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 next page.

B. LZWGA

The main difference between LZWGA and GA is that an LZWGA chromosome is in a compressed format. Therefore, the LZWGA chromosome has to be decompressed before its fitness can be evaluated. In [2], LZWGA is compared with the simple GA using the same parameters except the length of individuals (compressed vs no compression). For OneMax problem, by using the same amount of time, the best chromosome that simple GA can find is a little more than half of solution fitness (LZWGA can find a solution). LZWGA requires less memory and time to transfer data from one generation to the next generation. For example, to solve one-million-bit problem, each chromosome in LZWGA have 40,000 genes or 640,000 bits (40,000 × 16) but GA used 10^6 bits per each chromosome. LZWGA spends less time than GA during genetic operations (e.g., crossover, mutation, and reproduction). The pseudo code of LZWGA is shown below.

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

The variable Z is the population of compressed chromosome.

The variable P is the population of uncompressed binary chromosomes.

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
```

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*.

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+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 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 0th integer in the chromosome) is read and the algorithm output 1. In the first iteration, the algorithm reads "3" (the 1st integer), adds to dictionary the string 11 at the entry 2, and outputs 11. In the second iteration, the algorithm reads "3" (the 2nd 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+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 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. 1.

The main difference between LZWCGA and cGA are initializing and updating the probability matrix process. The sequence of LZWCGA process is shown below.

```
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
```

The first step in LZWCGA is to initialize the probability matrix. The pseudo code is shown below. 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
```

The variable *l* is length of an individual.

Then, we randomly generate two individuals *a* and *b* from the probability matrix using the pseudo code shown below.

```
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 ($r \leq interval$)
                $lzwChromosome[i] = j$
                break
            end if
        end for
    end for
```

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 the right column of this page.

By way of illustration, the initial probability matrix is shown in Fig. 1. The probability matrix after updating using values from *winner* and *loser* is shown in Fig. 2.

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 1. 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 2. 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 ($indexW \neq indexL$)
            if ($p[i][indexW] + (1/n) \geq 1.0$)
                $p[i][indexW] = 1.0$
                for $j$ = 1 to $i$+1 do
                    if ($j \neq indexW$)
                        $p[i][j] = 0.0$
                    end if
                end for
            else
                if ($p[i][indexL] - (1/n) \leq 0.0$)
                    $p[i][indexW]$ += $p[i][indexL]$
                    $p[i][indexL] = 0.0$;
                else
                    $p[i][indexW]$ += $(1/n)$
                    $p[i][indexL]$ -= $(1/n)$
                end if
            end if
        end if
    end for
```

The last step of LZWCGA 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:

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

B. Trap Problem

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

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

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:

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

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

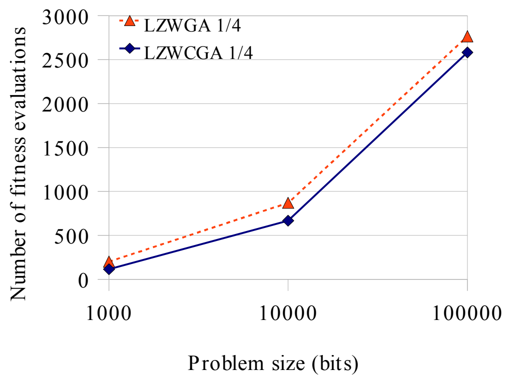
Parameter	Value
Population size	128, 512, 1024
Trap size	5
Total trap	100, 1000, 10000
Problem size (bits)	Trap size x Total trap
Chromosome compression ratio	1/4 of problem size
Max generation (for LZWGA)	500
Max round (for LZWCGA)	500 x population size
Crossover rate (for LZWGA)	1
Mutation rate (for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

V. Results and Discussion

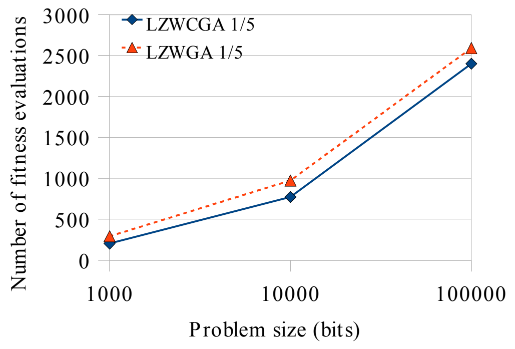
The experimental results show that LZWCGA outperforms LZWGA on both OneMax and Trap problems (see Fig. 3 and Fig. 4). We found that the bigger problem size needs more fitness evaluations. Moreover, higher compression ratio requires more fitness evaluations.

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.)

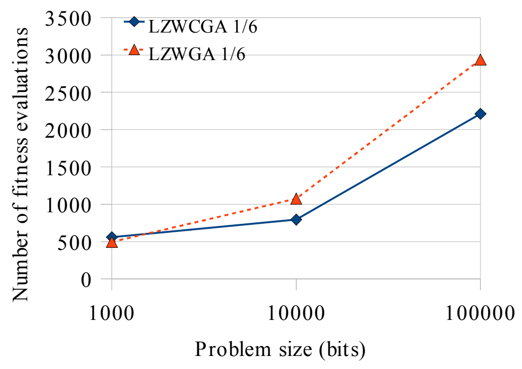
A visual representation for an LZWCGA probability matrix is shown in Fig. 5. The X-axis represents positions in a chromosome and the Y-axis represents probability that a value can occur in that position. The darker area indicates higher probability. The initial probability matrix is shown in the first 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 3. The number of fitness evaluations of LZWCGA and LZWGA when solving various sizes of OneMax problem.

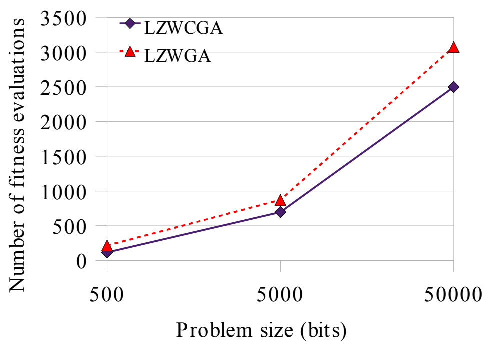


Figure 4. The number of fitness evaluations when using LZWCGA and LZWGA to solve Trap problem. The compression ratio is 1/4.

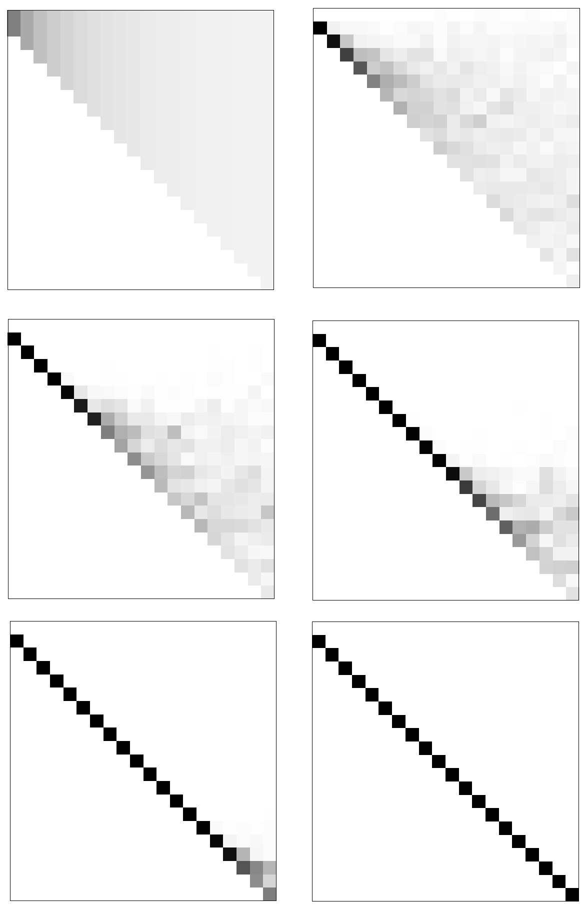


Figure 5. A visual representation for a probability matrix at 0, 10000, 20000, 30000, 40000, and 50000 fitness evaluations

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 makes the algorithm find the solution more effectively. We found that the LZWCGA's performance is comparable to LZWGA on OneMax and Trap problem. This result is promising because we think that if LZW encoding is integrated with more advanced EDAs, the performance of the new algorithm might be better than the original LZWGA. 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.

Acknowledgements

This work was supported by Faculty of Science's Budget for Overseas Academic Conferences (BOAC) and also supported by Department of Computer Science, Kasetsart University.

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 Conference (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.