3,006.20	364.00
Trap	100	516.97	1,917.73	63.96	9.73
	400	2,012.07	7,051.00	13,899.67	235.20
	800	4,078.17	13,966.27	1,725.00	493.82
FourPeak	100	539.67	1,900.93	37.50	7,110.03
	400	2,200.67	6,984.93	13,854.60	27,370.43
	800	4,364.13	13,764.37	27,275.03	53,891.90
SixPeak	100	315.00	832.13	70.37	2,962.87
	400	1,238.30	2,967.77	1,470.91	11,252.67
	800	2,480.73	5,806.53	11,301.77	22,429.20
NK Landscape (K=2)	100	304.23	564.57	1,009.87	1,808.07
	400	1,195.37	2,088.83	3,810.67	7,096.60
	800	2,482.87	3,997.10	7,195.47	13,287.70
NK Landscape (K=4)	100	346.33	610.50	1,046.23	1,831.97
	400	1,430.00	2,220.40	3,856.33	6,882.73
	800	2,884.93	4,496.47	7,509.83	7,509.83
NK Landscape (K=6)	100	408.70	665.60	1,108.13	1,904.77
	400	1,673.37	2,394.07	4,087.23	7,228.53
	800	3,404.43	4,686.27	7,997.60	14,143.50

Appendix Table A3 Performance of CGA and LZWCGA (averaged only runs that found the optimal). The numbers in parenthesis are the percentage of runs that optima are found. In its absence, the algorithm can always find the optima

Problem	Problem Size	Time (ms)			
		cGA	LZWCGA		
			0.5x	1x	2x
OneMax	100	119.60	-	50.97	7.73
	400	988.83	-	893.07	124.73
	800	2,959.50	-	3,006.20	364.00
Trap	100	-	-	(80) 63.96	9.73
	400	-	-	-	235.20
	800	-	-	(3.33) 1,725.00	(93.33) 493.82
FourPeak	100	(20.00) 539.67	-	(6.67) 37.50	-
	400	-	-	-	-
	800	-	-	-	-
SixPeak	100	(3.33) 315.00	-	70.37	-
	400	-	-	(36.67) 1,470.91	-
	800	-	-	-	-

Appendix Table A4 Performance of MIMIC and LZWMIMIC averaged over 30 runs

Problem	Problem Size	Time (ms)			
		MIMIC	LZWMIMIC		
			0.5x	1x	2x
OneMax	100	599.90	4,759.00	195.57	446.87
	400	35,147.50	93,769.43	6,854.33	11,609.90
	800	495,394.40	260,757.00	45,789.73	92,526.97
Trap	100	14,877.87	5,558.73	180.10	444.47
	400	382,682.83	94,977.30	6,755.83	11,617.30
	800	2,108,066.57	312,075.00	74,822.30	99,271.63
FourPeak	100	3,526.30	426.33	3,994.00	36,011.20
	400	391,920.83	10,054.00	202,994.10	2,601,477.40
	800	2,225,756.97	110,180.00	1,729,770	12,079,990.5
SixPeak	100	2,916.40	2,260.50	379.60	91,255.40
	400	390,226.80	88,313.30	27,537.50	2,203,625.70
	800	2,086,227.50	424,466.10	272,565.17	15,143,376.20
NK Landscape (K=2)	100	7,452.97	2,752.23	9,187.93	34,948.83
	400	171,793.63	34,922.20	194,139.73	2,912,653.10
	800	2,664,056.00	200,407.83	2,899,188.67	11,423,891.00
NK Landscape (K=4)	100	7,348.37	2,746.67	9,062.57	34,195.77
	400	186,196.53	35,030.37	212,201.47	2,933,411.33
	800	2,764,056.79	204,721.57	2,883,761.71	11,249,154.33
NK Landscape (K=6)	100	7,822.70	2,956.57	9,436.30	35,043.47
	400	174,365.80	35,709.50	212,045.97	2,877,802.70
	800	2,757,302.67	196,402.77	2,850,004.36	11,029,222.00

Appendix Table A5 Performance of MIMIC and LZWMIMIC (averaged only runs that found the optimal). The numbers in parenthesis are the percentage of runs that optima are found. In its absence, the algorithm can always find the optima

Problem	Problem Size	Time (ms)			
		MIMIC	LZWMIMIC		
			0.5x	1x	2x
OneMax	100	599.90	(3.33) 4,759.00	195.57	446.87
	400	35,147.50	-	6,854.33	11,609.90
	800	495,394.40	(6.67) 260,757.00	45,789.73	92,526.97
Trap	100	-	-	180.10	444.47
	400	-	-	6,755.83	11,617.30
	800	-	(13.33) 312,075.00	74,822.30	99,271.63
FourPeak	100	3,526.30	(30.00) 426.33	(20.00) 3,994.00	-
	400	-	(10.00) 10,054.00	-	-
	800	-	(20.00) 110,180.00	-	-
SixPeak	100	2,916.40	(20.00) 2,260.50	379.60	-
	400	-	-	(80.00) 27,537.50	-
	800	-	-	(60.00) 272,565.17	-

Appendix Table A6 Performance of BOA and LZWBOA averaged over 30 runs. The letter X signifies the program cannot run on our machine because of memory constraints

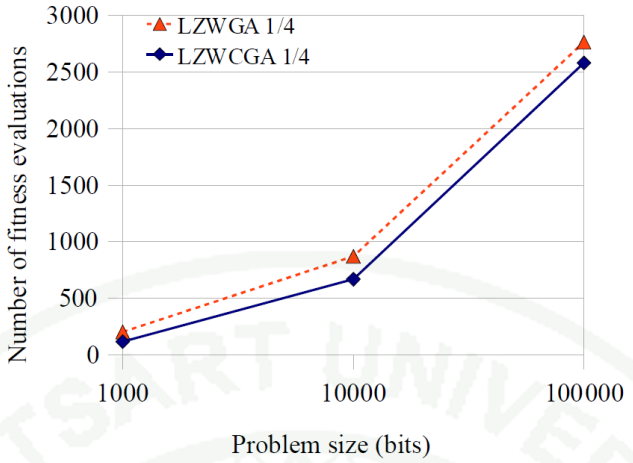
Problem	Problem Size	Time (ms)			
		BOA	LZWBOA		
			0.5x	1x	2x
OneMax	100	2,229.76	545.07	4.67	0.53
	400	82,919.20	20,328.97	24,823.40	33.27
	800	476,755.03	119,434.7	173,962.00	63.50
Trap	100	3,081.57	560.50	168.47	9.90
	400	99,951.93	21,560.23	28225.67	3,565.30
	800	580,244.77	121,534.73	190,229.80	63.93
FourPeak	100	9,146.40	1,486.60	35,422.07	106,733.77
	400	382,268.97	99,648.87	869,077.93	3,188,924.00
	800	2,031,587.00	599,292.93	4,193,013.03	X
SixPeak	100	5,538.00	1,421.67	36,004.63	76,354.50
	400	375,679.00	102,109.23	915,661.90	3,180,857.73
	800	2,056,924.67	601,390.47	4,124,909.96	X
NK Landscape (K=2)	100	2,905.27	1,390.87	19,206.83	119,137.93
	400	108,427.33	63,882.93	509,877.93	3,266,010.33
	800	715,760.67	384,001.33	2,882,068.03	X
NK Landscape (K=4)	100	4,830.23	1,518.93	20,246.90	119,857.13
	400	308,453.90	69,092.83	542,416.47	3,177,564.00
	800	2,145,275.33	403,911.73	2,984,521.23	X
NK Landscape (K=6)	100	7,430.80	1,583.37	21,089.80	121,737.47
	400	483,877.13	69,131.30	537,769.60	3,169,506.17
	800	2,880,123.20	407,599.47	3,008,901.03	X

Appendix Table A7 Performance of BOA and LZWBOA (averaged only runs that found the optimal). The numbers in parenthesis are the percentage of runs that optima are found. In its absence, the algorithm can always find the optima

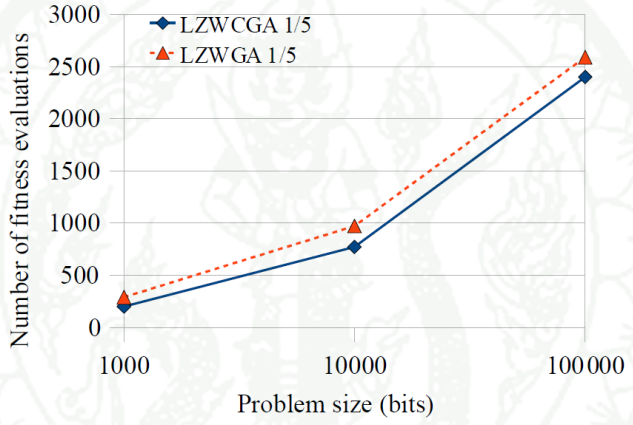
Problem	Problem Size	Time (ms)			
		BOA	LZWBOA		
			0.5x	1x	2x
OneMax	100	(56.67) 2,229.76	545.07	4.67	0.53
	400	-	20,328.97	24,823.40	33.27
	800	-	119,434.7	173,962.00	63.50
Trap	100	-	560.50	168.47	9.90
	400	-	21,560.23	28225.67	3,565.30
	800	-	121,534.73	190,229.80	63.93
FourPeak	100	-	-	-	-
	400	-	-	-	-
	800	-	-	-	X
SixPeak	100	-	-	-	(3.33) 76,354.50
	400	-	-	-	-
	800	-	-	-	X

Appendix Table A8 Average fitness evaluations of cGA and LZWCGA on robot arm control programs

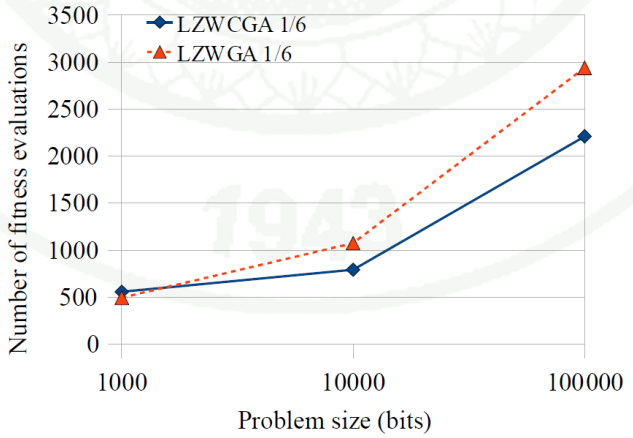
Algorithm	Chromosome Length	Success	Fitness Calculations
cGA	2400	90%	28,658.81
LZWCGA × 1/2	1200	80%	36,392.75
LZWCGA × 1	2400	100%	6,183.07
LZWCGA × 2	4800	100%	8,290.67



(a) Population size is 128. The compression ratio is 1/4

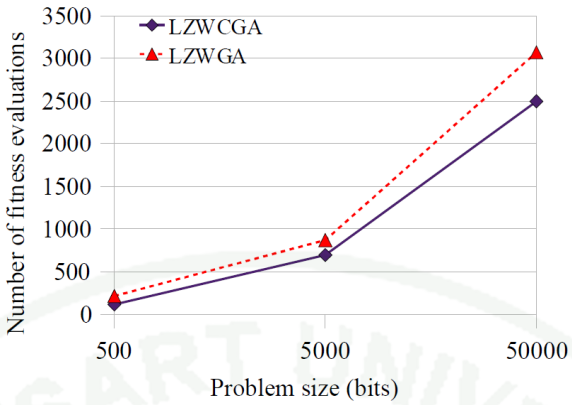


(b) Population size is 512. The compression ratio is 1/5

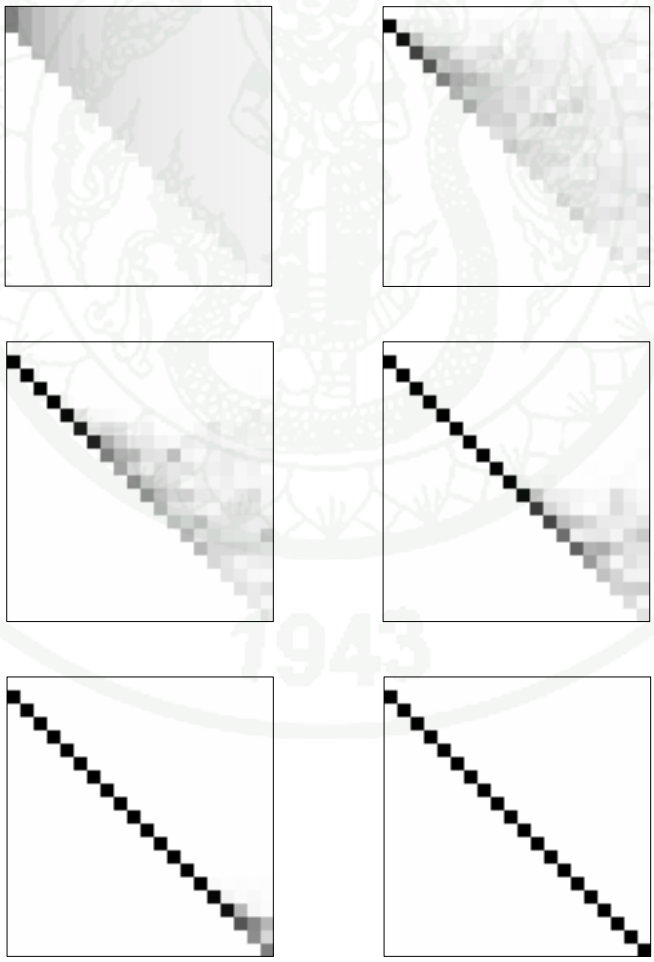


(c) Population size is 1024. The compression ratio is 1/6

Appendix Figure A1 The number of fitness evaluations of LZWCGA and LZWGA when solving various sizes of OneMax problem

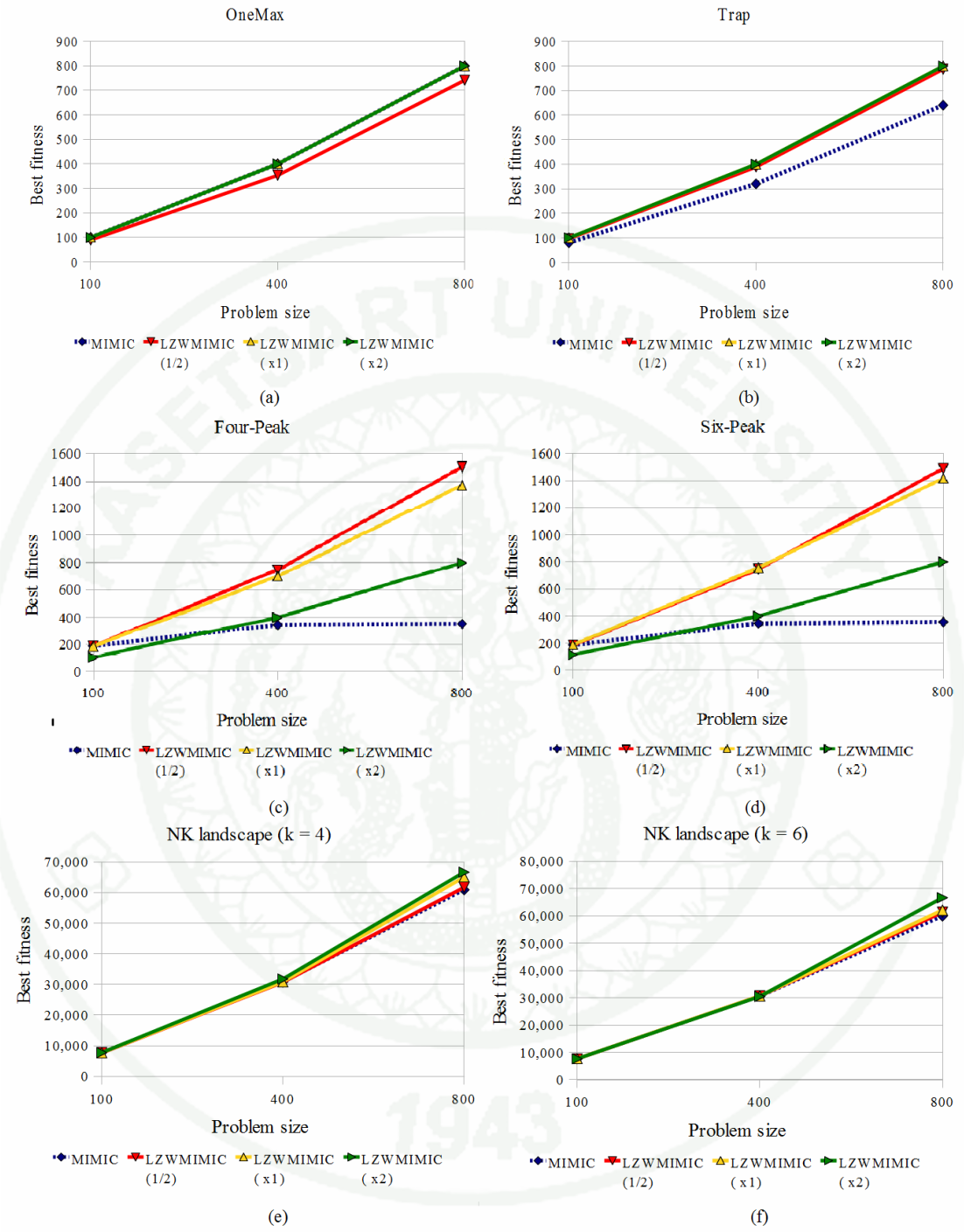


Appendix Figure A2 The number of fitness evaluations when using LZWCGA and LZWGA to solve Trap problem. The compression ratio is 1/4

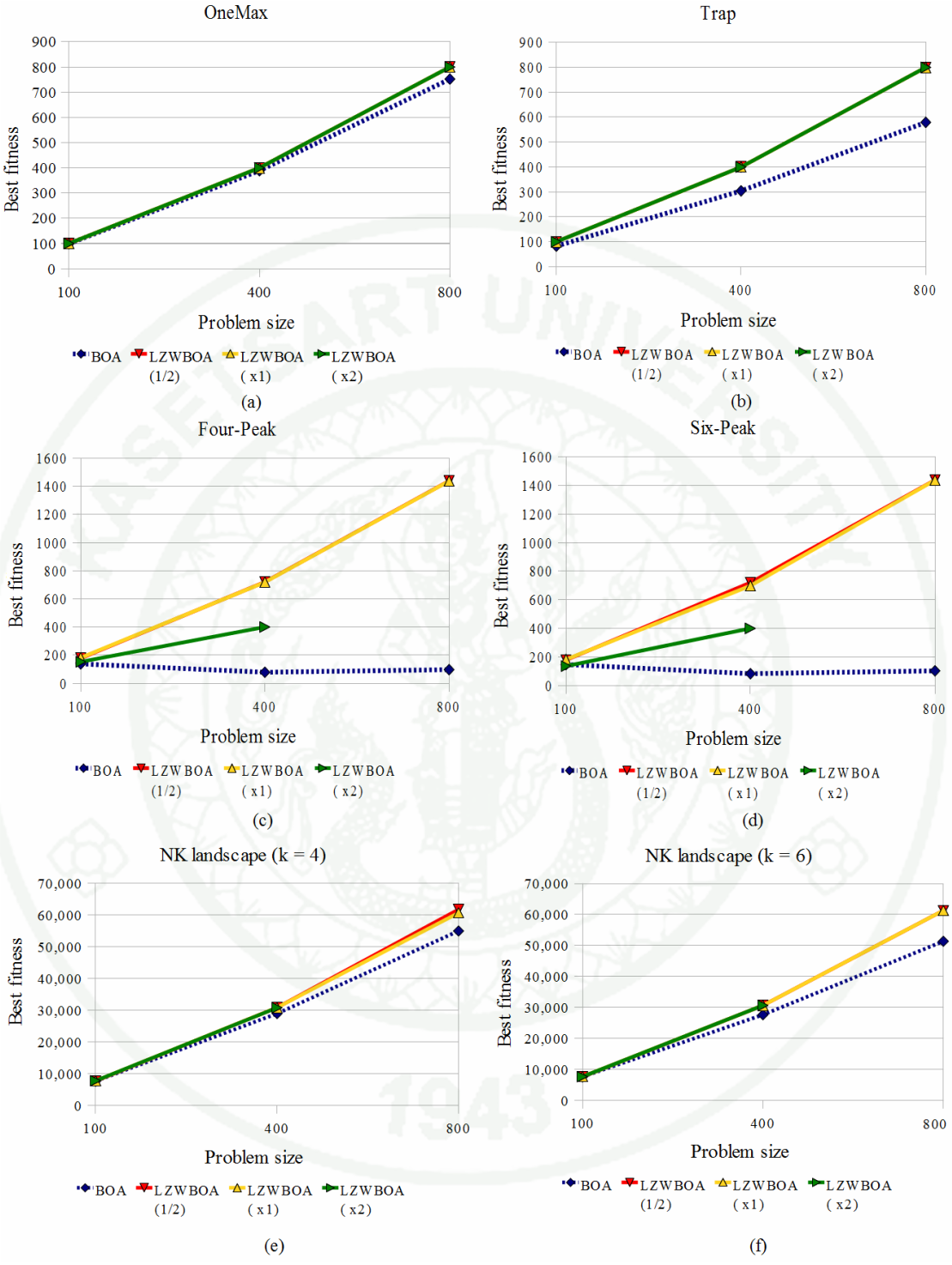


Appendix Figure A3 A visual representation for a probability matrix at 0, 10000, 20000, 30000, 40000, and 50000 fitness evaluations

Appendix Figure A4 Average best fitness value of cGA and LZWCGA for the OneMax (a), Trap (b), Four-Peak (c), Six-Peak (d) and NK landscape (e)(f) problems

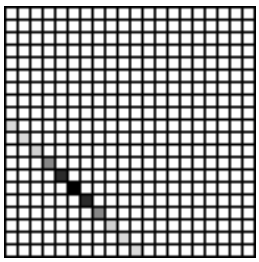


Appendix Figure A5 Average best fitness value of MIMIC and LZWMIMIC for the OneMax (a), Trap (b), Four-Peak (c), Six-Peak (d) and NK Landscape (e)(f) problems

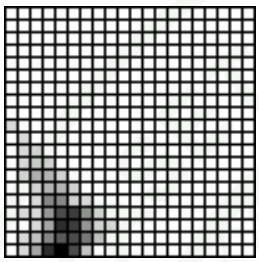


Appendix Figure A6 Average best fitness value of BOA and LZWBOA for the OneMax (a), Trap (b), Four-Peak (c), Six-Peak (d) and NK landscape (e) (f)

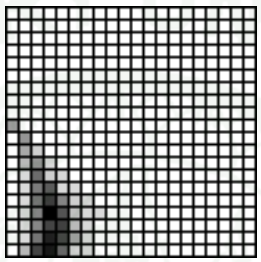
Appendix Figure A7 Average best fitness value of LZWEDA for the NK landscape vary by *K* value



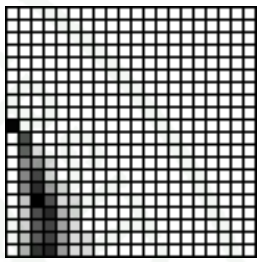
(a) 10-bit uncompressed chromosome



(b) 8-bit LZW chromosome

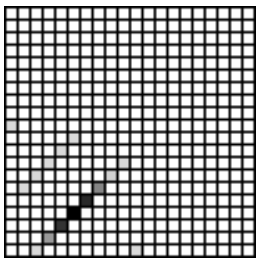


(c) 11-bit LZW chromosome

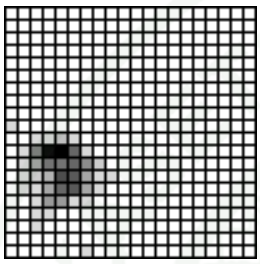


(d) 14-bit LZW chromosome

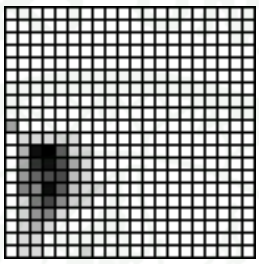
Appendix Figure A8 Fitness landscape of the OneMax problem for ordinary and LZW chromosomes. The X-axis is the number of bits by which a chromosome differs from the solution. The Y-axis is the chromosome's fitness value. The darker area indicates a higher chromosome density



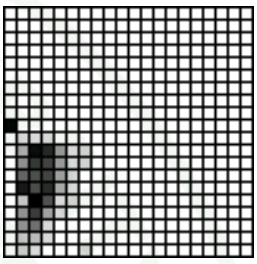
(a) 10-bit uncompressed chromosome



(b) 8-bit LZW chromosome

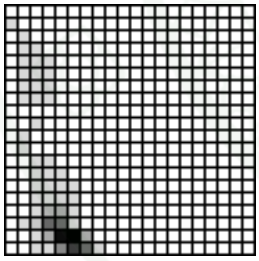


(c) 11-bit LZW chromosome

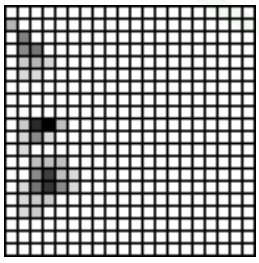


(d) 14-bit LZW chromosome

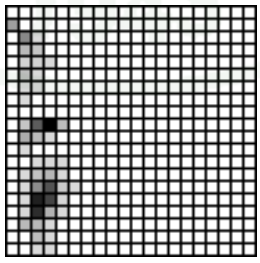
Appendix Figure A9 Fitness landscape of the Trap problem for ordinary and LZW chromosomes



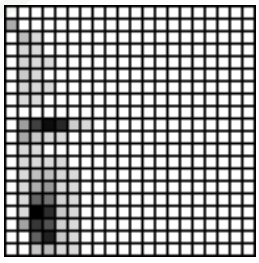
(a) 10-bit uncompressed chromosome



(b) 8-bit LZW chromosome



(c) 11-bit LZW chromosome



(d) 14-bit LZW chromosome

Appendix Figure A10 Fitness landscape of the Four-Peak problem for ordinary and LZW chromosomes

Appendix B

Publications

An Estimation of Distribution Algorithm using the LZW Compression Algorithm

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

***Abstract*-This paper proposes a new evolutionary algorithm called LZWCGA. LZWCGA is an algorithm that combines the LZW compressed chromosome encoding and compact genetic algorithm (cGA). The advantage of LZW encoding is to reduce the search space thus speed up the evolutionary search. cGA is one of Estimation of Distribution Algorithms. Its advantage is compact representation of the whole binary-string genetic algorithm population.**

Keywords-Estimation of Distribution Algorithms; Lempel-Ziv-Welch Algorithm; Compression Algorithm; Compact Genetic Algorithm

I. Introduction

Genetic Algorithm (GA) is an algorithm that solves problems by simulating natural evolution [1]. To solve a problem using GA, a candidate solution must be encoded into a binary string. The length of this string represents the size of the problem. As the length of the binary string increases, the size of the search space also increases at an exponential rate. For example, the size of search space for 10-bit chromosome is 2^{10}. While the size of search space for 100-bit chromosome is 2^{100}.

To reduce the search space, one approach is to utilize a compressed encoding chromosome. Kunasol et. al. proposed LZWGA, which is a GA that uses LZW compressed chromosomes [2]. An LZWGA chromosome has to be decompressed by an LZW decompression algorithm before its fitness can be evaluated. LZWGA can solve very large problem such as one-million-bit OneMax, RoyalRoad and Trap functions.

Estimation of Distribution Algorithm (EDA) is a new approach in evolutionary computation [3][4]. EDA models highly-fit individuals in each generation by assuming a particular distribution. After the model is created, EDA generates new individuals from the model and inserts them to the population. Modeling and generating can avoid the disruption of partial solution resulted from genetic operations such as crossover and mutation. EDAs include Compact Genetic Algorithm (cGA) [5], Mutual Information Maximization for Input Clustering (MIMIC) [6], Bayesian Optimization Algorithm (BOA) [7], etc.

In this paper, we combine LZW compressed encoding with cGA. cGA has an advantage of a compact representation. A chromosome in cGA is a probability vector which represents the whole GA's binary string population. cGA considers all variables independently. Each item in the probability vector represents the probability that the gene will be 0 or 1. However, because the LZW encoded chromosome is an integer array, we have to modified cGA to handle the integer value.

The remainder of this paper is organized as follows. Section II presents technical background. Section III gives details about LZWCGA. Section IV describes the experiments. Section V shows experimental results and discussion. Finally, we conclude our work and suggest future work in Section VI.

II. Technical Background

A. Lempel-Ziv-Welch (LZW) Algorithm

The LZW is a lossless data compression algorithm [8]. The compression algorithm starts with a dictionary in which each entry contains one character. 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 because the algorithm replaces the whole string with its code.

A nice property of LZW is that the dictionary does not have to be packed with a compressed data. LZW decompression does not require a dictionary because the algorithm can reconstruct the dictionary while decompressing data. When using LZW to decompress an English text, the dictionary is initialized with all English characters and symbols. However, when this algorithm is used with GA, the dictionary is initialized with the number 0 and 1 because the output of the decompression algorithm must be a binary string.

A pseudo code for LZW decompression used in LZWGA is shown in Fig. 1.

B. LZWGA

The main difference between LZWGA [2] and GA is that an LZWGA chromosome is in a compressed format. Therefore, the chromosome has to be decompressed before its fitness can be evaluated. The pseudo code of LZWGA is shown in Fig. 2.

The algorithm begins by creating the first generation of compressed chromosomes. Before evaluating the fitness of a chromosome, the compressed chromosome is decompressed using LZW Decompression algorithm. The fitness evaluation is performed on the uncompressed chromosome.

After that, the new population is created to replace the old population. The algorithm repeats the process of

decompression, fitness evaluation, and creating a new population until the termination criterion is met. The algorithm terminates when a solution is found or a maximum generation is reached.

```
Algorithm LZW Decompress
    add entries 0 and 1 to the dictionary
    read one code from input to c
    output str(c)
    p = c
    while input are still left
        read one code from input to c
        if the code c is not in the dictionary
            add str(p)+fc(str(p)) to the dictionary
            output str(p)+fc(str(p))
        else
            add str(p)+fc(str(c)) to the dictionary
            output str(p)
        end if
        p = c
    end while
```

Figure 1. LZW decompress pseudo code

The variable *c* is used to store a code read from input.

The variable *p* is the previous value of *c*.

The function str(*code*) returns a string associated with *code*.

The function fc(*string*) returns the first character in *string*.

```
Algorithm LZWGA
    Z ← create_first_generation()
    repeat
        P ← decompress(Z)
        evaluate(P)
        Z ← create_next_generation(Z)
    until is_terminate()
```

Figure 2. LZWGA pseudo code

The variable Z is the population of compressed chromosome.

The variable P is the population of uncompressed binary chromosomes.

1) Creating the First Generation

Unlike a canonical GA, a chromosome in LZWGA is encoded as integers. The chromosome in LZWGA is in a compressed format. LZWGA chromosome is an array of integer. Each integer is a code for an index of an entry in the dictionary. Chromosomes in the first generation are created as a random integer strings with the constraint that the i^{th} integer of a chromosome must not have value greater than i+1.

For example, an LZWGA chromosome that can be successfully decompressed is (1,2,3). The decompression algorithm will output a binary string 111111. After decompression, a dictionary has the entries (0,0), (1,1), (2,11), and (3,111). Another valid chromosome is (0,1,2). The decompression algorithm will output a binary string 0101.

If the i^{th} integer in an LZWGA chromosome is invalid, the dictionary look up in will be failed after the $(i+1)^{th}$ integer is read. An example of an invalid chromosome is (1,3,3). Before entering the loop, the input "1" (the 0^{th} integer in the chromosome) is read and the algorithm output 1. In the first iteration, the algorithm reads "3" (the 1^{st} integer), adds to dictionary the string 11 at the entry 2, and outputs 11. In the second iteration, the algorithm reads "3" (the 2^{nd} integer), and fail when trying to execute str("3").

In order to generate the value of the i^{th} integer, a random non-negative integer is modulo with i+1.

2) Decompression

Because the chromosome in LZWGA is compressed, it has to be decompressed before its fitness evaluation. A compressed chromosome is decompressed using LZW decompression algorithm. The result is a binary chromosome.

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. After decompression, the decompressed binary string is evaluated. A fitness of a compressed chromosome is equals to the fitness of the decompressed chromosome.

3) Creating the Next Generation

LZWGA creates the population of the next generation by selecting, recombining, and mutating compressed chromosomes. A highly fit chromosome is likely to be selected using any selection method such as tournament or roulette-wheel selection. Compressed chromosomes can be recombined using single-point, two-point, or uniform crossover. Because each of these crossover methods does not change the position of each integer, it automatically creates valid chromosomes that each integer satisfies the constraint. Therefore, the offspring can be decompressed. Mutation changes an integer in uncompressed chromosome to a random value that satisfies the constraint.

C. Compact Genetic Algorithm (cGA)

Harik et al. [5] introduced a compact genetic algorithm (cGA). The performance of cGA is comparable to GA with uniform crossover. cGA is a graphical representation of the probability model of EDAs without independencies. This algorithm uses a single probability vector to represent the whole GA population. Therefore cGA consumes less memory than traditional GA.

III. LZWCGA

LZWCGA combines LZWGA with cGA. cGA uses a probability vector to represent the whole GA population. In contrast, LZWCGA uses a probability matrix instead of a single probability vector because LZWGA's chromosome is an array of integer. Each column of the probability matrix is

a probability that a particular value will occurs for each gene. An example of a probability matrix is shown in Fig. 6.

The main difference between LZWCGA and cGA are initializing and updating the probability matrix process. The LZWCGA algorithm consists of 6 steps shown in Fig. 3.

```
Step 1. Initialize the probability matrix
Step 2. Generate two individuals
Step 3. Decompress both individuals
Step 4. Evaluate both individuals
Step 5. Update the probability matrix
Step 6. Check if the probability matrix has converged or the solution is
found, if not return to Step 2
```

Figure 3. A sequence of LZWCGA process

The first step in LZWCGA is to initialize the probability matrix. The pseudo code is shown in Fig. 4. The sum of the probability in one column of the matrix is 1.

```
Algorithm Initialize Probability Matrix
    for i = 1 to l do
        for j = 1 to i + 1 do
            p[i][j] = 1 / (i + 1)
        end for
    end for
```

Figure 4. Pseudo code for initializing probability matrix

The variable l is length of individual.

Then, we randomly generate two individuals a and b from the probability matrix using the pseudo code in Fig. 5.

```
Algorithm Generate Individuals
    for i = 1 to l do
        r = random()
        interval = 0
        for j = 1 to  i+1 do
            interval += p[i][j]
            if (r ≤  interval)
                lzwChromosome[i] = j
                break
            end if
        end for
    end for
```

Figure 5. Pseudo code for generating individuals

Next, we decompress both individuals using LZW. Then, we evaluate their fitness. The individual with higher fitness score is called the *winner*, whereas the other is called the *loser*. The probability matrix is updated according to values from *winner* and *loser*. The main idea is to increase the probability value at the winner's position by $1/n$ (the variable n is the population size) and decrease value in *loser* positions by $1/n$. The pseudo code for updating the probability matrix is shown in Fig. 8.

By way of illustration, the initial probability matrix is shown in Fig. 6. The probability matrix after updating using values from *winner* and *loser* is shown in Fig. 7.

0.50	0.33	0.25	0.20	0.17
0.50	0.33	0.25	0.20	0.17
	0.33	0.25	0.20	0.17
		0.25	0.20	0.17
			0.20	0.17
				0.17

Figure 6. The initial probability matrix of LZWCGA population when the length of each individual is 5

winner	0	1	2	3	5
loser	1	0	1	0	2
	0.60	0.23	0.25	0.10	0.17
	0.40	0.43	0.15	0.20	0.17
		0.33	0.35	0.20	0.07
			0.25	0.30	0.17
				0.20	0.17
					0.27

Figure 7. The probability matrix after updating (population size n is 10)

```
Algorithm Update Probability Matrix
    for i = 1 to l do
        indexW = winner[i]
        indexL = loser[i]
        if (indexW ≠ indexL)
            if (p[i][indexW] + (1/n) ≥ 1.0)
                p[i][indexW] = 1.0
                for j = 1 to i+1 do
                    if (j ≠ indexW)
                        p[i][j] = 0.0
                    end if
                end for
            else
                if (p[i][indexL] - (1/n) ≤ 0.0)
                    p[i][indexW] += p[i][indexL]
                    p[i][indexL] = 0.0;
                else
                    p[i][indexW] += (1/n)
                    p[i][indexL] -= (1/n)
                end if
            end if
        end if
    end for
```

Figure 8. Pseudo code for updating probability matrix

The last step is to check whether the probability matrix has been converged or the solution is found. If not, the evolution process is repeated starting from step 2.

IV. Experiments

We conducted experiments to compare the performance of LZWCGA and LZWGA on OneMax and Trap problems.

A. OneMax Problem

The OneMax problem [9] (or bit counting) is a widely used problem for testing the performance of various genetic algorithms. Formally, this problem can be described as finding a string $\vec{x}=\{x_1, x_2, ..., x_k\}$, where $x_i \in \{0,1\}$, that maximizes the following equation:

$$F(\vec{x})=\sum_{i=1}^{k} x_i \quad (1)$$

B. Trap Problem

The general *k*-bit trap functions [9] are defined as:

$$F(\vec{x})=\begin{cases} f_{high} & ;if\ u=k \\ f_{low}-(u\times f_{low})/(k-1) & ;otherwise \end{cases} \quad (2)$$

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 denoted by $F_{m\times k}$ are defined as:

$$F_{m\times k}(K_1 ... K_m)=\sum_{i=1}^{m} F_k(K_i),\ K_i \in \{0,1\}^k \quad (3)$$

The *m* and *k* are varied to produce a number of test functions. The Trap functions fool the gradient-based optimizers to favor zeros, but the optimal solution is composed of all ones. The Trap function is a fundamental unit for designing test functions that resist hill-climbing algorithms.

C. Parameters

The parameters for both algorithms are shown in Table I and II. The Table I shows parameters for OneMax problem. Table II shows parameters for Trap problem. The size of compressed chromosome is set to 4, 5 and 6 times on OneMax and 4 times smaller than the size of a decompressed chromosome on Trap problem. We call the ratio the chromosome compression ratio. We compare the performance of LZWCGA and LZWGA for various compression ratios. LZWGA uses tournament selection (tournament size = 4). It uses uniform crossover and does not use mutation.

All experimental results are the average performance obtained from 30 runs.

TABLE I. Parameters of LZWGA and LZWCGA for OneMax problem

Parameter	Value
Population size	128, 512, 1024
Problem size (bits)	1000, 10000, 100000
Chromosome compression ratio	1/4, 1/5, 1/6 of problem size
Max generation (for LZWGA)	500
Max round (for LZWCGA)	500 x population size
Crossover rate (for LZWGA)	1
Mutation rate (for LZWGA)	0

TABLE II. Parameters of LZWGA and LZWCGA for Trap problem

Parameter	Value
Population size	128, 512, 1024
Trap size	5
Total trap	100, 1000, 10000
Problem size (bits)	Trap size x Total trap
Chromosome compression ratio	1/4 of problem size
Max generation (for LZWGA)	500
Max round (for LZWCGA)	500 x population size
Crossover rate (for LZWGA)	1
Mutation rate (for LZWGA)	0

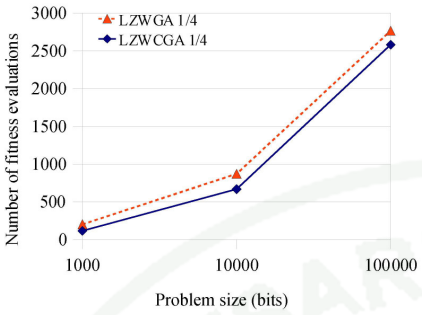
V. Results and Discussion

The experimental results show that LZWCGA outperforms LZWGA on both OneMax and Trap problems (see Fig. 9 and Fig. 10). We found that the bigger problem size needs more fitness evaluations. Moreover, higher compression ratio requires more fitness evaluations.

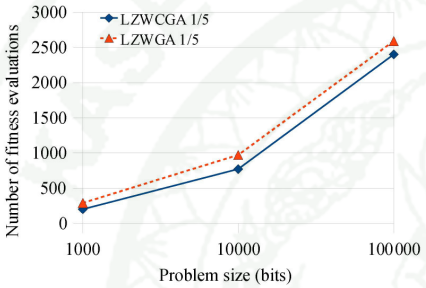
LZWGA's memory requirement depends on chromosome length and population size while LZWCGA depends only on chromosome length. For equal chromosome length, LZWGA will use approximately the same amount of memory as LZWCGA when the population size is equal to the length of the chromosome. For example, when compressed chromosome length is 1000, LZWGA with 1003 individuals uses the same amount of memory as LZWCGA. (Note that each item in an LZWGA individual is 16-bit unsigned integer and each item in an LZWCGA matrix is 32-bit float.)

A visual representation for an LZWCGA probability matrix is shown in Fig. 11. The darker area indicates higher probability. The initial probability matrix is shown in the first sub figure. Each column in the first sub figure has the same shade of gray because the initial probability that each value in each gene will occur is equal. However, during the evolution, the probability is changed. The second, third, fourth sub figure is a probability matrix at 10000, 20000, 30000 fitness evaluations and so on. In the last sub figure, the probability matrix converges. Normally, LZWGA finds a solution before the probability matrix converges. In one

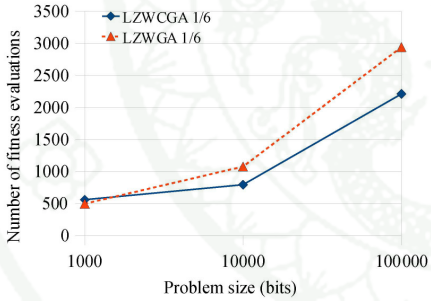
experiment, LZWCGA found a solution around 35000 fitness evaluations while the probability matrix converges around 45000 fitness evaluations.



(a) Population size is 128. The compression ratio is 1/4.



(b) Population size is 512. The compression ratio is 1/5.



(c) Population size is 1024. The compression ratio is 1/6.

Figure 9. The number of fitness evaluations of LZWCGA and LZWGA when solving various sizes of OneMax problem.

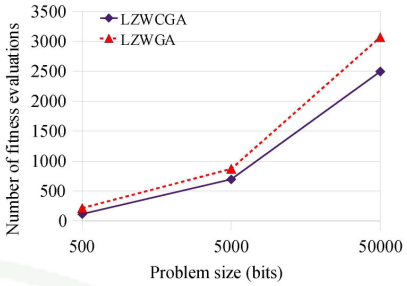


Figure 10. The number of fitness evaluations when using LZWCGA and LZWGA to solve Trap problem. The compression ratio is 1/4.

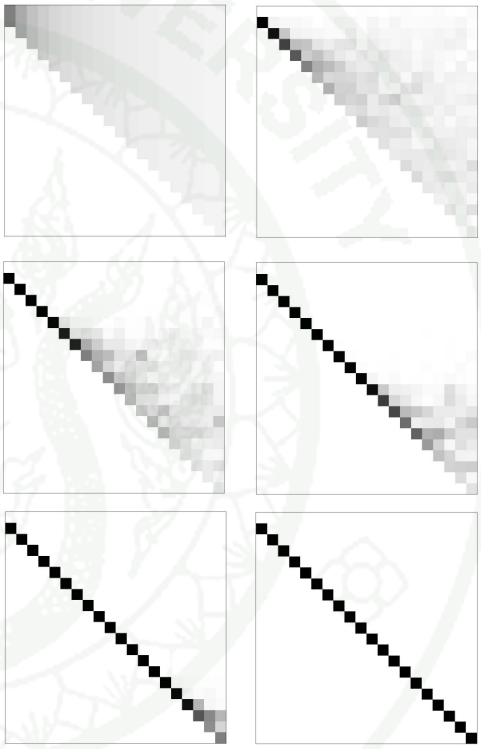


Figure 11. A visual representation for a probability matrix at 0, 10000, 20000, 30000, 40000, and 50000 fitness evaluations

VI. Conclusion and Future Work

We proposed the algorithm LZWCGA which combines the compress encoding and probabilistic model building. The main feature of LZWCGA is an ability to reduce the search space which make the algorithm find the solution effectively. We found the LZWCGA's performance is comparable to LZWGA on OneMax and Trap problem. This result is promising because, in the future, we will improve the update process for probability matrix and apply LZW with more advanced EDAs such as MIMIC (Mutual Information Maximization for Input Clustering), which can solve combinatorial optimization problems with bivariate dependencies.

References

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

[2] Naris Kunasol, Worasait Suwannik, and Prabhas Chongstitvatana, "Solving One-Million-Bit Problems Using LZWGA," Proc. International Symposium on Communications and Information Technologies (ISCIT), Oct. 2006, pp. 32-36.

[3] Pedro Larrañaga and Jose A. Lozano, Estimation of Distribution Algorithms A New Tool for Evolutionary Computation, Ed., Kluwer academic publishers, Boston, 2002.

[4] Topon K. Paul and Hitoshi Iba, "Linear and Combinatorial Optimizations by Estimation of Distribution Algorithms," Proc. 9th MPS Symposium on Evolutionary Computation, IPSJ, 2002.

[5] Georges R. Harik, Fernando G. Lobo, and David E. Goldberg, "The Compact Genetic Algorithm," IEEE Transaction on Evolutionary Computation, vol. 3, no. 4, Nov. 1999, pp. 287-297.

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

[7] Martin Pelikan, David E. Goldberg, and Erick Cantù-Paz, "BOA: The Bayesian Optimization Algorithm," Proc. The Genetic and Evolutionary Computation Converence (GECCO), 1999, pp. 525-532.

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

[9] Melanie Mitchell, An Introduction to Genetic Algorithms, MIT Press, 1998.

Conditional Probability Mutation in LZWGA

Orawan Watchanupaporn and Worasait Suwannik
Department of Computer Science
Kasetsart University
Bangkok, Thailand
66-2562-5555

orawan.liu@gmail.com, worasait.suwannik@gmail.com

ABSTRACT

LZWGA is an algorithm that combines LZW compression algorithm with genetic algorithm (GA). An LZWGA chromosome can be decompressed to a GA binary-string chromosome. In this paper, we propose a new mutation operator for LZWGA called conditional probability mutation (CPM). In contrast to original LZWGA mutation which randomly changes some values in an individual, CPM takes advantage of the relationship between gene positions. We compare the performance of LZWGA with original mutation and LZWGA with CPM on non random and random version of standard benchmark problems. Furthermore, we vary mutation rate to see its effect in the performance. The experimental results show that our proposed mutation outperforms original LZWGA mutation in non random problems.

Categories and Subject Descriptors

I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, Search - *Heuristic methods.*

General Terms

Algorithms, Experimentation.

Keywords

Estimation of Distribution Algorithm, Evolutionary Algorithms, Lempel-Ziv-Welch Algorithm, Compression, Mutation method.

1. INTRODUCTION

Genetic Algorithm (GA) solves problems by simulating natural evolution [1]. To solve a problem using GA, a candidate solution is normally encoded into a binary string. The length of this string represents the size of the problem. As the length of the binary string increases, the size of the search space also increases at an exponential rate. For example, the size of search space for a 10-bit chromosome is 2^{10}. While the size of search space for a 100-bit chromosome is 2^{100}.

To reduce the search space, one approach is to utilize a compressed encoding chromosome. Kunasol et. al. proposed LZWGA, which is a GA that uses LZW compressed chromosomes [2]. LZWGA's chromosome is an array of integer. An LZWGA chromosome has to be decompressed by an LZW decompression algorithm before its fitness can be evaluated. LZWGA efficiently solve very large problem such as one-million-bit OneMax, RoyalRoad and Trap functions.

LZWGA uses crossover and mutation to produce the offspring. Crossovers in LZWGA are similar to those in GA. However, a mutation in LZWGA is different from GA's binary mutation. Instead of flipping bit between one and zero, an LZWGA mutation randomly change a value in a chromosome to an integer between 0 and i+2, where i is a mutated position. If the integer is not in the range, a chromosome cannot be decompressed. A single mutation can change several positions in a chromosome. However, the change occurs randomly and independently. If there are dependencies between positions in the chromosome, a mutation unlikely improves the fitness of a chromosome. This is because the original mutation does not know which positions should be changed at the same time.

In this paper, we propose a new mutation operator. Instead of randomly changes a chromosome, the new operator changed a chromosome in a principled way. We assume a particular relationship between positions of highly fit chromosomes. The relationship is represented by conditional probability. The new mutation operator changes a chromosome according to the conditional probability.

The remainder of this paper is organized as follows. Section 2 review some related work. Section 3 presents motivation and gives details about the proposed mutation. Section 4 describes the experiments. Section 5 shows results and discussion. Finally, we conclude our work and suggest future work in Section 6.

2. RELATED WORKS

This section presents related works which are on LZW compression algorithm and LZWGA.

2.1 Lempel-Ziv-Welch (LZW) Algorithm

LZW is a lossless data compression algorithm [3]. The algorithm uses a dictionary which is a map of a code word and a string associated with the code. The compression algorithm starts with a dictionary in which each entry contains one character. 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 because the algorithm replaces the whole string with its code.

A nice property of LZW is that the dictionary does not have to be packed with a compressed data. LZW decompression does not require a dictionary because the algorithm can reconstruct the dictionary while decompressing data. When using LZW to decompress an English text, the dictionary is initialized with all English characters and symbols. However, when this algorithm is used with GA, the dictionary is initialized with the number 0 and 1 because the output of the decompression algorithm must be a binary string. A pseudocode for LZW decompression used in LZWGA is shown in Figure 1.

2.2 LZWGA

The main difference between LZWGA and GA is that an LZWGA chromosome is in a compressed format. Therefore, the chromosome has to be decompressed before its fitness can be evaluated. The pseudocode of LZWGA is shown in Figure 2.

The algorithm begins by creating the first generation of compressed chromosomes. Before evaluating the fitness of a chromosome, the compressed chromosome is decompressed using LZW Decompression algorithm. The fitness evaluation is performed on the uncompressed chromosome.

After that, the new population is created to replace the old population. The algorithm repeats the process of decompression, fitness evaluation, and creating a new population until the termination criterion is met. The algorithm terminates when a solution is found or a maximum generation is reached.

```
Algorithm LZW Decompress
    add entries 0 and 1 to the dictionary
    read one code from input to c
    output str(c)
    p ← c
    while input are still left
        read one code from input to c
        if the code c is not in the dictionary
            add str(p)+fc(str(p)) to the dictionary
            output str(p)+fc(str(p))
        else
            add str(p)+fc(str(c)) to the dictionary
            output str(p)
        end if
        p ← c
    end while
```

Figure 1. Pseudocode for LZW decompress

The variable *c* is used to store a code read from input.
The variable *p* is the previous value of *c*.
The function str(*code*) returns a string associated with *code*.
The function fc(*string*) returns the first character in *string*.

```
Algorithm LZWGA
    Z ← create_first_generation()
    repeat
        P ← decompress(Z)
        evaluate(P)
        Z ← create_next_generation(Z)
    until is_terminate()
```

Figure 2. Pseudocode for LZWGA

The variable *Z* is the population of compressed chromosome.
The variable *P* is the population of uncompressed binary chromosomes.

2.2.1 Creating the First Generation

Unlike a canonical GA, a chromosome in LZWGA is encoded as integers. The chromosome in LZWGA is in a compressed format. LZWGA chromosome is an array of integer. Each integer is a code for an index of an entry in the dictionary. Chromosomes in the first generation are created as a random integer strings with the constraint that the i^{th} integer of a chromosome must not have value greater than i+l. Otherwise a chromosome cannot be decompressed.

For example, an LZWGA chromosome that can be successfully decompressed is (1,2,3). The decompression algorithm will output a binary string 111111. After decompression, a dictionary has the entries (0,0), (1,1), (2,11), and (3,111).

2.2.2 Decompression

Because the chromosome in LZWGA is compressed, it has to be decompressed before its fitness evaluation. A compressed chromosome is decompressed using LZW decompression algorithm. The result is a binary chromosome. The length of the output 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. After decompression, the decompressed binary string is evaluated. A fitness of a compressed chromosome is equals to the fitness of the decompressed chromosome.

2.2.3 Creating the Next Generation

LZWGA creates the population of the next generation by selecting, recombining, and mutating compressed chromosomes. A highly fit chromosome is likely to be selected using any selection method such as tournament or roulette-wheel selection. Compressed chromosomes can be recombined using single-point, two-point, or uniform crossover. Because each of these crossover methods does not change the position of each integer, it automatically creates valid chromosomes that each integer satisfies the constraint. Therefore, the offspring can be decompressed. Mutation changes an integer in uncompressed chromosome to a random value that satisfies the constraint.

3. LZWGA with CPM

This section presents motivation and gives implementation details of LZWGA with CPM.

3.1 Motivation

The proposed mutation is inspired by MIMIC (Mutual-Information-Maximizing Input Clustering) [4]. MIMIC is one of estimation of distribution algorithms [5]. It has two major assumptions. First, it assumes several dependencies between two positions in a binary chromosome. Second, it assumes that each position is related with another position (except one position with the lowest entropy). Two positions with the lowest conditional entropy have the strongest relation. It finds two related positions using greedy method. Finding related positions has a time complexity $O(n^2)$. After a distribution is chosen, generating new individual also has a time complexity $O(n^2)$.

Our mutation also has the same two major assumptions as MIMIC. However, we further assume that two consecutive positions in an LZW chromosome are related. The additional assumption is that the value of the 1^{st} position depends on the value of 0^{th} position (or notationally written as 0←1) and 1←2, 2←3, 3←4, and so on.

The reason behinds the additional assumption is the characteristic of LZW compression algorithm. In MIMIC, the position with the lowest entropy is assumed to be independent of other positions. The 0^{th} position in LZW chromosome is likely to have the lowest

entropy. This is because the 0^{th} position can have only two possible values while the other position can have more possible values. Therefore, we pick the 0^{th} position as the starting point. Moreover, because a dictionary entry always contains a code in a prior entry, a value in a compressed chromosome depends on a value in the previous position of the chromosome. Therefore, we assumed that the i^{th} position depends on the $(i\text{-}1)^{th}$ position. Implementation details as explained in Section 3.2.

3.2 Implementation Details

The main steps of LZWGA with CPM are the same as in the LZWGA. A difference between LZWGA and LZWGA with CPM is mutation method which uses a conditional probability model of selected individuals instead of randomly change each value in an individual. The algorithm can be illustrated by a simplified pseudocode in Figure 3.

```
Algorithm LZWGA with CPM
    Z ← create_first_generation()
    repeat
        P ← decompress(Z)
        evaluate(P)
        S ← selection(Z)
        M ← model(S)
        Z ← create_next_generation(M, S)
    until is_terminate()
```

Figure 3. Pseudocode for LZWGA with CPM

The conditional probability model contains the frequencies of first position and the conditional frequencies of the remaining positions from *count* which each entry in count is initialized with 1. The variable *count* is initialized with 1. After that used to store the frequencies. An entry *count*[*a*][*b*][*c*] is frequencies that the $(a\text{-}1)^{th}$ position has value *b* and the ath position has value *c*. The exceptions are *count*[0][0][0] and *count*[0][0][1] which are frequencies that the other positions have value 0 and 1 respectively. To create the next generations, the algorithm first selects the best individual from the previous generation. Then, the algorithm selects *n*-1 individuals from previous generation using the tournament selection (*n* is population size). After *n* individual are selected, the model is updated using the following pseudocode. The model can be updated very fast.

```
for i = 0 to n-1 do
    count[0][0][selectedIndividuals[i][0]]++
end for
for index = 1 to l-1 do
    for i = 0 to n-1 do
        row = selectedIndividuals[i][index-1];
        col = selectedIndividuals[i][index];
        count[index][row][col]++
    end for
end for
```

The operation of mutation changes the values at two related positions of individuals by using a value in the previous position and *count* of an individual. The mutation starts at random the position and generates values to the last position based on the conditional probability stored in the variable *count* values to the final position.

The operation of mutation can be illustrated as follows:

```
if (random < CPMmutationRate)
    position = randomPosition()
    individual[position] = randomLzwGene(position)
    for i = 1 to l-1 do
        individual[i] = proportionalChoose(count[i]
                        [individual[i-1]])
    end for
end if
```

4. EXPERIMENTS

This section described test problems and experimental parameters.

4.1 Test Problems

We conducted experiments to compare the performance of original LZWGA mutation and conditional probability mutation (CPM) on OneMax, RandomMax, Trap and RandomTrap problems.

4.1.1 OneMax

The OneMax Problem [6] (or bit counting) is a widely used problem for testing the performance of various genetic algorithms. Formally, this problem can be described as finding a string $\vec{x}=\{x_1, x_2, \ldots, x_k\}$, with $x_i \in \{0,1\}$, that maximizes the following equation:

$$F(\vec{x})=\sum_{i=1}^{k} x_i \tag{1}$$

4.1.2 RandomMax

The optimum of OneMax is in the string of all ones. RandomMax problem is based on OneMax problem but an optimal solution is a random binary string. The objective is to maximize the number of bits that match that random string. An example of this is shown in Figure 4. The fitness of an example individual is 3 because there are 3 bits that match the solution.

Solution	1	0	1	1	0
Individual	1	1	0	1	0

Figure 4. Example of RandomMax problem

4.1.3 Trap

An individual is divided into partitions of *k* bits each. The length of an individual is a multiple of *k*. The general *k*-bit trap functions [7] are defined as:

$$F(\vec{x})=\begin{cases} f_{high} & ;if\ u=k \\ f_{low}-(u\times f_{low})/(k-1) & ;otherwise \end{cases} \tag{2}$$

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 denoted by $F_{m\times k}$ are defined as:

$$F_{m\times k}(K_1 \ldots K_m)=\sum_{i=1}^{m} F_k(K_i),\ K_i\in\{0,1\}^k \quad (3)$$

The m and k are varied to produce a number of test functions. The Trap functions fool the gradient-based optimizers to favor zeros, but the optimal solution is composed of all ones. The Trap function is a fundamental unit for designing test functions that resist hill-climbing algorithms.

4.1.4 RandomTrap

The RandomTrap problem of order k is defined using the same partitions as trap of order k, but each part is compared against a random binary string defined as a solution. See Figure 5 to visualize RandomTrap for 10-bit strings. Number of fitness values for this example is 8.

Scores	4	3	2	1	0	5				
Random block	1	0	1	1	0					
Individual	0	1	1	0	1	1	0	1	1	0

Figure 5. Example of RandomTrap problem

4.2 Experiments Description

In each experiment, 30 independent runs were performed. The population size is 2,000. Selection method is tournament selection with size equals to 4. Mutation rate of LZWGA is 1% and LZWGA with CPM is 5%. The problem size is the maximum length that a chromosome can be decompressed, which is

$$\frac{chromosome\ length\times(chromosome\ length+1)}{2} \quad (4)$$

For the other experiments were done on parameters (see in Table 1 and Table 2). We measure the performance of LZWGA with CPM using the average number of generations in the case that the solution is found (i.e., OneMax and Trap) and using the average number of fitness values in the case that the solution is not found (i.e., RandomMax and RandomTrap).

The effect of different problem sizes on the new mutation method is investigated by fixing the population size and varying the problem size (or individual length).

Table 1. Parameters of LZWGA for OneMax and Trap problems

Parameter	Value
Problem size (bits)	55, 465, 1275
Chromosome length	10, 30, 50
Maximum generation	1000

Table 2. Parameters of LZWGA for RandomMax and RandomTrap problems

Parameter	Value
Problem size (bits)	55, 1275, 5050
Chromosome length	10, 50, 100
Maximum generation	2000

5. RESULTS AND DISCUSSION

Although we have tried various LZWGA with CPM mutation rate which are 0.01, 0.05, 0.10, …, 0.95. However, for brevity, we only show results for mutation rate is 0.01 which gives good experimental results.

Plots of the average number of generations is in Figure 6 and Figure 8. The solid line is for the proposed mutation, and the dashed line is original mutation. LZWGA with both mutations can find a solution for OneMax and Trap problems. Therefore, a mutation method that requires less generations is preferable. Figure 6 and Figure 8 confirm high efficiency of LZWGA with CPM. These graphs show the scalability of LZWGA with CPM with sizes of the individuals length for OneMax and Trap problem. We found that LZWGA with CPM used less number of generations than LZWGA.

LZWGA with both mutations cannot find a solution for both problems. Therefore, a mutation method that give higher fitness values is preferable. We plot the average number of fitness values for RandomMax and RandomTrap problems in Figure 7 and Figure 9, which show that both the LZWGA and LZWGA with CPM give similar results.

In summary, the results show that the proposed mutation can find solution faster than the original mutation on standard benchmark problems. However, when investigating random problems, both mutations give similar results. This suggests that our assumption that the i^{th} position depends on the $(i\text{-}1)^{th}$ position does not enhance the performance of the algorithm on random problems.

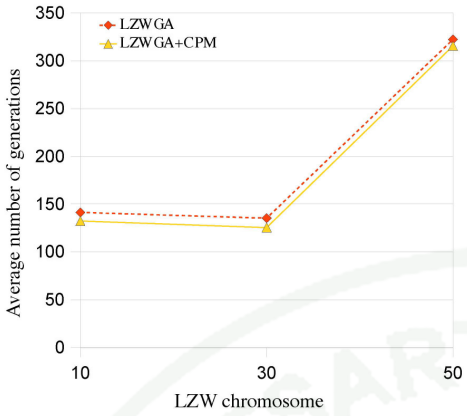


Figure 6. Average number of generations as LZW chromosome length is varied on OneMax problem

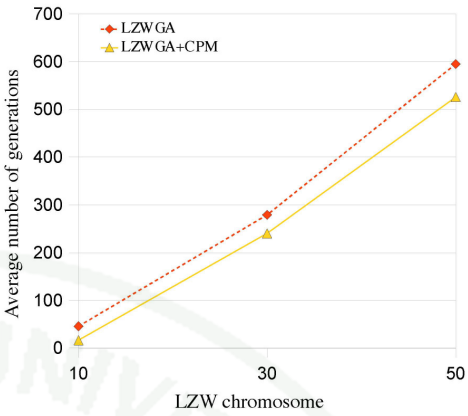


Figure 8. Average number of generations as LZW chromosome length is varied on Trap problem

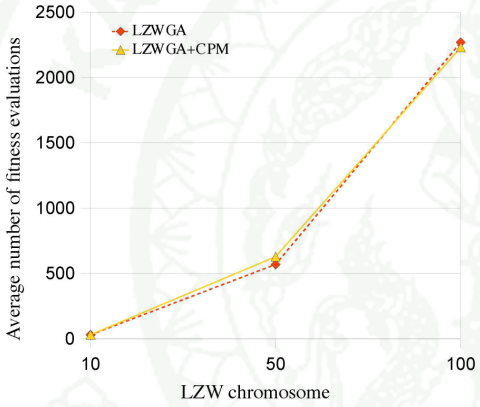


Figure 7. Average best fitness after 2,000 generations as LZW chromosome is varied on RandomMax problem

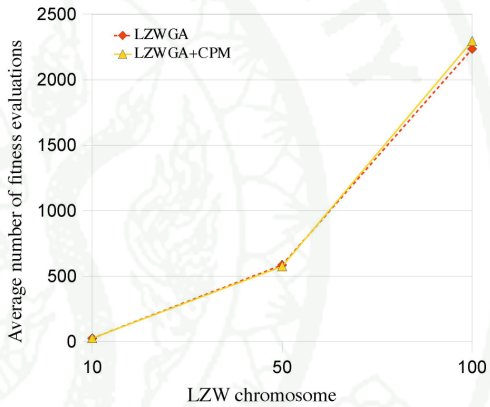


Figure 9. Average best fitness after 2,000 generations as LZW chromosome is varied on RandomTrap problem

6. CONCLUSIONS AND FUTURE WORK

We proposed CPM, a new LZWGA mutation method which uses conditional probability learned from all generations of selected individuals. We tested the efficiency of our mutation method with widely used in genetic algorithms and random solution. From the experiments, CPM outperforms the original mutation in non random version of the well-known benchmark problems and gives similar result for random version of those benchmark problems. Our assumption about relationship between two consecutive positions in a LZW chromosome facilitates model construction. The model can be constructed very quickly. In the future, we will relax the assumption by assuming a relationship between any two positions (similar to MIMIC) or any two positions which the right position depends on the value of the left position.

7. REFERENCES

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

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

[3] Welch, T. A. 1984. A Technique for High-Performance Data Compression. *Trans. IEEE Computer.* 17, 6 (Jun. 1984). 8-19.

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

[5] Larrañaga, P. and Lozano, J. A. Eds. 2002. Estimation of Distribution Algorithms A New Tool for Evolutionary Computation. *Kluwer academic publishers. Boston.*

[6] Schaffer, J. D. and Eshelman, L. J. 1991. On crossover as an evolutionary viable strategy. In R.K. Belew and L.B. Booker, editors. *Proceedings of the 4th International Conference on Genetic Algorithms*. Morgan Kaufmann. 61-68.

[7] Mitchell, M. 1998. An Introduction to Genetic Algorithms. *MIT Press*.

LZW Mutual-Information-Maximizing Input Clustering Algorithm

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

***Abstract*— This paper proposes a new evolutionary algorithm called LZWMIMIC. The proposed algorithm combines the LZW compressed chromosome encoding and Mutual-Information-Maximizing Input Clustering (MIMIC) algorithm. The advantage of LZW encoding is that it reduces the search space thus speeds up the evolutionary search. The advantage of MIMIC is that it can solve complex problem by finding a relationship between gene positions. The performance of the original MIMIC and LZWMIMIC are compared on standard benchmark problems. Further, compressed chromosome length and problem size are varied to see their effect in the performance. The experimental results show that our proposed algorithm outperforms the original MIMIC.**

Keywords; LZW, MIMIC, EDA

I. INTRODUCTION

Evolutionary Algorithms (EAs) are a group of algorithms inspired by natural evolution. EAs include Genetic Algorithm (GA) [1, 2], Genetic Programming (GP) [3], Evolutionary Strategies (ES) [4], etc. The algorithms can solve various kinds of problems. Mostly, they are applied to solve optimization problems especially the one which the structure is unknown. An evolutionary search procedure consists of fitness evaluation, selection, recombination and mutation. Evolutionary search normally requires a long computation time because each operation is performed on a population of chromosomes. There are at least two approaches that can speed up evolutionary search.

The first approach is to use Estimation of Distribution Algorithm (EDA). EDA is a new kind of evolutionary algorithms [5, 6]. It extends GA in a principle manner. As a result, EDAs can scale up better than its predecessor. EDAs use various probabilistic and statistic approaches to find the relationship between gene positions. EDAs can be categorized by the relationship between variables: independent variable, bivariate dependencies, and multiple dependencies [7]. EDAs include PBIL [8], cGA [9], MIMIC [10], BOA [11], etc.

The second approach that can speed up evolutionary search is to reduce search space. The size of search space can be reduced by adding a heuristic into one of evolution operations. For example, in [12], the specific type of crossover that preserves some constraints can beneficially reduce the search space. The result shows that the proposed crossover can find better solution for a flow shop scheduling problem. Another approach to reduce the search space is to utilize a compressed chromosome encoding. For example, a GA with LZW compressed chromosomes can solve very large problem such as one-million-bit OneMax, RoyalRoad, and Trap functions [13]. A million-bit problem has a very large search space (i.e., $2^{1000000}$ or 9.90×10^{301029}point).

This paper proposes LZWMIMIC, which combines LZW compressed encoding with MIMIC. We use a compressed encoding as a means to reduce search space because it does not require any domain knowledge. As for the choice of EDAs, we think that univariate (dependent variables) EDAs might not be sufficient and multivariate EDAs are very time consuming. MIMIC, which is a bivariate EDA, is chosen because the algorithm is very fast. Its time complexity for each iteration is $O(n^2)$. MIMIC approximates a real distribution by finding bivariate dependencies between variables. MIMIC can find highly fit solution for optimization problems very fast.

The remainder of this paper is organized as follows. Section II presents some related works and technical background. Section III describes LZWMIMIC. Section IV explains the experiments. Section V shows experimental results. Section VI discusses experimental results. Finally, we conclude our work and discuss a future work in Section VII.

II. TECHNICAL BACKGROUND

A. *Lempel-Ziv-Welch (LZW) Algorithm*

LZW is a lossless data compression algorithm [14, 15]. This compression algorithm is well known, simple, and widely used in a variety of applications that need the compression mechanism such as text compression [16] and lossless image compression [17].

LZW is a dictionary based compression algorithm. Initially, each entry in the dictionary contains only one character. For example, when using LZW to decompress an English text, the dictionary is initialized with all English characters and symbols. However, when this algorithm is used to decompress a binary chromosome in GA, the dictionary is

initialized with the number 0 and 1. A nice property of LZW is that the dictionary does not have to be packed with a compressed data. The LZW decompression algorithm can reconstruct the dictionary while decompressing.

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 entry. Data is compressed because the algorithm replaces the whole string with its code.

The output of the compression algorithm is an array of integers that refer to the code in the dictionary. Thus, the array of integers will be an input to the decompression algorithm. LZWGA uses only LZW decompression algorithm [13]. The LZWGA chromosome is an array of integer which will be decompressed to a binary GA chromosome before the fitness evaluation.

B. *Mutual-Information-Maximizing Input Clustering (MIMIC) Algorithm*

Each iteration of an EDA consists of selecting highly fit chromosomes, modeling them, and using the model to generate the next generation of chromosomes. The algorithm tries to model the real joint probability distribution p(X) of selected chromosome, which is:

$$p(X) = p(X_1|X_2...X_n)p(X_2|X_3\ldots X_n)\ldots p(X_{n-1}|X_n)p(X_n)$$

The performance of EDAs depends on the quality of the estimation of the real probability distribution. MIMIC estimates the real joint distribution using a pairwise conditional distribution. The estimated pairwise distribution is in the following form:

$$p'(X) = p(X_{i_1} \mid X_{i_2})\, p(X_{i_2} \mid X_{i_3})...p(X_{i_{n-1}} \mid X_{i_n})\, p(X_{i_n})$$

where i_n to i_1 is a permutation of positions in a chromosome

MIMIC uses a greedy method to find the value of i_n to i_1 that makes $p'(X)$ closest match the real distribution $p(X)$. i_n is the position of a chromosome with the lowest entropy. i_{n-1} is the position of a chromosome with the lowest conditional entropy $h(X_{i_{n-1}} \mid X_{i_n})$. i_{n-2} is the position of a chromosome with the lowest conditional entropy $h(X_{i_{n-2}} \mid X_{i_{n-1}})$. And so on. These values together with the selected individuals are the model of selected chromosomes.

After the model is created, the next generation is created. First, the i_n^{th} bit in the chromosome is generated based on the probability $p(X_{i_n})$. Then, i_{n-1}^{th} bit in the chromosome is generated based on the previous value and the conditional probability $p(X_{i_{n-1}} \mid X_{i_n})$. The remaining bits are generated in the same manner.

Both modeling and generating can be done in $O(n^2)$.

C. *Gray Code*

MIMIC operates on binary strings. However, an LZW can only decompress an array of integer. Therefore, a straightforward way to use LZW with MIMIC is to define a MIMIC chromosome using the binary numbers and convert it to an LZW's integer array. The conversion uses Gray code decoding.

Gray code is a numerical code in which consecutive integers are represented by binary numbers differing in only one bit.

III. LZWMIMIC

LZWMIMIC combines LZW compression algorithm with MIMIC EDA. The pseudo code of LZWMIMIC is shown in Figure 1. The algorithm is similar to MIMIC except there are 2 steps added to the beginning of the loop. The additional step is required because LZWMIMIC chromosome is in a compressed form. The chromosome has to be decompressed before its fitness can be evaluated. As show in Figure 2, LZWMIMIC's chromosome is a binary string encoded with Gray code. It will be decoded into an array of integer. Next, the array is decompressed into a binary string using an LZW decompression algorithm. After that, the binary string (from the last step) can be evaluated.

Adding decompression does not necessary slow down the algorithm. In some case, adding decompression causes the algorithm to run faster. For example, when solving 1-million bit problems, one iteration of LZWGA is faster than GA [13]. This is because recombination and mutation takes much longer in GA due to a much larger chromosome.

The fitness evaluation is performed on the uncompressed chromosome. After selection, a highly fit population of compressed chromosome is modeled. Then, the new population is generated to replace the old population. The algorithm repeats the process of decoding, decompression, fitness evaluation, selecting, modeling, and generating a new population until the termination criterion is met. The algorithm terminates when a solution is found or a maximum generation is reached.

```
Algorithm LZWMIMIC
    G ← create_first_generation()
    repeat
        L ← decode(G)
        B ← decompress(L)
        evaluate(B)
        S ← select(B)
        M ← model(S)
        G ← generate(M, S)
    until is_terminate()

G is the population of Gray code chromosome.
L is the population of LZW chromosome.
B is the population of uncompressed binary chromosomes.
S is selected individuals.
M is a model.
```

Figure 1. LZWMIMIC pseudo code.

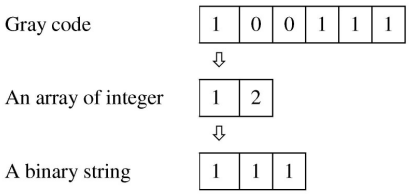


Figure 2. Decoding and decompressing a chromosome. In general, the length of a compressed chromosome is shorter than that of a decompressed chromosome.

IV. EXPERIMENTS

We conduct experiments to compare the performance of LZWMIMIC and MIMIC. The performance of LZWMIMIC is measured based on the number of fitness evaluations. We test algorithms against two optimization problems. Both problems which were originally proposed to show the efficiency of MIMIC. In the original MIMIC paper, the algorithm was tested with small problems which are 30 to 80-bit Four-Peak and 20 to 60-bit Six-Peak problems. But this paper, we test with large problems which are 100, 400, and 800 bits. We set the max decompressed length of binary chromosome equal to the problem size. The size of compressed chromosome (Gray code) is one half, equal to, and double of that of decompressed chromosome for these problems.

Two benchmark problems in this study are Four-Peak and Six-Peak problems.

A. Four-Peak problem

The Four-Peak problem [10, 18] has two global maxima and two suboptimal local optima. The problem is defined as follows.

$$f(X, T)=\max[t(0, X), h(1, X)] + r(X, T)$$

where

$t(b, X)$ = number of trailing b's in X

$h(b, X)$ = number of leading b's in X

$$r(X, T) = \begin{cases} N & \text{if } t(0, X)>T \text{ and } h(1, X) > T \\ 0 & \text{otherwise} \end{cases}$$

X is an input vector.
N is the length of a chromosome.
T is a threshold. This paper sets the value of T to 10% of N.
b is a value of bit.

B. Six-Peak problem

The Six-Peak problem [10] is harder than the Four-Peak problem. It has 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.

$$r(X, T) = \begin{cases} N & \text{; if } (t(0, X)>T \text{ and } h(1, X) > T) \text{ or} \\ & \quad (t(1, X)>T \text{ and } h(0, X) > T) \\ 0 & \text{; otherwise} \end{cases}$$

C. Parameters

Table I shows the experimental parameters of LZWMIMIC and MIMIC for Four-Peak and Six-Peak problems. All experiments are averaged over 30 runs.

TABLE I. EXPERIMENTAL PARAMETERS

Parameter	Value
Population size	1000
Problem size (bits)	100, 400, 800
Chromosome compression ratio (for LZWMIMIC)	1/2, x1, x2 of problem size
Max generation	200

V. RESULTS

We compare the LZWMIMIC with the original MIMIC. The performance criterion is the number of fitness evaluations. Both algorithms were tested against 100, 400, and 800-bit Four-Peak and Six-Peak problems. Plots of the average number of function evaluations are shown in Figure 3 and Figure 4. The performance of LZWMIMIC and MIMIC are shown using solid and dash lines respectively.

The experimental results show that, when a problem is small, MIMIC can quickly converge to the solution and has average best fitness value than LZWMIMIC. However, we found that when the problem size is large (400 and 800 bits) LZWMIMIC performs better than MIMIC. Moreover, Figure 3 and Figure 4 show the scalability of LZWMIMIC. As the problem size increase, LZWMIMIC can find better solution than MIMIC.

Finally, this experiment uses different length of the compressed chromosome. The length of compressed chromosome is half of, equal to, and double the size of uncompressed chromosome. LZWMIMIC with three different lengths can find a solution for Four-Peak and Six-Peak problems. However, the best compressed chromosome length for Four-Peak problem is a half and for Six-Peak problem is equal to the problem size.

VI. DISCUSSION

In this paper, LZWMIMIC is run on two problems: Four-Peak and Six-Peak. Both of the problems are multimodal. The original MIMIC performs well on small problems. However, when the problem becomes larger, LZWMIMIC performs better. The reason is because LZWMIMIC reduces the search space thus speeding up the evolutionary search.

We found that the compressed chromosome length affects the performance of LZWMIMIC. From the problems that we experimented, the best length of binary chromosome depends on the problems. For example, LZWMIMIC used compressed chromosome length is half and equal to MIMIC that it can solve the Four-Peak and Six-Peak problem respectively very well.

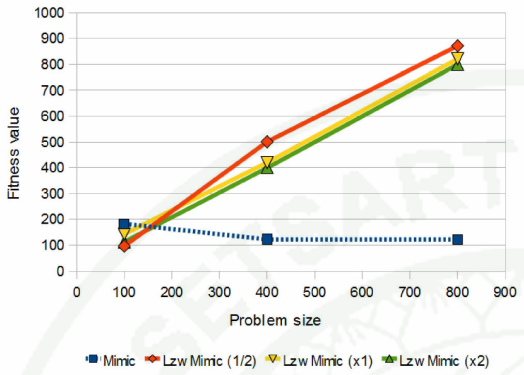


Figure 3. Average best fitness value for the Four-Peak problem

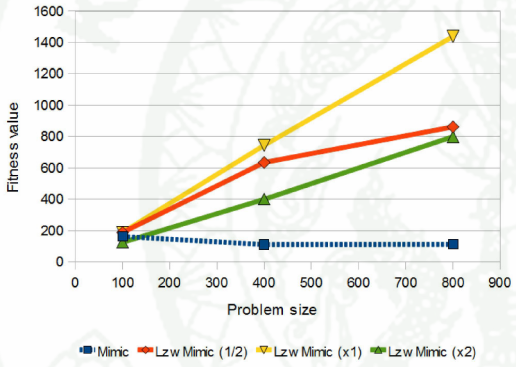


Figure 4. Average best fitness value for the Six-Peak problem

VII. Conclusions and Future Work

LZWMIMIC uses the compressed encoding and population modeling. The main feature of LZWMIMIC is an ability to reduce the search space so that it can effectively find a solution. LZWMIMIC performs well when the problem is large. We investigated two parameters related to the compressed encoding in LZWMIMIC and found that the length of max decompressed binary chromosome and binary chromosome affect the performance of LZWMIMIC.

This study suggests that the length of chromosome is an important factor. An empirical analysis might be done to see the effect of the chromosome length and the algorithm effectiveness.

In this paper, we combine LZW compressed encoding with MIMIC EDA. The future work might combine another type of compression algorithms with another type of EDAs. For example, arithmetic coding compression algorithm might be combined with the Bayesian Optimization Algorithm (BOA). However, since different EDAs may have different characteristics, more work needs to be done for the generalization of our approach.

Acknowledgment

We would like to thank the Faculty of Science, Kasetsart University for the Budget for Overseas Academic Conference (BOAC) and the Graduate School, Kasetsart University for supporting this research.

References

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

[2] Mitchell, M.: An Introduction to Genetic Algorithms. MIT Press (1998)

[3] Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. The MIT Press (1992)

[4] Michalewicz, Z., Fogel, D.B.: How to Solve It: Modern Heuristics. 2nd edn, Springer-Verlag Berlin Heidelberg (2004)

[5] Mühlenbein, H., Paaß, G.: From recombination of genes to the estimation of distributions I. Binary parameters. In Lecture Notes in Computer Science 1411.: Parallel Problem Solving from Nature-PPSN IV (1996) 178-187

[6] Larrañaga, P., Lozano, J.A., eds.: Estimation of Distribution Algorithms: A New Tool for Evolutionary Computation. Kluwer Academic Publishers (2002)

[7] Paul, T. K., and Iba, H.: Linear and Combinatorial Optimizations by Estimations of Distribution Algorithms, MPS Symposium on Evolutionary Computation, 2002.

[8] Baluja, S.: Population-based incremental learning: A method for integrating genetic search based function optimization and competitive learning. Technical Report CMU-CS-94-163, Carnegie Mellon University (1994)

[9] Harik, G., Lobo, F. G., and Golberg, D. E.: The compact genetic algorithm. In Proceedings of the IEEE Conference on Evolutionary Computation (1998) 523-528

[10] De Bonet, J.S., Isbell, C.L., Viola, P.: MIMIC: Finding optima by estimating probability densities. Advances in Neural Information Processing Systems, Volume 9 (1997)

[11] Pelikan, M., Goldberg, D.E., Cantú-Paz, E.: BOA: The Bayesian optimization algorithm. Genetic and Evolutionary Computation Conference (GECCO-99) (1999) 525-532

[12] Chen, S., Smith, S.: Improving Genetic Algorithms by Search Space Reduction (with Applications to Flow Shop Scheduling). GECCO-99. Proceedings of the Genetic and Evolutionary Computation Conference, Morgan Kaufmann (1999)

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

[14] Welch, T.A.: A Technique for High-Performance Data Compression. IEEE Computer, Volume 17, Number 6 (1984) 8-19

[15] Sayood, K.: Introduction to Data Compression. 3rd edition. Morgan Kaufmann Publishers, New York (2006)

[16] Tan, L.S, Lau, S.P., Tan, C.E.: Optimizing LZW Text Compression Algorithm via Multithreading Programming. Proceedings of the 2009 IEEE 9th Malaysia International Conference on Communications, Kuala Lumpur, Malaysia, (2009) 592-596

[17] Abu Taleb, S.A., Musafa, H.M.J., Khtoom, A.M., Gharaybih, I.K.: Improving LZW Image Compression. European Journal of Scientific Research, Volume 44, Number 3 (2010) 502-509

[18] Baluja, S., Caruana, R.: Removing the Genetics from the Standard Genetic Algorithm. Proceedings of the Twelfth International Conference on Machine Learning, Lake Tahoe, CA (1995)

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 latter uses compressed encoding. 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.**

Keywords-EDA; Mutation; LZW

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. Another 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 adds mutation to 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 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. 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 cGA and 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. In LZWcGA, 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

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

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

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

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.

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

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 ratio 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.

Arithmetic Coding Differential Evolution for Binary Encoding

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

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

***Abstract*—Differential Evolution (DE) is a fast and robust real vector optimizer. In order to apply DE to solve discrete optimization problems, this paper integrates arithmetic coding decompression to the original algorithm. Experimental result shows that this approach gives high quality results and the evolution time is very fast.**

Keywords - Differential Evolution; Arithmetic coding; Discrete optimization;

I. INTRODUCTION

Differential Evolution (DE) is an evolutionary algorithm designed for solving real value optimization problems [1]. The key idea of this algorithm is using the weighted difference between two vectors to mutate another vector. DE is very fast and efficient. It was ranked the third in the First International Contest on Evolutionary Optimization in 1996. However, it is more robust than those optimizers finished before [2]. In addition, DE is very compact. The core of the algorithm can be implemented in less than 20 lines of C code, which is available on-line [3].

DE performs very well in continuous optimization. However, for discrete optimization, there are a few works that investigate DE's effectiveness [4]. This paper proposes a method for adapting DE for discrete optimization. By combining Arithmetic Coding (AC) [5] to DE, DE's real value vector can be transformed to a binary string. As a result, DE can solve discrete optimization problems.

The organization of this paper is as follows. The next section describes AC and DE. After that, in Section III, the proposed algorithm namely Arithmetic Coding Differential Evolution (ACDE) is explained. This section also explains how AC is applied to DE. Section IV explains problems and describes the experiments. Section V discusses the result. Finally, Section VI summarizes the paper.

II. RELATED WORK

This section describes Differential Evolution and Arithmetic Coding.

A. Differential Evolution

Differential Evolution (DE) is an evolutionary optimization method. The first generation of real vectors is initialized with random values. Each vector has D variables. A population consists of NP vectors.

A new generation is created by the following methods. Each vector competes with its trial vector. The one with less cost (i.e., the better one) is selected to the next generation. A trial vector is created by combining the original vector with a mutant vector. The combination is similar to crossover in Genetic Algorithm. The mutant vector is created by adding a random vector with a weight difference of other two random vectors (hence the name Differential Evolution).

B. Arithmetic Coding

Arithmetic coding compression algorithm represents a binary string by two real numbers ranged between [0, 1). The first number is the probability that zero will occurs in the binary string. The second number is the compressed message. The first number is denoted by p and the second number is denoted by c.

The coding is best explained by an illustration. The following example demonstrates a decompression of $(p, c) = (0.4, 0.6)$ to a 4-bit binary string. As shown in Fig 1, p divides the interval [0, 1) into 2 sub-intervals: [0, 0.4) and [0.4, 1). Since the compressed message c is in the second sub-interval, the algorithm outputs 1.

Next, the algorithm partitions the second interval [0.4, 1) into two sub-intervals proportional to p. The resulting sub-intervals are [0.4, 0.64) and [0.64, 1). Since the compressed message c is in the first sub-interval, the algorithm outputs 0.

Then, the algorithm partitions the first interval [0.4, 0.64) into two sub-intervals proportional to p. The resulting sub-intervals are [0.4, 0.496) and [0.496, 0.64). Since the compressed message c is in the second sub-interval, the algorithm outputs 1.

Finally, the algorithm partitions the second interval [0.496, 0.64) into two sub-intervals proportional to p. The resulting sub-intervals are [0.496, 0.5536) and [0.5536, 0.64). Since the message c is in the second sub-interval, the algorithm outputs 1.

The proposed algorithm ACDE required only a decompression algorithm. However, ACDE with local search uses both decompression and compression. A pseudo code for Arithmetic Coding decompression used in ACDE is shown in Fig 2. The algorithm runs in $O(l)$ time, where l is the number of bits to be produced.

III. ARITHMETIC CODING DIFFERENTIAL EVOLUTION

ACDE (Arithmetic Coding Differential Evolution) combines Arithmetic Coding compression algorithm (AC) with Differential Evolution (DE). DE evolves vectors of real numbers. During the evolution, AC decompresses each vector of real numbers to a binary string. After that, the binary string is evaluated and its fitness is returned to DE. Therefore, ACDE is an algorithm that evolves a population of binary strings.

By applying AC to DE, we made the following modifications to DE algorithm. The inputs of AC decompression algorithm are two numbers p and c (probability and code). In this paper, we fixed the value of the probability p to 0.5. This value can be changed if we know the distribution of the solution. Each variable in the vector is the code c. The range of the code is [0, 1). Therefore, each variable in a vector is randomly initialized within the range [0, 1). Moreover, the result of trial vector calculation has to be constraint to the range [0, 1), while there is no such constraint in the original DE.

The evaluation of a vector of real variables is normally separated from the DE algorithm. For example, in our Java implementation, fitness evaluation is done by a class that implements an interface Evaluator. Before the fitness evaluation, a vector of real variables C is decompressed to a binary string X. Each code $C[i]$ is decompressed to the binary string in the $(i \times r)^{th}$ to $((i+1) \times r - 1)^{th}$ positions, where r is a compression ratio or the number of bits that each code produces. After the decompression, the binary string is evaluated.

To improve the performance of ACDE, we add a local search to the algorithm. For each generation, the best real vector is decompressed to a binary string. The local search is done on the binary string to find a better neighbor. For each position in the binary string, a bit is mutated and the mutated binary string is evaluated. If the new binary string is not better than the old one, the bit is reverted to the old value. Otherwise, the local search is terminated. The better binary string is compressed. The best vector in that generation is replaced by the output of compression algorithm.

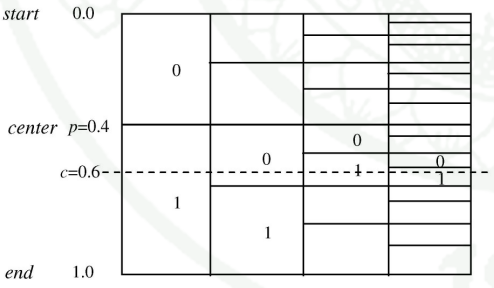


Figure 1. Decompressing 4 bits from (p, c)=(0.4,0.6). The output is 1011.

Algorithm: Arithmetic Coding Decompression

```
1:   input:   p, c : double
2:   output:  data : bit array
3:     start ← 0
4:     center ← p
5:     end ← 1.0
6:     for (i ← 0; i < data.length; i++) {
7:       if (c < center) {
8:         data[i] ← 0
9:         end ← center
10:        center ← start + (center − start) × p
11:      } else {
12:        data[i] ← 1
13:        start ← center
14:        center ← start + (end − center) × p
         }
       }
```

start is the starting point of the first interval.
center is the starting point of the second interval and the ending point of the first interval.
end is the ending point of the second interval.

Figure 2. Arithmetic Coding Decompression pseudo code

IV. EXPERIMENTS

This section describes the benchmark problems which are Random Trap, NK landscape, and Ising Spin Glass. The experimental parameters are also described in this section.

A. Problems

1) Random Trap

The general k-bit trap functions are defined as:

$$F_k(b_1...b_k) = \begin{cases} f_{\text{high}} & ; \text{if } u=k \\ f_{\text{low}} - (u \times f_{\text{low}})/(k-1) & ; \text{otherwise} \end{cases} \quad (1)$$

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

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

The m and k are varied to produce a number of test functions. The Trap functions fool the gradient-based optimizers to favor zeros, but the optimal solution is composed of all ones. The Trap function is a fundamental unit for designing test functions that resist hill-climbing algorithms.

Random Trap problem is similar to Trap problem. The difference is that instead of counting 1's in the block, the function counts the bits that are equal to the randomly preset solution. This problem is designed to reduce the effectiveness of compressed encoding evolutionary algorithm when solving a problem with high regularity solutions.

2) NK Landscape

The problem [6] is defined as follows:

$$f(X) = \sum_{i=1}^{N} k(X, i) \tag{3}$$

Where X is a binary string
i is a position in a binary string
N is the length of binary string

The value of $k(X, i)$ is obtained from a table of the size 2^k using the value from X_i to X_{i+K-1} bit as an index. Each entry in the table is randomly initialized. The fitness of this function depends on the value of the gene and its neighbors. The goal is to maximize the function. Unlike the previous problem, the optimal is unknown and depends on the seed of the random number generator. Therefore, in our experiment the performance of each algorithm is compared using the same set of seeds.

3) Ising Spin Glass

A 2D spin glass model [7] is a grid of atoms. Each atom has an atomic spin S_i which is either +1 or −1. A coupling $J_{i,j}$ between a pair of atom i and j is randomly initialized. The problem is to find a configuration of spins which has the lowest energy. The energy is obtained by the following formula

$$energy = \sum_{<i,j>} s_i J_{i,j} s_j \tag{4}$$

B. Experiments

We conducted an experiment to compare the performance of ACDE with simple real to binary conversion DE. The simple conversion simply converts each a real value in the vector to a binary using the rule ($X_i < 0.5$? 0 : 1). We simply referred to this method as DE.

Table I shows the experimental parameters. The compression ratio (i.e., the number of decompressed bits per code) is set to 5 because the problem size is divisible by this number. Any numbers that can divide the problem size can be used but the result may not be the same. The population size for DE and ACDE is set to 10 times the number of variables in a vector. However, since the compression ratio of ACDE is set to 5, the population is ACDE is 5 times smaller than that of DE. All experimental results are the average performance obtained from 30 runs.

TABLE I. EXPERIMENTAL PARAMETERS

Parameter	Value		
	RandomTrap (Trap size: 5)	*NKlandscape*	*IsingSpinGlass*
Problem size (bits)	50	100	25, 100, 225, 400
Pop. size (DE)	500	1000	250, 1000, 2250, 4000
Pop. size (ACDE, ACDE-local)	100	200	50, 200, 450, 800
ACDE's compression ratio	5		
Maximum generations	200		

V. RESULTS

The results are presented in Table II to IV. The performance of ACDE is better than the simple real to binary conversion scheme (DE) in terms of solution quality and time. For Trap and NK-Landscape problem, the higher the fitness value means the better solution. However, in our experiment, the lower value is the better because we multiply the fitness function with −1. For Ising Spin Glass, the lower fitness value already means the better solution.

Visualizations of Table III and IV are in Fig 3. ACDE with local search gives the best result in both NK-Landscape and Ising Spin Glass problems. In subfigure (a), the X-axis shows the number of K in NK-Landscape problem. The larger K makes the problem more difficult. In subfigure (b), the X-axis shows the *width* of Ising Spin Glass model. The problem size is equal to $width^2$. ACDE with local search scales better than DE and ACDE.

TABLE II. SOLVING 50-BIT RANDOM TRAP PROBLEM

Algo.	Average run					
	F	*CR*	*Fitness*	*Eval.*	*Time (ms)*	*Found (%)*
DE	0.9	0.9	-48.40	99,450.00	171.50	27
ACDE	1.0	0.1	-50.00	12,107.27	30.17	**100**
ACDE-local	1.0	0.1	-50.00	17,912.57	31.20	**100**

TABLE III. SOLVING 100-BIT NK-LANDSCAPE PROBLEM

Algo. / K	Average run seed 0-29, F = 0.5, CR = 0.1					
	DE		ACDE		ACDE-local	
	Fit.	*Time (ms)*	*Fit.*	*Time (ms)*	*Fit.*	*Time (ms)*
4	-7,248.08	835.67	-7,730.33	312.53	-7,780.43	325.20
6	-7,072.41	957.83	-7,602.02	340.60	-7,715.08	370.00
8	-6,974.24	1,113.83	-7,408.89	380.67	-7,504.06	428.37
10	-6,830.64	1,253.73	-7,160.05	422.23	-7,324.18	476.83

TABLE IV. SOLVING ISINGSPINGLASS PROBLEM

Algo. / Width	Average run seed 0-29					
	DE		ACDE		ACDE-local	
	Fit.	*Time (ms)*	*Fit.*	*Time (ms)*	*Fit.*	*Time (ms)*
5	-29.73	53.57	-29.67	10.60	-26.33	2.70
10	-125.87	787.30	-73.73	286.40	-100.52	28.83
15	-147.40	4,299.20	-116.67	2,439.27	-227.00	547.05
25	-195.2	13,503.43	-161.00	12,021.83	-401.11	4,793.59

VI. CONCLUSIONS

This paper proposes a method to apply DE to solve discrete optimization problems. We tested the problem with difficult benchmarks. The result shows that our proposed method can give high quality result in a very short amount of time. Moreover, it scales best among tested algorithms.

REFERENCES

[1] R. Storn and K. Price, "Differential Evolution - A simple and efficient adaptive scheme for global optimization over continuous spaces," 1995.

[2] K. V. Price, R. M. Storn and J. A. Lampinen, "Differential Evolution: A Practical Approach to Global Optimization," Springer, 2005.

[3] K. Price and R. Storn, Differential Evolution, http://www.drdobbs.com/architecture-and-design/184410166, 1997.

[4] T. Gong and A. Tuson, "Differential Evolution for Binary Encoding," Soft Computing in Industrial Applications, vol. 39, pp. 251-262, 2007.

[5] J. Rissanen, "Generalized Kraft Inequality and Arithmetic Coding," IBM J. Res. Develop., vol. 20, pp. 198-203, 1976.

[6] M. Pelikan, "NK Landscapes, Problem Difficulty, and Hybrid Evolutionary Algorithms," GECCO'10, Portland, Oregon, USA, pp. 665-672, July, 2010.

[7] R. Kindermann and J. L. Snell., Markov Random Fields and Their Applications. AMS, 1980.

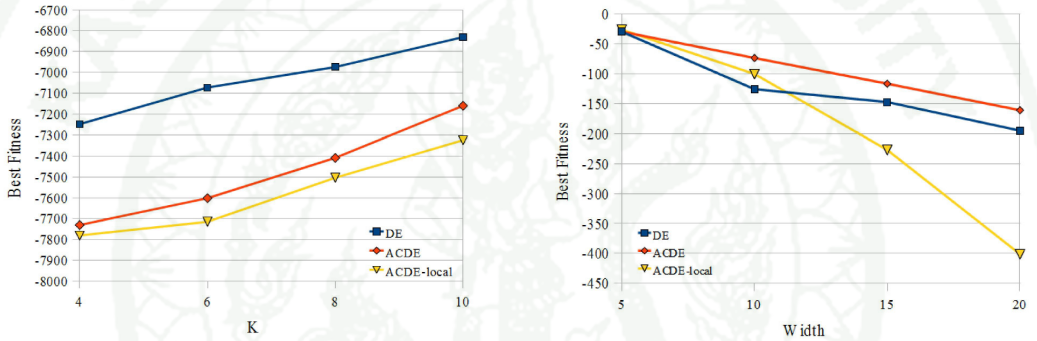


Figure 3. Average best fitness of (a) 100-bit NKlandscape problem and (b) Ising Spin Glass problem.

LZW Differential Evolution for Binary Encoding

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

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

***Abstract*—Differential Evolution (DE) is a fast and robust real vector optimizer. This paper applies DE to discrete problems by converting a real chromosome to an integer chromosome and then decompress to a binary chromosome using LZW algorithm. Experimental result shows that this approach is better than the previous work and the evolution time is very fast.**

Differential Evolution; LZW; Discrete optimization;

I. INTRODUCTION

Differential Evolution (DE) is an evolutionary algorithm designed for solving real value optimization problems [1]. DE is very fast and efficient. It was ranked the third in the First International Contest on Evolutionary Optimization in 1996. However, it is more robust than those optimizers finished before [2]. In addition, DE is very compact. The core of the algorithm can be implemented in less than 20 lines of C code, which is available on-line [3].

DE performs very well in continuous optimization. However, for discrete optimization, there are a few works that investigate DE's effectiveness [4]. This paper presents two alternative methods for adapting DE for discrete optimization. The first method directly maps a real value chromosome to a binary chromosome. The second method combines compressed chromosome encoding with DE. Compressed encoding enables evolutionary algorithm to solve very large problems [5][6][7]. For example, LZW encoding in Genetic Algorithm can solve one-million-bit problems. To use compression with GA, the individual is in a compressed form and has to be decompressed before the fitness evaluation. Another advantage of this approach is low memory requirement.

The organization of this paper is as follows. The next section describes DE. After that, in Section 3, LZW compression algorithm is explained. LZW is used for decompressing a chromosome. This section also explains how LZW compressed encoding is applied to DE. Section 4 explains benchmark problems. Section 5 describes the experiment. Section 6 discusses the result. Finally, Section 7 summarizes the paper.

II. DIFFERENTIAL EVOLUTION

Differential Evolution (DE) is an evolutionary optimization method. The first generation of real vectors is created by randomly filled the values in the vectors. Each vector has D values. A population consists of NP vectors. There are two schemes (i.e., DE1 and DE2) presented in [4]. In this paper, DE1 is used.

A new generation is created by the following method. Each vector competed with its trial vector. The one with less cost is selected to the next generation. A trial vector is created by combining the vector with a mutant vector. The combination is similar to crossover in Genetic Algorithm [8]. A mutant vector is created by adding a random vector with a weight difference of other two random vectors (hence the name Differential Evolution). The mathematical formula for creating a mutant vector is as follows:

$$X'_c = X_c + F(X_a - X_b)$$

The parameters in DE are listed below.

- NP (or population size) should be 5-10 times the number of parameters D.
- F (i.e., the weight) should start with 0.5. F and NP should be increased if the algorithm converges prematurely.
- CR (or the crossover rate) should be 0.9, 0.1, or 0.

III. LZW DIFFERENTIAL EVOLUTION

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. The dictionary does not have to be stored because the algorithm can construct the dictionary during the compression or decompression process.

To use LZW compressed encoding with DE, we add a conversion and decompressing step before a fitness evaluation.

The real value chromosome is converted to an array of integers. After that, the array is decompressed to a binary string. Because LZW cannot decompress arbitrary input, each code in an integer array must satisfy the following constraint [6].

$0 \le a_i \le i+2$, where i is a zero-base array index

Any positive integer can be changed to satisfy the constraint by modulo with $i+3$. An example of converting a random real vector to a binary string is shown in Figure 1.

Note that LZWDE evolves a direct representation of an individual as a "compressed" string. There is no compression step involved in LZWDE.

Implementing an LZW chromosome encoding in object-oriented language is easy. The core algorithm does not have to be modified to support LZW encoding. Rather, for each benchmark problem, we implemented the interface for fitness evaluation using two classes: one for a normal chromosome and the other is for a compressed chromosome. For DE's point of view, it still evolves real vectors. It does not know that it is evolving compressed encoding chromosomes.

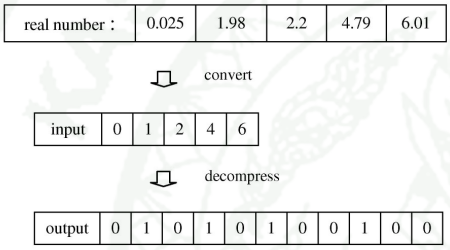


Figure 1. Converting a real value chromosome to an integer chromosome and decompress it to a binary chromosome

IV. 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 a simpler structure.

In [4], the author applied DE to solve the following discrete problems: OneMax, Royal Road, Order-3 Deceptive, and Long Path problems. We test the performance of our algorithm using the same benchmark. Every benchmark problem is a maximization problem. However, since DE is a global minimize, the fitness is transformed by multiplying the cost function with -1.

A. OneMax Problem

The OneMax problem [10] (or bit counting) is a widely used problem for testing the performance of various genetic algorithms. Formally, this problem can be described as finding a string $X=\{x_1, x_2, ..., x_k\}$, where $x_i \in \{0,1\}$, that maximizes the following equation:

$$F(X) = \sum_{i=1}^{k} x_i \quad (1)$$

B. Royal Road Problem

Royal Road problem [11] is designed to investigate the role of GA crossover and building block hypothesis. The problem can be solved using GA which uses crossover. However, it is difficult for a hill climbing algorithm or GA with a single-bit mutation to solve the problem.

This function involves a set of schemas $S = \{s_1,\ldots,s_{20}\}$, and is defined as:

$$F(X) = \sum_{s \in S} c_s \sigma_s \quad (2)$$

where x is a bit string, each c_s is a value assigned to the schema s.

C. Deceptive Order-3 Problem

In Deceptive problem [12], an individual composes of several blocks. Each of the blocks is evaluated by a deceptive function. The deceptive 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. The order-3 deceptive function is defined as:

$$\begin{aligned} f(000) &= 28 \\ f(001) &= 26 \\ f(010) &= 22 \\ f(100) &= 14 \\ f(011) &= 0 \\ f(101) &= 0 \\ f(110) &= 0 \\ f(111) &= 30 \end{aligned}$$

The deceptive problem can be decomposed to several deceptive functions. The problem, denoted by f_m, is defined as:

$$f_m(K_1 \cdots K_m) = \sum_{i=1}^{m} f(K_i), K_i \in \{0,1\}^3 \quad (3)$$

D. Long Path Problem

Long Path problem [13] is a problem that can be solved by a hill-climbing algorithm. However, it is not practical to solve this problem using hill climbing algorithm. This is because the length of hill (or the path) is exponentially long. Each point in the path is differed by one bit. The path is constructed such that is exponentially long. The height from the bottommost of the hill to the top is equal to:

$$HillHeight\ (l) = 3 \times 2^{\lfloor (l-1)/2 \rfloor} + l - 2 \quad (4)$$

where l is a chromosome length.

V. EXPERIMENT

We conducted the experiment to compare the performance of LZWDE with Gong and Tuson's binary adapted DE operators [4] and with simple real to binary conversion DE. The latter scheme, which is simply called DE, converts a real value to a binary using the rule $(X_i < 0.5 \ ? \ 0 : 1)$

Table I shows the experimental parameters. The length of an LZWDE chromosome is less than DE chromosome which are 1/5 of OneMax problem's problem size, 1/4 of Royal Road problem's problem size, 1/12 of Deceptive order-3 problem's problem size, and about 1/3 of Long Path problem's problem size. Before a fitness evaluation, the compressed chromosome is decoded and decompressed with LZW decompression algorithm. The length of the decompressed binary chromosome is varied depending on the code in the integer array. If the length is more than the size of the problem size, the excess bits are discarded. However, if the length is less than the problem size, LZWDE will evaluate the fitness of available bits. All experimental results are the average performance obtained from 30 runs.

TABLE I. EXPERIMENTAL PARAMETERS

Parameter	OneMax	Royal Road	Deceptive order-3	Long Path
Population size	50	30	100	30
Problem size	500	80	300	29
LZW chromosome length	**100**	**20**	**25**	**10**
Maximum generation	500	500	2000	300

VI. RESULTS

Gong and Tuson [4] used different sets of parameters for OneMax, Royal Road, Deceptive Order-3 and Long Path problems. They reported the result of 4 DE strategies which are: 1) any-change mutation and exponential crossover-DE/any/exp, 2) any-change mutation and binomial crossover-DE/any/bin, 3) restricted-change mutation and exponential crossover-DE/res/exp, and 4) restricted-change mutation and binomial crossover-DE/res/bin for each problem. We choose the best of their experimental results and compare them with our best parameters for each problem.

For each benchmark problem, we compare the performance of binary-adapted DE, DE (simple real to binary conversion), and LZWDE. The result is shown in Figure 2. The X-axis shows the number of generations and the Y-axis shows the average-best fitness. LZWDE outperforms both DE and binary-adapted DE. Moreover, it is interesting to see that the performance of simple conversion is comparable to binary-adapted DE in Royal Road problem and better than binary-adapted DE in Long Path problem.

Table II shows the average evolution time. We ran the experiment on Intel Core i5 with 4GB of RAM. In this table, we report only the time that DE successfully finds the solution. The number in the parenthesis is the success rate. LZWDE can find the solution for every run. We do not have the data for binary-adapted DE. Therefore, we only compare the time of DE and LZWDE. In LZWDE, there is an LZW decompression step. Even with this step, the algorithm can still find a solution faster than DE. Simple DE cannot find a solution for Deceptive Order-3 problem.

TABLE II. AVERAGE EVOLUTION TIME (IN MILLISECONDS)

Problem	Algorithm	
	DE	***LZWDE***
OneMax	388.43 (100)	8.83 (100)
Royal Road	36.42 (87)	34.83 (100)
Deceptive order-3	- (0)	800.73 (100)
Long Path	13.53 (100)	11.43 (100)

VII. CONCLUSIONS

This paper proposes two methods to apply DE to solve discrete optimization problem. The first is simple real-to-binary conversion. The second is using LZW encoding. We compared the result with binary-adapted DE using the same benchmark problems. The result shows that LZWDE outperforms binary-adapted DE. In addition, in term of computation time, LZWDE is very fast even it has to decompress the chromosome. It can solve all benchmark problems in less than one second.

REFERENCES

[1] R. Storn and K. Price, "Differential Evolution - A simple and efficient adaptive scheme for global optimization over continuous spaces," 1995.

[2] K. V. Price, R. M. Storn and J. A. Lampinen, "Differential Evolution: A Practical Approach to Global Optimization," Springer, 2005.

[3] Kenneth Price and Rainer Storn, Differential Evolution, http://www.drdobbs.com/architecture-and-design/184410166, 1997.

[4] T. Gong and A. Tuson, "Differential Evolution for Binary Encoding," Soft Computing in Industrial Applications, vol. 39, pp. 251-262, 2007.

[5] 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.

[6] 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.

[7] W. Suwannik and P. Chongstitvatana, "Solving One-Billion Bit Noisy OneMax Problem using Estimation Distribution Algorithm with Arithmetic Coding," Proceedings of IEEE Congress on Evolutionary Computation (CEC 2008), pp. 1203-1206, 2008.

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

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

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

[11] M. Mitchell, S. Forrest, and J. H. Holland, "The Royal Road for Genetic Algorithms: Fitness Landscapes and GA Performance," in Proc. The First European Conference on Artificial Life, Cambridge, MA, MIT Press, 1991, pp. 245-254.

[12] D. E. Goldberg, "Genetic algorithms and Walsh functions: Part I, a gentle introduction," Complex Systems, vol. 3, pp. 129-152, 1989.

[13] J. Horn, D. E. Goldberg, and K. Deb, "Long Path Problems," Lecture Notes in Computer Science, vol. 866, pp. 149-158, 1994.

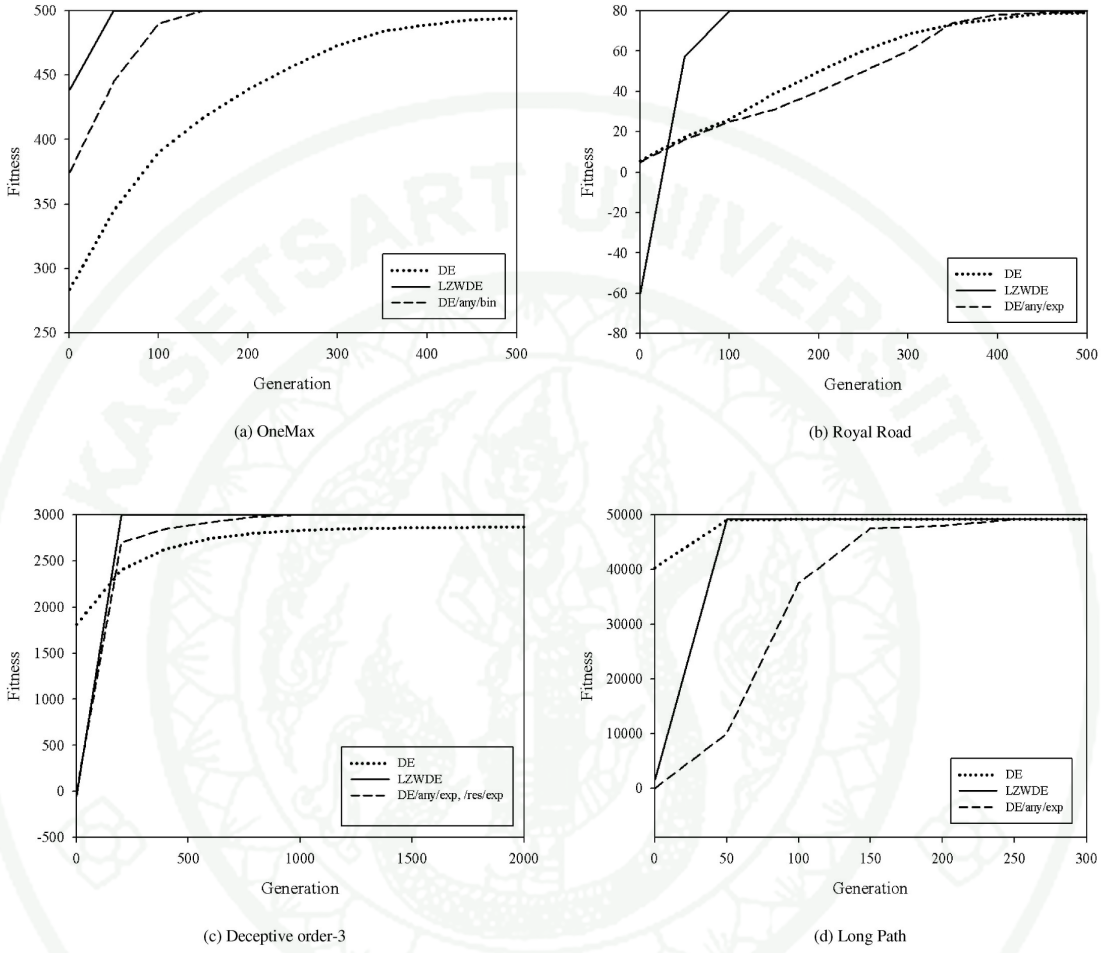


Figure 2. The average-best fitness plotted against generation

Arithmetic Coding Differential Evolution with Local Search

Orawan Watchanupaporn and Worasait Suwannik
Department of Computer Science, Kasetsart University, Bangkok 10900, Thailand

Differential Evolution (DE) is an effective continuous real value optimizer. In this paper, we apply DE to discrete optimization problems by using Arithmetic Coding compression and local search algorithm, which are hill climbing and simulated annealing. The test functions include random-version of widely used benchmark problems. We compare the performance of each algorithm based on the quality of solution and the running time. The result is an effective and very fast discrete optimization algorithm. Experimental results indicate that our proposed algorithm outperforms a sophisticated discrete optimizer in both solution quality and running time.

Keywords: Differential Evolution, Compression, Binary Optimization.

1. INTRODUCTION

Evolutionary algorithms (EAs) are probabilistic optimization methods based on the model of natural evolution. A common characteristic of this algorithm is population based. An individual in a population is evaluated and selected based on its fitness value. The fitness value indicates how well an individual is for a specific optimization problem. The selected individuals will be likely to survive to the next generation or they can generate offspring that will be put into the next generation. The offspring can be generated using a nature-inspired mechanism such as mutation or crossover. The process of fitness evaluation, selection, and creating a next generation is repeated until the solution is found or another termination criterion is met.

EA differs by representation of an individual. For example, Genetic algorithm (GA), proposed by Holland, uses binary string to represent individuals[1,2]; Genetic Programming's individual is a program tree[3,4]; individual in Evolutionary Strategy (ES) is a real value vector[5]. Different representations have direct affects on the genetic operators. For example, mutation in GA is simple bit inverting while in GP a new subtree is grown.

*Email Address: orawan.liu@gmail.com

In addition, the representation also affects the foundation of theoretical establishment. For instance, a schema theorem that proves the effectiveness of GA cannot be applied to GP. Finally, and most importantly, the representation affects the class of the problems that they can solve efficiently. GA and its descendant, Estimation of Distribution Algorithm (EDA), are doing well in binary search space. Various benchmarks have been developed to show its strength over other approaches. For example, a Trap problem[6] and a Long Path problem[7], which are designed to resist a hill climbing search algorithm, can be solved by several EDAs.

While GA and EDA are normally used for binary optimization, ES and Differential Evolution (DE), are popular real value optimizers. ES was proposed by Rechenberg in 1960s[5]. DE, introduced by Price and Storn in 1990s[8], is implemented in Mathematica as NMinimize[9] and also in Matlab. DE is an evolutionary algorithm designed for solving real value optimization problems. DE is very compact. The core of the algorithm can be implemented in less than 20 lines of C code, which is available on-line[10]. In addition, DE is very fast and efficient. It was ranked the third in the First International Contest on Evolutionary Optimization in 1996. However, it is more robust than those optimizers finished before[11].

The speed and effectiveness of DE in continuous value optimization appeals many researchers[12-15]. However, for discrete optimization, there are a few works that investigate DE's effectiveness[16,17]. This paper presents methods for applying DE to discrete optimization problem. The first approach is simple real to binary conversion scheme, which we will refer as its original name, DE. This scheme is straightforward but quite wasteful because one variable in DE, which is normally 64-bit long, represents one bit. Thus, the second approach combines conversion with a compression algorithm to create a binary string. We used Lempel-Ziv-Welsh (LZW) compression algorithm and Arithmetic Coding (AC) compression algorithm. For example, in LZW approach, a real vector is converted to an integer array. After that, the array is sent to LZW decompression algorithm. LZW will output binary string which will be used for in a discrete problem fitness evaluation.

Compressed encoding is compact and enables evolutionary algorithm to solve very large problems. For example, LZW encoding in Genetic Algorithm can solve one-million-bit problems[18,19]. To use compression with evolutionary algorithm, the individual is stored in a compressed form and has to be decompressed before the fitness evaluation. Another advantage of this approach is low memory requirement which leads to faster running time per generation due to less data transfer.

However, the disadvantage of compressed encoding is the bias toward high regularity solution. Chamlamai and Suwannik created a random version of an existing benchmark[20]. The experimental result shows that the performance of LZW encoding is decreased in the random version while the performance of uncompressed encoding remains the same.

To solve the random-version of benchmark problem, we add two types of local search to our scheme. We combined simple hill climbing and Simulated Annealing[21] (SA) in our search method. The result is compared against the Bayesian Optimization Algorithm[22] (BOA), which is one of the best EDA.

The remainder of this paper is organized as follows. The next section presents the background and related work. Section 3 describes the benchmark problems. Section 4 describes the experimental parameters. Section 5 discusses the result. Finally, section 6 concludes the paper.

2. BACKGROUND AND RELATED WORK

In this section, we describe algorithms and techniques that are used in our experiment.

2.1 Bayesian Optimization Algorithm (BOA)

EDA are the next generation of GA. EDA can be classified based on the assumption about relationship among optimized variables. A univariate EDA assumes no relationship among variables. A bivarate EDA assumes pairwise relationship. A multivariate EDA assumes relationship among group of variables. Thus, multivariate EDA can solve more complex problems than the previous classes of EDA.

BOA is a multivariate EDA. A Bayesian network is a directed acyclic graph (DAG) that represents a set of random variables and their dependencies. In BOA, a Bayesian network can be used to model inter-dependencies among chromosome positions. Bayesian network is constructed as a model of selected individuals. Any search method and any metric can be used to construct the network. In BOA, the construction uses a greedy search which adding edges from an empty graph. Networks are scored using Bayesian-Dirichlet Equivalence metric[22].

BOA can solve difficult problems such as 3-deceptive, trap-5, and 6-bipolar. However, it is very time consuming. Its time complexity for each iteration is $O(n^2N + n^3)$, where n is the length of a chromosome and N is the size of selected individuals.

2.2 Differential Evolution

Differential Evolution (DE) is an evolutionary optimization method. The first generation of real vectors is created randomly. Each vector has D values. A population consists of NP vectors. There are two schemes (i.e., DE1 and DE2) presented in Storn and Price[8]. In this paper, DE1 is used.

A new generation is created by the following method. Each vector competes with its trial vector. The one with less cost is selected to the next generation. A trial vector is created by combining the vector with a mutant vector. The combination is similar to crossover in Genetic Algorithm. A mutant vector X_c' is created by adding a random vector X_c with a weight difference of other two random vectors $F(X_a - X_b)$ (hence the name Differential Evolution). The mathematical formula for creating a mutant vector is as follows:

$$X_c' = X_c + F(X_a - X_b) \quad (1)$$

DE's parameters and their suggested settings (by its inventor[10]) are listed below.

NP (or population size) should be 5-10 times the number of parameters D.

F (i.e., the weight) should start with 0.5. F and NP should be increased if the algorithm converges prematurely.

CR (or the crossover rate) should be 0.1. However, one might try CR = 0.9, 0.1, or 0.0.

2.3 LZW Differential Evolution

Generally, DE evolves vectors of real numbers. However, DE can be applied to solve problem that has binary solution. The method is as follows. Before the fitness evaluation, a DE real vector is converted to an array of integer by removing the fraction part. After that,

the array is sent to LZW compression algorithm, which will output a binary string. Finally, the binary string is evaluated according to a fitness function. The fitness will be returned to DE. Therefore, LZWDE can be considered an algorithm that evolves a population of binary strings.

2.3.1 Lempel-Ziv-Welch (LZW)

LZW is a lossless dictionary-based data compression algorithm[23]. 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 or decompression algorithm starts 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 or 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 Evolutionary Algorithm, 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. The dictionary does not have to be stored because the algorithm can construct the dictionary during the decompression process.

2.3.2 LZWDE

To use LZW compressed encoding with DE, we add a conversion and decompressing step before a fitness evaluation. As shown in Figure 1, the real value DE vector is converted to an LZW chromosome, which is an array of integers. To convert a real number to an integer, a fraction part is truncated. After that, the integer array is decompressed to a binary string. LZW decompression algorithm cannot decompress arbitrary input. Each code in an integer array must satisfy the following constraint [24].

$$0 \le a_i \le i + 2 \qquad (2)$$

where i is a zero-based array index.

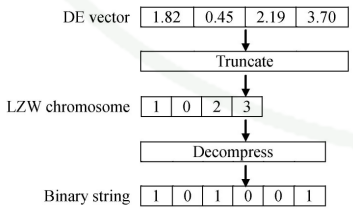


Fig.1. LZWDE Encoding.

2.4 Arithmetic Coding Differential Evolution

To solve a binary problem with DE, the simplest scheme is to convert a real vector to a binary string. The simple conversion simply converts each a real value in the vector to one bit using the rule ($X_i < 0.5$? 0 : 1). In this paper we referred to this method as DE. However, this scheme is quite wasteful because the length of one real variable is 32 or 64 bits.

Arithmetic Coding Differential Evolution (ACDE) efficiently uses a real variable. It combines Arithmetic Coding compression algorithm (AC) with Differential Evolution (DE). DE evolves vectors of real numbers. During the evolution, AC decompresses each vector of real numbers to a binary string (see Fig. 2). After that, the binary string is evaluated and its fitness is returned to DE. Therefore, ACDE is an algorithm that evolves a population of binary strings.

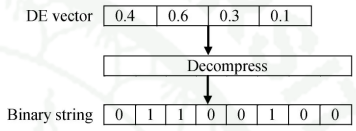


Fig.2. ACDE Encoding.

By applying AC to DE, we made the following modifications to DE algorithm. The inputs of AC decompression algorithm are two numbers p and c (probability and code) that are a real number and the output is a binary number. For example the input (0.4, 0.6) will produce the output 1011 (see Fig. 3). In this paper, instead of evolving both p and c, we fixed the value of the probability p to 0.5. This value can be changed if we know the distribution of the solution. Each variable in the optimized vector is the code c. The range of the code is [0, 1). Moreover, the result of trial vector calculation has to be constraint to the range [0, 1), There is no such constraint in the original DE.

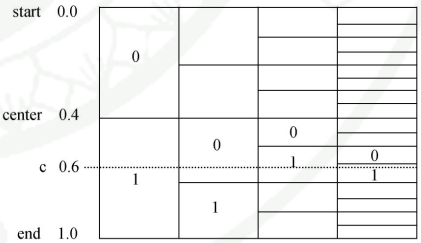


Fig.3. Decompress (p,c) = (0.4, 0.6) to a 4-bit binary string. The output is 1011.

The evaluation of a vector of real variables is normally separated from the DE algorithm. For example, in our Java implementation, fitness evaluation is done by a class that implements an interface Evaluator. Before the fitness

evaluation, a vector of real variables C is decompressed to a binary string X. Each code $C[i]$ is decompressed to the binary string in the $(i \times r)^{th}$ to $((i+1) \times r - 1)^{th}$ positions, where r is a compression ratio or the number of bits that each code produces. After the decompression, the binary string is evaluated.

The pseudo code for decompressing a DE vector C to a binary string X is as follows.

```
Function Decompress(C)
1     BEGIN
2       X = allocate a binary string with the length
            r×C.length
3       For i=0 to C.length−1
4         BEGIN
5           start = i×r
6           end = (i+1)×r−1
7           X[start..end] = AC_decompress(C[i])
8         END
9         Return X
10    END
```

where

C is a real value vector.

X is a binary string.

r is a number of bits that each code will be decompressed.

2.5 ACDE with Local Search

To further improve the performance of ACDE, we add a local search to the algorithm. For each generation, the best real vector is decompressed to a binary string. Then, the local search is applied on the binary string to find a better neighbor. In this paper, we use DE as global search and employ two local search techniques: hill climbing (HC) and simulated annealing (SA). After that, the output from the local search is compressed into a real value vector. Finally, the best vector in that generation is replaced by the output of compression algorithm (see Fig. 4).

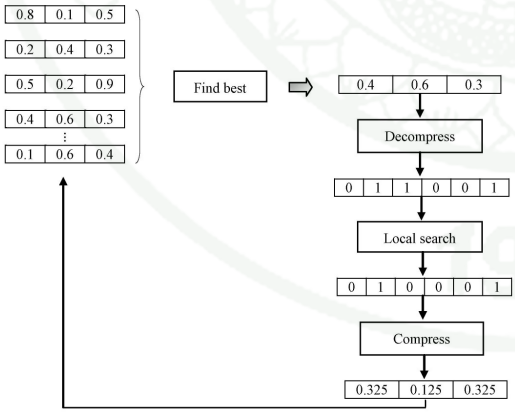


Fig.4. Local search in ACDE.

3. TEST FUNCTIONS

This section describes the benchmark problems studied in this work. The problems include RandomMax, Random Trap, NK landscape, and Ising Spin Glass. RandomMax and Random Trap is a random-version of OneMax and Trap respectively.

3.1 RandomMax

OneMax problem is a widely used problem for testing the performance of various genetic algorithms. The problem can easily be solved even by univariate EDA. The problem can be called bit counting. Formally, this problem can be described as finding a string $\vec{x} = \{x_1, x_2, \ldots, x_k\}$, where x_i in $\{0,1\}$, that maximizes the following function:

$$F(x) = \sum_{i=1}^{k} x_i \tag{3}$$

A random version of OneMax is called RandomMax. This problem was designed to be unfavorable against an optimizer that has a bias toward high regularity solution such as LZW compressed encoding GA[20]. Instead of counting the number of 1's in the binary string, this function counts the number of matches between a binary string and a prespecified random binary string. Although this problem is difficult for LZWGA, it can still easily be solved even by univariate EDA.

3.2 Random Trap

The general k-bit trap functions are defined as:

$$F(x) = \begin{cases} f_{high} & ; \text{if } u = k \\ f_{low} - \dfrac{u \times f_{low}}{k-1} & ; otherwise \end{cases} \tag{4}$$

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

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

$$F_{m \times k}(K_1 \ldots K_m) = \sum_{i=1}^{m} F_k(K_i), \qquad K_i \in \{0,1\}^k \tag{5}$$

The m and k are varied to produce a number of test functions. The Trap functions fool the gradient-based optimizers to favor zeros, but the optimal solution is composed of all ones. The Trap function is a fundamental unit for designing test functions that resist hill climbing algorithms.

Random Trap problem is similar to Trap problem. The difference is that instead of counting 1's in the block, the function counts the bits that are equal to the randomly preset solution. This problem is designed to reduce the effectiveness of compressed encoding evolutionary

algorithm in solving a problem with high regularity solutions.

3.3 NK Landscape

The problem is defined as follows[25]:

$$F(x) = \sum_{i=1}^{N} k(X, i) \quad (6)$$

where X is a binary string,
i is a position in a binary string (base-1 array),
N is the length of binary string.

The value of $k(X, i)$ is obtained from a table of the size 2^K using the value from X_i to X_{i+K-1} bit as an index. Each entry in the table is randomly initialized. The fitness of this function depends on the value of the gene and its neighbors. The goal is to maximize the function. Unlike the previous problem, the optimal is unknown and depends on the seed of the random number generator. Therefore, in our experiment, the performance of each algorithm is compared using the same set of seeds.

3.4 Ising Spin Glass

A 2D spin glass model is a grid of atoms[26]. Each atom has an atomic spin S_i which is either +1 or −1. A coupling $J_{i,j}$ between a pair of atom i and j is randomly initialized. The problem is to find a configuration of spins which has the lowest energy. The energy is obtained by the following formula:

$$energy = \sum_{<i,j>} s_i J_{i,j} s_j \quad (7)$$

4. EXPERIMENTAL PARAMETERS

Table 1 shows the experimental parameters. We varied the problem size from 100 to 800 to see the scalability of each algorithm. The population sizes for each algorithm are varied. DE is given largest population while ACDE and ACDE with local search are given the smallest size. Normally, for evolutionary algorithm, more population is likely to give higher quality solution but requires more computation time. All experimental results are the averaged performance obtained from 30 runs.

Table.1. Experimental Parameters.

PARAMETER	VALUE
Problem size (bits)	100, 400, 800/900(for Ising)
Population size (DE)	Problem size × 10
Population size (LZWDE)	(Problem size ÷ 2) × 10
Population size (ACDE, ACDE-local)	(Problem size ÷ 5) × 10
Maximum generation	200

For Ising Spin Glass, we varied the width from 10 to 30. For NK Landscape, we consider table of the size K = 4. When K=1, there is no dependency among variable. The larger K means more dependencies.

5. EXPERIMENTAL RESULTS

We compare the performance of each optimization algorithm based on the quality of the solution. Optimization will stop when the optimum is found. However, for the case that the optimal is unknown, we are interested in the best fitness value obtained when we reach the maximum number of generations. In our experiment, for RandomMax, Random Trap, and NK Landscape, the lower value is the better because we multiply the fitness function with −1. For Ising Spin Glass, the lower fitness value already means the better solution.

Experimental results are shown in Table 2. LZWDE performs poorly but outperforms DE only in 400- and 800-bit NK Landscape problem. Since we are interested in solving large problem, the focus would be at the result from 800-bit problem. At this problem size, adding compression can improve the effectiveness. ACDE can give solution with 7.45% better fitness value than DE. Adding local search can improve the optimization even further. ACDE with hill climbing and ACDE with SA is 29.87% and 37.81% better than ACDE. When compared to sophisticated discrete optimization BOA, ACDE with SA can give 10.14% better quality and 28.61 times much faster than BOA.

6. CONCLUSIONS

ACDE combines two completely different types of algorithm together. Arithmetic Coding is a compression algorithm. Differential Evolution (DE) is a real value optimizer. This combination allows DE to solve binary optimization problems. In this paper, we apply local search to ACDE and test them with random version of existing benchmark. The experimental results indicate that the proposed algorithm outperforms one of the best multivariate EDA in term of solution quality and running time.

REFERENCES

[1] J. H. Holland. Outline for a logical theory of adaptive systems. J.Assoc. Comput. Mach.(ACM), 3 (1962) 297-314.

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

[3] J. R. Koza. Genetic programming: a paradigm for genetically breeding populations of computer programs to solve problems. Technical Report STANCS-90-1314, Department of Computer Science, Stanford University, (1990).

[4] J. R. Koza. Genetic Programming on the Programming of Computers by Means of Natural Selection. Cambridge, MA: MIT Press, (1992).

[5] H. -G. Beyer and H. -P. Schwefel. Evolution Strategies: A Comprehensive Introduction. Journal Natural Computing, 1(1), (2002) 3–52.

[6] D. H. Ackley. A connectionist machine for genetic hill climbing. Boston: Kluwer Academic, (1987).
[7] J. Horn, D. E. Goldberg, K. Deb. Long Path Problems. Lecture Notes in Computer Science, 866 (1994) 149-158.
[8] R. Storn and K. Price. Differential Evolution - A simple and efficient adaptive scheme for global optimization over continuous spaces. (1995).
[9] Wolfram MathWorld. Differential Evolution. http://mathworld.wolfram.com/DifferentialEvolution.html, accessed on Dec. 15, 2012.
[10] K. Price, R. Storn. Differential Evolution. http://www.drdobbs.com/architecture-and-design/184410166, (1997), accessed on Jul. 8, 2012.
[11] K. V. Price, R. M. Storn, J. A. Lampinen. Differential Evolution: A Practical Approach to Global Optimization. Springer, (2005).
[12] J. Vesterstrøm, R. Thomsen. A Comparative Study of Differential Evolution, Particle Swarm Optimization, and Evolutionary Algorithms on Numerical Benchmark Problems. IEEE International Congress on Evolutionary Computation, 2 (2004) 1980 – 1987.
[13] R. Storn, K. Price. Differential Evolution–A Simple and Efficient Heuristic for Global Optimization over Continuous Spaces. Journal of Global Optimization, Kluwer Academic Publishers. Printed in the Netherlands, 11 (1997) 341–359.
[14] S. Muelas, A. LaTorre, J. Peña. A Memetic Differential Evolution Algorithm for Continuous Optimization. Ninth International Conference on Intelligent Systems Design and Applications, IEEE, (2009) 1080-1084.
[15] A. K. Qin, V. L. Huang, P. N. Suganthan. Differential Evolution Algorithm with Strategy Adaptation for Global Numerical Optimization. IEEE Transactions on Evolutionary Computation, 13 (2009) 398-417.
[16] T. Gong, A. Tuson. Differential Evolution for Binary Encoding. Soft Computing in Industrial Applications, 39 (2007) 251-262.
[17] D. Lichtblau. Differential Evolution in Discrete Optimization. International Journal of Swarm Intelligence and Evolutionary Computation, 1 (2012).
[18] N. Kunasol, W. Suwannik, P. Chongstitvatana. Solving One-Million-Bit Problems Using LZWGA. Proceedings of International Symposium on Communications and Information Technologies (ISCIT), (2006) 32-36.
[19] W. Suwannik, P. Chongstitvatana. Solving One-Billion Bit Noisy OneMax Problem using Estimation Distribution Algorithm with Arithmetic Coding. Proceedings of IEEE Congress on Evolutionary Computation (CEC), (2008) 1203-1206.
[20] S. Chamlamai and W. Suwannik. Benchmark Problems for Compressed Genetic Algorithm. Proceedings of Electrical Engineering Conference (EECON-30) (abstract in English), (2007) 653-656.
[21] L. Ingber. Simulated annealing: Practice versus theory. Mathematical and Computer Modelling, 18:11(1993) 29-57.
[22] M. Pelikan, D. E. Goldberg, E. Cantú-Paz. BOA: The Bayesian Optimization Algorithm. Proceedings of The Genetic and Evolutionary Computation Conference (GECCO), (1999) 525–532.
[23] T. A. Welch. A Technique for High-Performance Data Compression. IEEE Computer, 17(6)(1984) 8-19.
[24] N. Kunasol, W. Suwannik, P. Chongstitvatana. LZW-Encoding in Genetic Algorithm. Proceedings of Electrical Engineering Conference (EECON-28), (2005) 861-864.
[25] M. Pelikan. NK Landscapes, Problem Difficulty, and Hybrid Evolutionary Algorithms. GECCO'10, Portland, Oregon, USA, July, (2010) 665-672.
[26] R. Kindermann, J. L. Snell. Markov Random Fields and Their Applications. AMS, (1980).

Received: 22 September 2010. Accepted: 18 October 2010

Table.2. Experimental results for each testing function averaged over 30 independent runs. The bold numbers mean the optimal is found and the numbers in parentheses are the percentage of times that the optimal is found.

Problem	Size	BOA		DE		LZWDE		ACDE		ACDE-HC		ACDE-SA	
		Fit.	*Time (ms)*	*Fit.*	*Time (ms)*	*Fit.*	*Time (ms)*	*Fit.*	*Time (ms)*	*Fit.*	*Time (ms)*	*Fit.*	*Time (ms)*
RandomMax	100	**-100** (100.00%)	3,048.56	**-100** (100.00%)	788.03	-83.30	299.50	-95.67	238.57	**-100** (100.00%)	39.07	**-100.00** (100.00%)	137.00
	400	-386.4	266,602.73	**-400.00** (100.00%)	12,471.90	-249.20	5,648.40	-304.10	10,789.57	**-400** (100.00%)	7,666.67	-331.40	10,989.27
	800	-745.17	762,946.83	-794.80	40,025.53	-470.60	26,631.40	-534.60	74,905.20	-699.97	75,015.63	-586.40	78,150.50
RandomTrap	100	-82.9	4,750.10	**-100.00** (13.33%)	601.00	-61.73	278.63	**-100** (100.00%)	260.97	**-100** (100.00%)	128.67	-96.13	260.40
	400	-306.33	262,648.80	-300.07	11,330.73	-191.80	5,089.10	-291.67	10,733.27	-345.33	10,647.90	-251.67	11,039.10
	800	-584.83	1,261,020.90	-563.40	40,718.80	-355.00	16,990.60	-478.63	76,909.60	-604.73	76,315.57	-600.60	80,021.80
Ising Spin Glass	100	-128.80	4,896.80	-123.20	677.10	-66.00	509.97	-127.80	226.07	-128.27	232.30	-122.40	189.10
	400	-436.33	564,240.23	-361.40	13,692.20	-213.60	10,642.90	-392.87	10,777.13	-421.13	10,831.27	-505.53	9,965.23
	900	-827.53	5,466,826.60	-353.60	57,572.00	-306.00	41,632.00	-603.33	110,969.80	-942.53	112,141.63	-1,103.87	107,264.60
NK Landscape	100	-7,680.95	5,141.43	-7,692.11	830.73	-7,401.68	347.93	-7,818.18	265.10	-7,849.80	256.77	-7,798.36	309.40
	400	-29,738.21	616,745.00	-27,343.23	14,928.13	-29,176.47	5,626.60	-29,515.45	11,147.40	-30,900.34	11,131.77	-28,370.89	11,353.17
	800	-57,021.00	3,119,407.46	-52,781.78	53,118.80	-58,226.70	19,562.40	-56,452.41	78,113.90	-59,833.48	81,644.27	-55,425.38	82,170.30

Analysis of LZW Differential Evolution for Binary Encoding

Orawan Watchanupaporn and Worasait Suwannik

Abstract— **Differential Evolution (DE) is a fast and robust real vector optimizer. This paper applies DE to discrete problems by converting a real chromosome to an integer chromosome and then decompress to a binary chromosome using LZW algorithm. Experimental result shows that this approach is better than the previous work and the evolution time is very fast. Analysis result shows that the fitness landscape of LZW encoding is less complex than the original encoding for each test problem.**

Index Terms— **Differential Evolution, LZW, Discrete optimization, Fitness Landscape**

I. Introduction

DIFFERENTIAL Evolution (DE) is an evolutionary algorithm designed for solving real value optimization problems [1]. DE is very fast and efficient. It was ranked the third in the First International Contest on Evolutionary Optimization in 1996. However, it is more robust than those optimizers finished before [2]. In addition, DE is very compact. The core of the algorithm can be implemented in less than 20 lines of C code, which is available on-line [3].

DE performs very well in continuous optimization. However, for discrete optimization, there are a few works that investigate DE's effectiveness [4]. This paper presents two alternative methods for adapting DE for discrete optimization. The first method directly maps a real value chromosome to a binary chromosome. The second method combines compressed chromosome encoding with DE.

Compressed encoding enables evolutionary algorithm to solve very large problems [5][6][7]. For example, LZW encoding in Genetic Algorithm can solve one-million-bit problems. To use compression with GA, the individual is in a compressed form and has to be decompressed before the fitness evaluation. Another advantage of this approach is low memory requirement.

The motivation for using compress encoding is to reduce the size of the search space so that the solution can be found faster. However, in some cases, LZW encoding can solve problem faster even when the size of the search space is equal to the original encoding. This means the LZW encoding not only can reduce the search space, it also aid the evolutionary search process. While the effect of search space reduction can be measure easily by comparing the size of the search space, the effect of using LZW encoding is difficult to be explained. Another contribution of this paper is to analyze LZW compressed encoding.

We use the method proposed by Uludag and Uyar [8] to analyze the fitness landscape of DE. The idea of the analysis method is to random walk on the fitness landscape. A new step is obtained from the proposed neighborhood function which is suitable for DE. In addition, we define a new distance metric that is suitable for LZW encoding.

The organization of this paper is as follows. Section 2 describes DE. Section 3 explains how LZW compressed encoding is applied to DE. Section 4 explains benchmark problems. Section 5 describes the experiment. Section 6 discusses the result. Section 7 analyzes LZW encoding and its interaction with DE. Finally, Section 8 summarizes the paper.

Manuscript received March 20, 2013.

O. Watchanupaporn is with Department of Computer Science, Kasetsart University, Bangkok, Thailand (phone: +66-8-9148-6740; e-mail: orawan.liu@gmail.com).

W. Suwannik is with Department of Computer Science, Kasetsart University, Bangkok, Thailand (e-mail: worasait.suwannik@gmail.com).

II. Differential Evolution

Differential Evolution (DE) is an evolutionary optimization method. The first generation of real vectors is created by randomly filled the values in the vectors. Each vector has D values. A population consists of NP vectors. There are two schemes (i.e., DE1 and DE2) presented in [4]. In this paper, DE1 is used.

A new generation is created by the following method. Each vector competes with its trial vector. The one with less cost survives to the next generation. A trial vector is created by combining the vector with a mutant vector. The combination is similar to crossover in Genetic Algorithm [9]. A mutant vector is created by adding a random vector with a weight difference of other two random vectors (hence the name Differential Evolution). The mathematical formula for creating a mutant vector is as follows:

$$X'_c = X_c + F(X_a - X_b) \quad (1)$$

The parameters in DE are listed below.

- NP (or population size) should be 5-10 times the number of parameters D.
- F (i.e., the weight) should start with 0.5. F and NP should be increased if the algorithm converges prematurely.
- CR (or the crossover rate) should be 0.9, 0.1, or 0.

III. LZW Differential Evolution

Lempel-Ziv-Welch Algorithm (LZW) is a lossless dictionary-based data compression/decompression algorithm [10]. 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. In LZWDE, only decompression algorithm is used. The pseudo code of LZW decompression is given in Fig. 1.

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. Fig. 2 shows an example of decompressing an array of integer to a binary string. 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. The dictionary does not have to be stored because the algorithm can construct the dictionary during the compression or decompression process.

To use LZW compressed encoding with DE, we add a conversion and decompressing step before a fitness evaluation. The real value chromosome is converted to an array of integers. After that, the array is decompressed to a binary string. Because LZW cannot decompress arbitrary input, each code in an integer array must satisfy the following constraint [6].

$0 \le a_i \le i+1$, where i is a zero-based array index

Any positive integer can be changed to satisfy the constraint by modulo with $i+2$. An example of converting a real vector to a binary string is shown in Fig. 3.

Implementing an LZW chromosome encoding in object-oriented language is easy. The core algorithm does not have to be modified to support LZW encoding. Rather, for each benchmark problem, we implemented the interface for fitness evaluation using two classes: one for a normal chromosome and the other is for a compressed chromosome. For DE's point of view, it still evolves real vectors. It does not know that it is evolving compressed encoding chromosomes.

Note that LZWDE evolves a direct representation of an individual as a "compressed" string. There is no compression step involved in LZWDE.

IV. Benchmark Problems

We use 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 has 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 a simpler structure.

```
Algorithm LZW Decompress
    add entries 0 and 1 to the dictionary
    read one code from input to c
    output str(c)
    p = c
    while input are still left
        read one code from input to c
        if the code c is not in the dictionary
            add str(p) + fc(str(p)) to the dictionary
            output str(p) + fc(str(p))
        else
            add str(p) + fc(str(c)) to the dictionary
            output str(c)
        else if
        p = c
    end while

The variable c stores a code read from input.
The variable p is the previous value of c.
The function str(code) returns a string associated with code.
The function fc(string ) returns the first character in string.
```

Fig. 1. The pseudo code of LZW decompression.

Decode		Dictionary	
Input	Output	Index (c)	Full string
		Initial table	
		0	0
		1	1
		Start enter string to dictionary	
0	0	-	-
2	00	2	00
1	1	3	001
3	001	4	10
1	1	5	0011
1	1	6	11

Fig. 2. Example of decompressing an array of integers to a binary string.

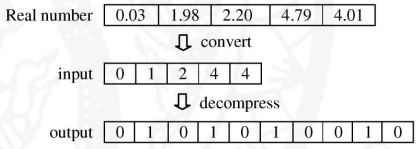


Fig. 3. Converting a real value chromosome to an integer chromosome and decompressing it to a binary chromosome.

In [4], the author applied DE to solve the following discrete optimization problems: OneMax, Royal Road, Order-3 Deceptive, and Long Path problems. We test the performance of our algorithm using the same benchmark. Every benchmark problem is a maximization problem. However, since DE is a global minimizer, the fitness is transformed by multiplying the cost function with −1.

A. OneMax Problem

The OneMax problem [11] (or bit counting) is a widely used problem for testing the performance of various genetic algorithms. Formally, this problem can be described as finding a string $X = \{x_1, x_2, \ldots, x_k\}$, where $x \in \{0,1\}$, that maximizes the following equation:

$$F(X) = \sum_{i=1}^{k} x_i \quad (2)$$

B. Royal Road Problem

Royal Road problem [12] is designed to investigate the role of GA crossover and building block hypothesis. The problem can be solved using GA which uses crossover. However, it is difficult for a hill climbing algorithm or GA with a single-bit mutation to solve the problem.

This function involves a set of schemas S and is defined as:

$$F(X) = \sum_{s \in S} c_s \sigma_s \quad (3)$$

where x is a bit string, each c_s is a value assigned to the schema s.

C. Deceptive Order-3 Problem

In Deceptive problem [13], an individual composes of several blocks. Each of the blocks is evaluated by a deceptive function. The deceptive 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. The order-3 deceptive function is defined as:

$$\begin{aligned} f(000) &= 28 \\ f(001) &= 26 \\ f(010) &= 22 \\ f(100) &= 14 \\ f(011) &= 0 \\ f(101) &= 0 \\ f(110) &= 0 \\ f(111) &= 30 \end{aligned}$$

The deceptive problem can be decomposed to several deceptive functions. The problem, denoted by f_m, is defined as:

$$f_m(K_1 \cdots K_m) = \sum_{i=1}^{m} f(K_i), K_i \in \{0,1\}^3 \quad (4)$$

D. Long Path Problem

Long Path problem [14] is a problem that can be solved by a hill-climbing algorithm. However, it is not practical to solve this problem using hill climbing algorithm. This is because climbing the hill (or the path) takes exponential time. Each point in the path is differed by one bit. The path is constructed such that is exponentially long. The height from the bottommost of the hill to the top is equal to:

$$HillHeight(l) = 3 \times 2^{\lfloor (l-1)/2 \rfloor} + l - 2 \quad (5)$$

where l is a chromosome length.

V. Experiment

We conducted the experiment to compare the performance of LZWDE with Gong and Tuson's binary adapted DE operators [4] and with simple real to binary conversion DE. The latter scheme, which is simply called DE, converts a real value to a binary using the rule ($X_i < 0.5$? 0 : 1)

Table I shows the experimental parameters. The length of an LZWDE chromosome is less than DE chromosome which are 1/5 of OneMax problem size, 1/4 of Royal Road problem size, 1/12 of Deceptive order-3 problem size, and about 1/3 of Long Path problem size. Before a fitness evaluation, the compressed chromosome is decoded and decompressed with LZW decompression algorithm. The length of the decompressed binary chromosome is varied depending on the code in the integer array. If the length is more than the size of the problem size, the excess bits are discarded. However, if the length is less than the problem size, LZWDE will evaluate the fitness of available bits. All experimental results are the average performance obtained from 30 runs.

TABLE I
Experimental Parameters

Parameter	OneMax	Royal Road	Deceptive order-3	Long Path
Population size	50	30	100	30
Problem size	500	80	300	29
LZW chromosome length	100	20	25	10
Maximum generation	500	500	2000	300

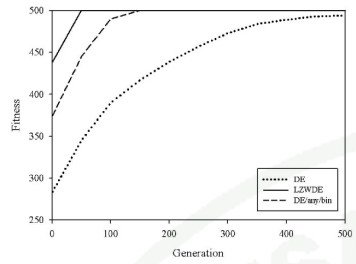
VI. Results

Gong and Tuson [4] used different sets of parameters for OneMax, Royal Road, Deceptive Order-3 and Long Path problems. They reported the result of 4 DE strategies which are: 1) any-change mutation and exponential crossover-DE/any/exp, 2) any-change mutation and binomial crossover-DE/any/bin, 3) restricted-change mutation and exponential crossover-DE/res/exp, and 4) restricted-change mutation and binomial crossover-DE/res/bin for each problem. We choose the best of their experimental results and compare them with our best parameters for each problem.

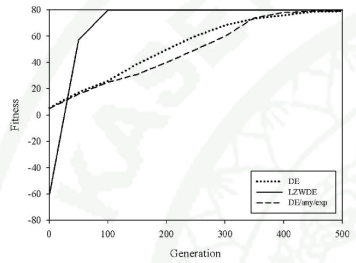
For each benchmark problem, we compare the performance of binary-adapted DE, DE (simple real to binary conversion), and LZWDE. The result is shown in Figure 4. The X-axis shows the number of generations and the Y-axis shows the average-best fitness. LZWDE outperforms both DE and binary-adapted DE. Moreover, it is interesting to see that the performance of simple conversion is comparable to binary-adapted DE in Royal Road problem and better than binary-adapted DE in Long Path problem.

Table II shows the average evolution time. We ran the experiment on Intel Core i5 with 4GB of RAM. In this table, we report only the time that DE successfully finds the solution. The number in the parenthesis is the success rate. LZWDE can find the solution for every run. We do not have the data for binary-adapted DE. Therefore, we only compare

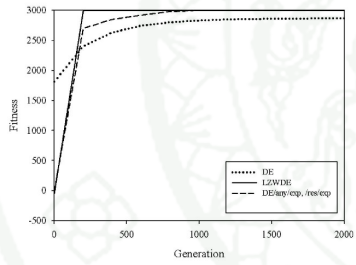
me of DE and LZWDE. In LZWDE, there is an LZW mpression step. Even with an additional step, the ithm can still find a solution faster than DE. Simple DE ot find a solution for Deceptive Order-3 problem.



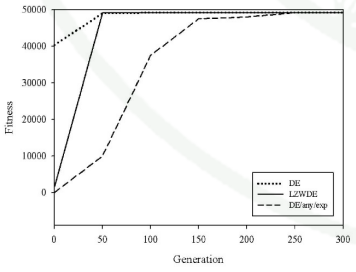
(a) OneMax



(b) Royal Road



(c) Deceptive order-3



(d) Long Path

Fig. 4. The average-best fitness plotted against generation

TABLE II
AVERAGE EVALUATION TIME (IN MILLISECONDS)

Problem	Algorithm			
	DE		*LZWDE*	
OneMax	388.43	(100)	6.50	(100)
Royal Road	36.42	(87)	29.10	(100)
Deceptive order-3	-	(0)	113.97	(100)
Long Path	13.53	(100)	45.69	(98.84)

VII. FITNESS LANDSCAPE ANALYSIS

In this section, we tried to explain why LZW encoding help improve the performance of DE.

A. *Binary Fitness Landscape*

The difficulty of a problem depends on two factors: the size of search space and the shape of fitness landscape. A problem with a larger search space is usually more difficult to solve. In addition, a problem with a more complex fitness landscape is more difficult. Example of complex fitness landscape is the one with many local minima or the one that leads evolutionary search away from the global minima.

To visualize the fitness landscape for binary optimization problem, we enumerate every possible chromosomes, evaluate their fitness and measure distance from the solution, then plot the graph using the fitness and the distance. Fig. 5(a) shows the fitness landscape of a 9-bit OneMax problem. The X-axis is the number of bits by which a chromosome differs from the solution. The Y-axis is the chromosome's fitness value. The darker area indicates a higher chromosome density. As shown in Fig. 5(a), as the fitness increases, the chromosome is closer to the OneMax solution. Since evolutionary algorithm use fitness value to guide a search process, OneMax is an easy problem because the fitness value can guide the search to the correct direction.

Fig. 5(c) shows the fitness landscape of a 9-bit Trap problem. The problem is more difficult to solve than OneMax because the fitness landscape deceives the search into moving away from the global optima. As the fitness increase, the chromosome is more different from the solution. If we try to solve the Trap problem using a local search which produces a neighbor with 1 bit different from the current position, the search will not be able to find the optimal solution.

Fig. 5(a) and (c) visualize the fitness landscape of two extreme. We can easily tell from the graph which problem is easier. However, for a problem with difficulty in between, a subjective judgment should not be used to judge the complexity of fitness landscape. Therefore, we quantify the shape of a fitness landscape as one single number called fitness-distance correlation (*fdc*). We compute *fdc* or a correlation between fitness and distance using the formula given below.

$$fdc = \frac{cov(F,D)}{\sigma(F)\sigma(D)} \tag{6}$$

where $cov(F,D)$ is a covariance of fitness F and distance D. $\sigma(F), \sigma(D)$ is a standard deviation of F and D respectively.

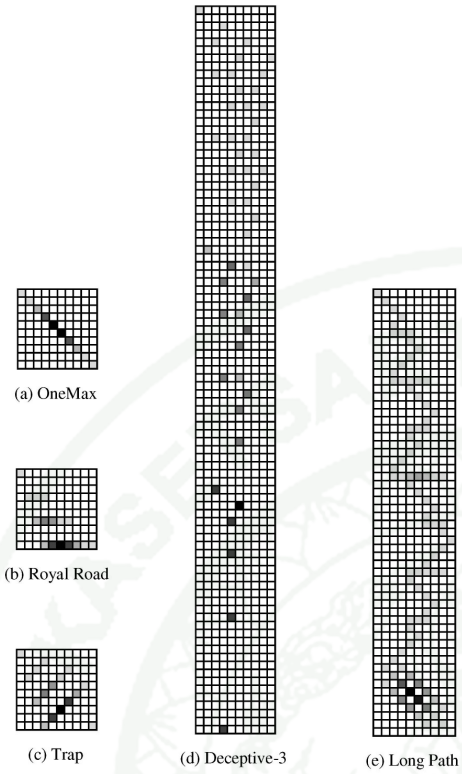


Fig. 5. The fitness landscape for binary optimization problems which are (a) OneMax, (b) Royal Road, (c) Trap, (d) Deceptive-3, and (e) Long Path. All problem sizes are 9-bit.

The *fdc* of the test problem is shown in Table III. For GA, OneMax's *fdc* is –1, which is the lowest. Deceptive problem has positive *fdc* which means that as the fitness increase, the chromosome is getting further from a solution.

TABLE III
THE FDC OF THE TEST PROBLEM

Problem	Algorithm		
	GA	***DE***	***LZWDE***
OneMax	-1.00	-0.63132	-0.78203
Royal Road	-0.65	-0.44755	-0.65961
Deceptive order-3	0.32	0.16866	-0.08296
Long Path	0.02	-0.00091	-0.07498

B. Real-value Fitness Landscape

Our paper use DE to solve binary problem. DE use real value vectors. The *fdc* cannot be calculated using the same method as in the previous subsection because of we cannot enumerate all possible real-value vectors as we enumerate all possible binary chromosome. For a binary optimization problem, there are finite amount of chromosomes given a fixed length binary string. A problem size n bit has 2^n possible chromosome. However, a single real-value in a DE vector, in theory, can have infinitely uncountable possible values.

Since we cannot enumerate all possible chromosomes, we instead explore the fitness landscape using random walk. While an analysis procedure performs random walk, it records a fitness and distance to a solution. Each step of random walk imitates a trial vector generation process in DE.

$$vector_{t+1} = vector_t + F(random\_unit\_vector) \quad (7)$$

In this paper, we set the value of F equals to 0.1 in order to make the step not too long. For each problem, an analysis procedure explores 100 random starting points. For each starting point, the procedure random walks for one million steps. A real value in the vector is constrained within the range [0, 1].

Another difference between binary and real value analysis is as follows. For a binary problem, we calculate a Hamming distance from a chromosome to an optimal solution. In DE, Euclidean distance is calculated. The distance calculation depends on how real-to-binary conversion is done. In this paper, the rule for converting is $X_i < 0.5 \; ? \; 0 : 1$. Therefore, if a one bit of binary solution is 1, and the corresponding real value is in the range [0.5, 1), the distance would be zero. Otherwise, the distance would be $0.5 – X_i$. If a binary solution is 0, the distance would be zero when the corresponding real value is in the range [0, 0.5). Otherwise, the distance would be $X_i – 0.5$.

Table III shows *fdc* for each problem. Real value *fdc* and binary *fdc* are different due to the way we measure the distance and perform the random walk.

C. LZW Real-value Fitness Landscape

Although both DE and LZWDE use real value vectors, the procedure to calculate the distance is different. In LZWDE, a real-value vector has to be converted to an array of integers before decompression and fitness evaluation. Thus, the distance calculation depends on how real to integer conversion is done. In this paper, conversion is done simply by truncating a fraction part of a real number. An example of measuring the distance is as follows. Suppose that one integer in a solution array is 3. If the corresponding real value Xi is in the range [3, 4), the distance would be zero. If X_i is less than 3, then the distance would be $3 – X_i$. Otherwise, the distance would be $X_i – 4$. To calculate a distance of a vector to a solution vector, the Euclidean distance formula is used.

The random walk process is similar to the previous subsection. The difference is that each real value X_i is constrained to the range [0, i+2). The value within this range can be converted to a valid input for LZW decompression algorithm.

For some problem such as OneMax, the original binary encoding has only one solution. However, when the problem is encoded with LZW, there might be more than one solution. For example, an LZW chromosome of length 4 has 2 solutions for 9-bit OneMax problem. In that case, the minimum distance from a vector to both solutions is used to compute *fdc*.

Table III shows *fdc* for each problem. For each test problems, LZW encoding has lower *fdc* than the original encoding. This explains why LZWDE performs better than DE.

VIII. Conclusion

This paper proposes two methods to apply DE to solve discrete optimization problem. The first is simple real-to-binary conversion. The second is using LZW encoding. We compared the result with binary-adapted DE using the same benchmark problems. The result shows that LZWDE outperforms binary-adapted DE and DE with simple real-to-binary conversion. In addition, in term of computation time, LZWDE is very fast even it has to decompress the chromosome. It can solve all benchmark problems in less than one second using a mid-range computer.

Using LZW can speed up evolutionary search because of reduction in search space and transformation of fitness landscape. The latter points are backed up by the analysis. This paper proposed two distance metrics, one for DE and another for LZWDE, to analyze simple real-to-binary conversion and LZW encoding. These metrics in used with a neighborhood function to compute fitness distance correlation (*fdc*). The result shows that, in the benchmark problems, LZW encoding can simplify the fitness landscape.

References

[1] R. Storn and K. Price, "Differential Evolution - A simple and efficient adaptive scheme for global optimization over continuous spaces," 1995.

[2] K. V. Price, R. M. Storn and J. A. Lampinen, "Differential Evolution: A Practical Approach to Global Optimization," Springer, 2005.

[3] K. Price and R. Storn, Differential Evolution, http://www.drdobbs.com/architecture-and-design/184410166, 1997.

[4] T. Gong and A. Tuson, "Differential Evolution for Binary Encoding," Soft Computing in Industrial Applications, vol. 39, pp. 251-262, 2007.

[5] 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.

[6] 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.

[7] W. Suwannik and P. Chongstitvatana, "Solving One-Billion Bit Noisy OneMax Problem using Estimation Distribution Algorithm with Arithmetic Coding," Proceedings of IEEE Congress on Evolutionary Computation (CEC 2008), 2008, pp. 1203-1206.

[8] G. Uludag and A. S. Uyar, "Fitness landscape analysis of differential evolution algorithms," Soft Computing, Computing with Words and Perceptions in System Analysis, Decision and Control (ICSCCW 2009), 2009, pp. 1-4.

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

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

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

[12] M. Mitchell, S. Forrest, and J. H. Holland, "The Royal Road for Genetic Algorithms: Fitness Landscapes and GA Performance," in Proc. The First European Conference on Artificial Life, Cambridge, MA, MIT Press, 1991, pp. 245-254.

[13] D. E. Goldberg, "Genetic algorithms and Walsh functions: Part I, a gentle introduction," Complex Systems, vol. 3, 1989, pp. 129-152.

[14] J. Horn, D. E. Goldberg, and K. Deb, "Long Path Problems," Lecture Notes in Computer Science, vol. 866, 1994, pp. 149-158.

Orawan Watchanupaporn is a Ph.D. student at the Department of Computer Science, Kasetsart University, Thailand. She obtained a bachelor's degree in Computer Science from Bangkok University in 2004 and master's degree from Kasetsart University in 2006. Her research interests include compressed compact genetic algorithms and Estimation of Distribution Algorithms.

Worasait Suwannik received a Ph.D. in computer engineering from Chulalongkorn University, Thailand, in 2006. At present, he is a lecturer at the Department of Computer Science, Kasetsart University, Bangkhen Campus, Thailand. His research interests include compressed genetic algorithms and GPU programming.

CURRICULUM VITAE

NAME : Miss. Orawan Watchanupaporn

BIRTH DATE : Dec 8, 1981

BIRTH PLACE : Rayong, Thailand

EDUCATION :

YEAR	INSTITUTE	DEGREE/DIPLOMA
2006	Bangkok Univ.	B.Sc.(Computer Science)
2009	Kasetsart Univ.	M.S.(Computer Science)

POSITION/TITLE : Lecturer

WORK PLACE : Faculty of Science Siracha, Kasetsart University Siracha Campus

SCHOLARSHIP/AWARDS : Thesis award from the Graduate School 2005 (Good quality)

PUBLICATION LISTS

1. Watchanupaporn, O. and W. Suwannik. 2010. An Estimation of Distribution Algorithm using the LZW Compression Algorithm, pp. 97-102. *In* Proceedings of the 2nd International Conference on Advanced Cognitive Technologies and Applications (COGNITIVE 2010), Lisbon, Portugal.

2. Watchanupaporn, O. and W. Suwannik. 2010. Conditional Probability Mutation in LZWGA, pp. 67-72. *In* Proceddings of the International ACM Conference on Management of Emergent Digital EcoSystems (MEDES'10), Bangkok, Thailand.

3. Watchanupaporn, O. and W. Suwannik. 2011. LZW Mutual-Information-Maximizing Input Clustering Algorithm, pp. 2273-2276. *In* Proceddings of the International Conference on Biomedical Engineering and Informatics (BMEI), Shanghai, China.

4. Watchanupaporn, O., W. Suwannik and P. Chongstitvatana. 2012. Mutation in Compressed Encoding in Estimation of Distribution Algorithm, pp. 308-311. *In* Proceedings of the International Conference on Genetic and Evolutionary Computing (ICGEC 2012), Japan.

5. Watchanupaporn, O. and W. Suwannik. 2012. LZW Differential Evolution for Binary Encoding, pp. 51-54. *In* Proceedings of the International Conference on Advanced Topics in Artificial Intelligence (ATAI 2012), Singapore.

6. Watchanupaporn, O. and W. Suwannik. 2012. Arithmetic Coding Differential Evolution for Binary Encoding. Advances in Information Technology and Applied Computing (AITAC) ISSN 2251-3418 1: 155-158.

7. Watchanupaporn, O. and W. Suwannik. Arithmetic Coding Differential Evolution with Local Search. Advanced Science Letter ISSN 1936-6612 (accepted for publication).

8. Watchanupaporn, O. and W. Suwannik. Analysis of LZW Differential Evolution for Binary Encoding. GSTF Journal on Computing ISSN 2251-3043 (accepted for publication).