Solving One-Million-Bit Problems using LZWGA

Naris Kunasol[*], Worasait Suwannik[*], and Prabhas Chongstitvatana[†]
[*] Faculty of Science
Kasetsart University, Bangkok, Thailand
Tel: 66-2-942-8026; fax: 66-2-942-8488
E-mail: g4764228@ku.ac.th and worasait.s@ku.ac.th
[†]Chulalongkorn University, Bangkok, Thailand
Tel: 66-2-218-6983, E-mail: prabhas@chula.ac.th

Abstract— **To solve a problem using Genetic Algorithm (GA), a 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. To reduce the search space, one approach is to use a compressed encoding chromosome. This work proposes LZWGA that used compressed chromosomes. An LZWGA chromosome has to be decompressed using an LZW decompression algorithm before its fitness can be evaluated. The paper reports how to solve one-million-bit OneMax, Royal Road and Trap functions using LZWGA.**

The search space of the original problem is $2^{1000000}$ or about 9.90×10^{301029} points. When using a compressed encoding, the search space was reduced to 8.37×10^{166717} points. The result from the experiment shows that it is practical to solve the problem of this size with the proposed method.

I. Introduction

GENETIC Algorithm (GA) is an algorithm inspired by natural evolution Ref. [1]. To solve a problem using GA, a candidate solution is encoded as a chromosome. For a binary encoding, a chromosome is encoded as a fixed length binary string. GA searches in the space of this representation. The length of a chromosome is related to the size of the search space, for n-bit chromosomes the search space is 2^n points. When n is large, the computational time becomes very long.

There are some approaches to reduce a search space. One approach is to apply a heuristic in an evolution process. In Ref. [2], 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 by using compressed encoding. In Compressed GA Ref. [3], a chromosome is represented using a compressed encoding format similar to run-length encoding. The result shows that Compressed GA uses 805 times less fitness evaluations than Simple GA when solving 128-bit OneMax problem.

To use Compressed GA, an appropriate number of bits of the repetition times (the run length) has to be specified. If the number of bits is too low or too high the effectiveness of compression is suffered. To overcome this problem, Kunasol et. al. Ref. [4] proposed LZWGA that used a compressed encoding that can be decompressed using Lempel-Ziv-Welch (LZW) decompression algorithm. The result shows that LZWGA outperforms Compressed GA for 2048-bit OneMax problem.

This paper extends the preliminary work in Ref. [4] by investigating LZWGA for a very large problem size. We present how LZWGA is used to solve one-million-bit OneMax, Royal Road, and Trap problems. The one-million-bit problem has an enormous search space. The search space of this problem is $2^{1000000}$ or 9.90×10^{301029} points. Solving the problem of this size using any canonical GA is not practical. Using LZWGA, the search space is reduced dramatically. Therefore, the problem can be solved in reasonable amount of time.

The organization of this paper is as follows. Section 2 gives an overview of LZW algorithm and describes LZW decompression algorithm. Section 3 explains LZWGA. Section 4 explains the experiment on solving one-million-bit OneMax, Royal Road, and Trap functions. Section 5 provides a discussion of the results. Finally, Section 6 concludes the paper.

II. Lempel-Ziv-Welch (LZW) Algorithm

The LZW is a lossless data compression algorithm Ref. [5]. The compression algorithm starts with a dictionary containing all characters. 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.

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. When using LZW to decompress an English text, the dictionary is initialized with all English characters and symbols. However, in this paper, the output of the decompression algorithm is a binary string. Therefore, the dictionary is initialized with the number 0 and 1.

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

```
Algorithm LZW Decompress
1:      add 0 and 1 to the dictionary
2:      read one code from input to c
3:      output str(c)
4:      p←c
5:      while input are still left
6:          read one code from input to c
7:          if the code c is not in the dictionary
8:                  add str(p)+fc(str(p)) to the dictionary
9:                  output str(p)+fc(str(p))
10:         else
11:                 add str(p)+fc(str(c)) to the dictionary
12:                 output str(p)
13:         end if
14:         p←c
15:     end while
```

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

Fig. 1. LZW Decompress pseudo code

III. LZWGA

The main difference between LZWGA and Simple GA is that a chromosome is in a compressed format. 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 LZWGA
1:      Z←create_first_generation ()
2:      repeat
3:          P←decompress(Z)
4:          evaluate(P)
5:          Z←create_next_generation(Z)
6:      until is_terminate()
```

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

Fig. 2. LZWGA pseudo code

A. 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. 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 ith integer of a chromosome must not have value greater than i+1. The first integer of the chromosome is either 0 or 1 because during the decompression, the dictionary is firstly initialized with 0 and 1.

For example, an LZWGA chromosome that can be successfully decompressed is (1,2,3). The decompression algorithm will output a binary string 111111. A dictionary has the entries ((0,0), (1,1), (2,11) (3,111)). Another valid chromosome is (0,1,2). The decompression algorithm will output a binary string 0101. A dictionary has the entries ((0,0), (1,1), (2,01) (3,10)).

If the ith 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 digit 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 ith integer, a random non-negative integer is modulo with i+1.

B. 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 length of the output chromosome is varied. If the length is more than the size of the solution encoding, the excess bits are discarded. If the length is less than the size of the solution encoding, the remaining bits are filled with 0's. After decompression, the decompressed binary string is evaluated. A fitness of a compressed chromosome is equals to the fitness of the decompressed chromosome.

C. 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 satisfied the constraint.

IV. Solving One-Million-Bit Problems

We solved one-million-bit OneMax, Royal Road, and Trap problems using LZWGA with the parameters shown in Table I. One of the most important parameters in LZWGA is the length of chromosome. This is because it affects the size of the search space. The size of the search space of n integers LZWGA chromosome is equals to (n+1)! In our experiment, a compressed chromosome has the length of 40,000 integers. The size of the search space is 40001! or 8.37×10^{166717} points. The number is enormous but it is much less than the size of the search space of a canonical GA chromosome for the same problem, which is $2^{1000000}$ or 9.90×10^{301029} points.

The algorithm is implemented in Java language. It was compiled and run using JDK 1.5 on Pentium 4HT 3GHz with 1 GByte of RAM. The operating system is Windows XP Professional. We repeated the experiment for 30 times and reported the average result.

A. *OneMax*

OneMax or a bit counting problem is a widely used problem for testing the performance of various genetic algorithms. The problem is formally defined as follows.

$$F_k(b_1 \ldots b_k) = \sum_{i=1}^{k} b_i \qquad (1)$$

where b_i is in {0,1}.

Fig. 3 shows the fitness curve of LZWGA in solving OneMax problem. On average, LZWGA can solve the one-million-bit OneMax in 154 generations or 15,400 fitness evaluations. LZWGA can find solution in about 18 minutes.

The result of LZWGA is compared with the result from Simple GA using the same parameters except the length of individual is one million bits (no compression). Within the same amount of time, the best chromosome that Simple GA can find has a fitness value of 506,540, or a little more than half of solution fitness.

Simple GA starts with a population with higher average fitness but progress much more slowly than LZWGA (in terms of CPU time). The first generation of Simple GA already has an average fitness about half million. This is because the population is the first generation is uniformly random binary strings. In contrast, the average fitness of the best chromosome in the first generation of LZWGA has an average fitness of 428,933.

In OneMax problem LZWGA save memory and time to transfer data from chromosomes' parent to chromosomes of next generation. The chromosomes of LZWGA have 40,000 genes or 640,000 bits (40,000 x 16) but GA used $1\text{x}10^6$ bits per chromosome (bits 0 or 1). Thus GA used memory and time more than LZWGA. The memory usages and time of LZWGA have effective in the genetic operation process (crossover, mutation and reproduction).

TABLE I
PARAMETERS OF LZWGA

Parameter	Value
Population size	100
Individual length	40,000
Selection method	Tournament (size=4)
Crossover rate	80%
Crossover method	Single point
Mutation rate	5%
Number of best individual to keep	10
Maximum generations	500

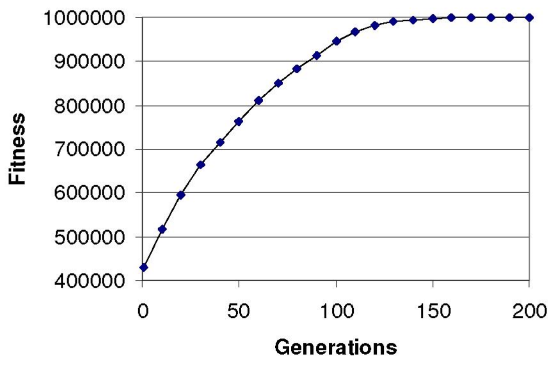

Fig. 3. Fitness curve of LZWGA and GA in solving one-million-bit OneMax problem.

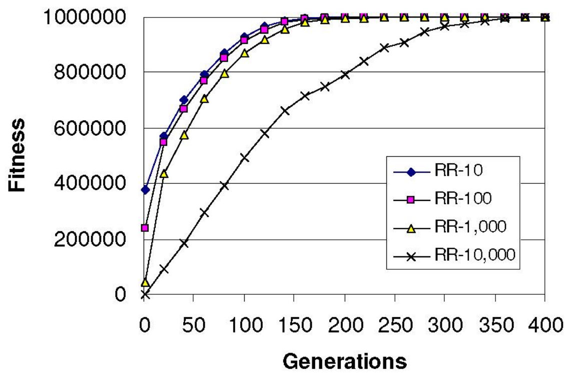

Fig. 4. Fitness curves of LZWGA in solving one-million-bit Royal Road functions

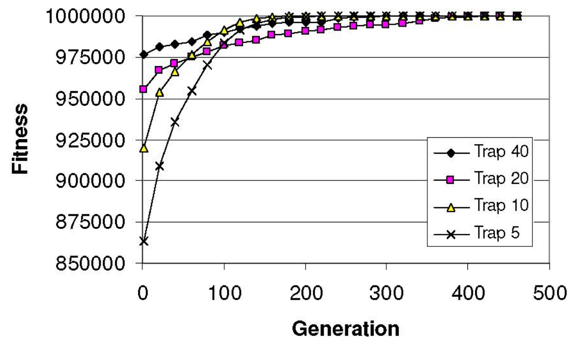

Fig. 5. Fitness curves of LZWGA in solving one-million-bit Trap functions.

B. *Royal Road*

A simple Royal Road functions Ref. [6] denoted by *R* are defined as:

$$R(x) = \sum_i c_i \delta_i(x), \quad \text{where } \delta_i(x) = \begin{cases} 1 \text{ if } x \in s_i \\ 0 \text{ otherwise.} \end{cases} \tag{2}$$

For a problem with block size *k*, s_i is a schema that have 1 defined in the range $i\times k$ to $((i+1)\times k)-1$. All other positions contain a wild card '*'. Each schema s_i is given with a coefficient c_i.

We conducted the experiment on various block sizes: 10; 100; 1,000; and 10,000. The fitness curves of LZWGA experiment on solving the Royal Road problem are showed in Fig 4. In our preliminary experiment, we tried to solve the problem with various small block sizes such as 5, 10, and 20. However, the results are very close to each other. Therefore, we varied the block sizes so that they differ in the order. The larger block size resulted in lower fitness in the early generations. For block size 10,000, the average fitness of the best chromosome over 30 runs is just 1,200. From 30 runs, LZWGA cannot find solution for 1 and 5 runs for the block size of 1,000 and 10,000 respectively.

C. *Trap*

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

$$F_k(b_1 \ldots b_k) = \begin{cases} f_{\text{high}} & ; \text{if } u=k \\ f_{\text{low}}-(u\times f_{\text{low}})/(k-1) & ; \text{otherwise} \end{cases} \tag{4}$$

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:

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

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.

We performed the experiment with Trap functions of various trap size: 5, 10, 20 and 40 bits. The chromosome length is one million bits. The fitness curves of LZWGA experiments on solving trap functions are shown Fig 5.

The success rate and the average generation for a successful run are shown in Table II. The increasing of trap size reduces the probability of LZWGA in finding the solution. The trap size 5 uses more generations than trap size 10 to convert solution and the trap size 20 use more generation than trap size 40 too. Why small block sizes have more generation than larger block size? LZWGA's algorithm can be found solution although it was trapped.

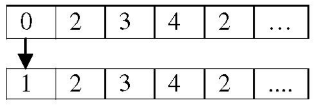

Fig. 6 LZWGA's chromosome mutate can change number 0 in first gene to 1

Fig. 6 show the chromosome that was trapped in the trap problem. When it decompressed the decompress chromosome is 000000000.... GA must crossover and mutate bits 0 become 1 but it is so difficult to change bits 0 to 1 at all. Instead of LZWGA can change number 0 in first gene be 1. When it decompressed the chromosome was 111111111.... Thus LZWGA can be easier than GA to solve this problem. Table III shows an interesting behavior of LZWGA. When the solution is not found, the maximum size of binary string in the dictionary is 22 bits. Even at this maximum length, an uncompressed chromosome is still shorter than one-million bits (i.e., 22×40,000 = 880,000). It can be interpreted that the deceived chromosome, or the one that converges to all zeroes, has short length representative.

TABLE II
SUCCESS RATE AND AVERAGE GENERATION IN SOLVING TRAP PROBLEM OF VARIOUS SIZES

Trapsize	Success	Generation
5	100%	158
10	90%	148
20	73%	214
40	27%	191

TABLE III
AVERAGE AND THE MAXIMUM SIZE OF STRINGS IN LZW DICTIONARY AFTER DECOMPRESSING THE BEST CHROMOSOME

Trapsize	Solution Found		Solution Not Found	
	Average	Maximum	Average	Maximum
5	26	43	-	-
10	26	43	10	22
20	26	47	10	22
40	26	44	10	22

V. Discussion

LZWGA requires a decompression step before the fitness evaluation. However, it adds only a linear time factor to the complexity of the algorithm. LZW Decompress has the time complexity of O(n), where n is the length of uncompressed string. Moreover, if the complexity of fitness evaluation for a particular problem is higher than linear, the decompression time will not be a dominant factor.

In terms of CPU time, a single iteration of LZWGA is much faster than Simple GA. The main factor is the size of chromosome. For very long chromosomes, it is likely that the computational time is dominated by the access to large size data structure which renders cache memory ineffective. For example, the time spent in creating the next generation in GA with very long chromosome is much longer than in LZWGA with compressed chromosome.

The search space of LZWGA is much smaller than that of GA. However, if the search space is too small, a solution might not exist in this space. For example, if the length of the chromosome is 10 digits, the chromosome cannot be decompressed into one million 1's. In general, if we set the LZWGA chromosome length too low, it will not be able to find a solution. On the contrary, if we set the length too high, the search space will become larger and it will likely take longer to solve a problem.

VI. Conclusion and Future Work

We used LZWGA to solve one-million-bit OneMax, Royal Road, and Trap functions. LZWGA is similar to Simple GA except that the chromosome is in compressed format. Before an LZWGA chromosome is evaluated, it has to be decompressed using LZW decompression algorithm. Our experiment used 40,000 digits of LZWGA chromosomes. LZWGA can solve one-million-bit OneMax problem using about 15,400 fitness evaluations or in about 18 minutes. It can also solve one-million-bit Royal Road with block size 10,000 and one-million-bit Trap function with block size 40.

OneMax problem and Trap functions have a high regularity solution. It would be interesting to investigate how well LZWGA solves other problems that the solution is not all 1's or all 0's.

References

[1] M. Mitchell, *Introduction to Genetic Algorithm*, MIT-Press, 1998.

[2] S. Chen and S. Smith, "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.

[3] W. Suwannik, N. Kunasol, P. Chongstitvatana, "Compressed Genetic Algorithm," Proceedings of Northeastern Computer Science and Engineering Conference, pages 203-211, March 31-April 1, 2005. (abstract in English)

[4] N. Kunasol, W. Suwannik, P. Chongstitvatana, "LZW-Encoding in Genetic Algorithm," Proceedings of Electrical Engineering Conference (EECON-28), pages 861-864, October 20-21, 2005. (abstract in English)

[5] A. Drozdek, *Data Structure and Algorithms in Java*, Brooks/Cole, USA, 2001.

[6] M. Mitchell, J. Holland, S. Forrest, "When Will a Genetic Algorithm Outperform Hill Climbing?," Advances in Neural Information Processing Systems, vol. 6, pages 51-58, 1994.