Solving Large Scale Problems using Estimation Distribution Algorithm with Arithmetic Coding

Worasait Suwannik[*], and Prabhas Chongstitvatana[†]
[*] Department of Computer Science
Kasetsart University, Bangkok, Thailand
Tel: 66-2562-5555; E-mail: worasait.suwannik@gmail.com
[†] Department of Computer Engineering
Chulalongkorn University, Bangkok, Thailand
Tel: 66-2218-6983, E-mail: prabhas@chula.ac.th

Abstract— **This work proposes an algorithm which combines Estimation Distribution Algorithm with a chromosome compression scheme to solve large scale problems. The search space reduction resulted from chromosome compression enables the proposed algorithm to solve one-million-bit problems and a one-billion-bit problem. Arithmetic Coding represents a compressed binary string with two real numbers. Using this representation, a model of highly fit individuals can be constructed. This model can be used to evolve the solution in the manner of Estimation Distribution Algorithm. The proposed algorithm is applied to large scale problems which are one-million-bit OneMax, Royal Road, Trap functions. It is also applied to one-billion-bit OneMax problem. The experimental result shows that the proposed algorithm can solve million-bit OneMax problem in 4 seconds and billion-bit OneMax problem in 92 minutes using a normal PC-class computer.**

I. INTRODUCTION

RECENTLY in genetic algorithm research, there has been a growing interest in solving very large scale problems. Kunasol et. al. solved one-million-bit genetic algorithm benchmark problems using genetic algorithm with LZW compression algorithm encoding (LZWGA) [1]. Sastry et. al. presented a parallel compact genetic algorithm to solve billion-bit problems [2]. Both methods attack very large scale problems from different points of view. The first method solves the problems by search space reduction. The second method solves the problem by parallelization and population modeling.

There are varieties of way to reduce the search space. For example, applying heuristic to a genetic operator [3], compressing introns [4], or using compressed encoding. In Compressed GA [5], a chromosome is in a compressed encoding format similar to run-length encoding. The experimental 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. The run length affects the performance of the algorithm. To overcome this problem, Kunasol et. al. [6] proposed LZWGA that used a compressed encoding that can be decompressed using Lempel-Ziv-Welch (LZW) decompression algorithm. LZWGA can solve one-million-bit OneMax problem in 18 minutes on average [1].

Both Compressed GA and LZWGA have the following disadvantages. First, we have to determine the appropriate length of the compressed chromosome. For the same problem, the length of a compressed chromosome is normally shorter than the length of a GA chromosome. However, if the length is too small, the algorithm will not find a solution. If it is too large, the algorithm might take longer time to run. Second, the length of the decompressed chromosome depends on an instance of a compressed chromosome and a decompression algorithm. It is likely that the decompressed chromosome is shorter or longer than that is required to solve a problem. If it is longer, we simply cut the excess. If it is shorter, we have to systematically fill the chromosome with either 0 or 1. This heuristic may not be suitable for all problems. Third, it is necessary to determine many parameters that are appropriate for various genetic operators such as a crossover rate and a mutation rate.

This paper proposed another form of search space reduction using Arithmetic Coding. Arithmetic Coding is a lossless compression technique. It represents a binary string with two real numbers. Moreover, these numbers are modeled by assuming bivariate normal distribution. Therefore, the proposed algorithm combines Estimation of Distribution Algorithm (EDA) [7] with Arithmetic Coding. The combination of EDA with Arithmetic Coding (abbreviated as EDAAC) alleviates most of the disadvantages of Compressed GA and LZWGA. To demonstrate its scalability, EDAAC is applied to solve several large scale test problems of the size one million bits and upto one billion bits.

The organization of this paper is as follows. Section 2 gives an overview of Arithmetic Coding and describes Arithmetic Coding Decompression algorithm. Section 3 explains EDAAC. Section 4 explains the experiment on

1-4244-0977-2/07/$25.00

solving one-million-bit OneMax, Royal Road, and Trap functions. Section 5 explains the experiment on solving one-billion-bit OneMax function. Section 6 provides a discussion of the compressed encoding. Finally, Section 7 concludes the paper.

II. Arithmetic Coding

Arithmetic coding is a lossless data compression algorithm [8]. The 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 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 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.

Next, the algorithm partitions the first interval [0.4, 0.64) into 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 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 process is summarized in Table I.

EDAAC required only a decompression algorithm. A pseudo code for Arithmetic Coding decompression used in EDAAC is shown in Figure 2. The algorithm runs in O(l) time, where l is the number of bits to be produced.

III. Estimation Distribution Algorithm with Arithmetic Coding

The pseudo code of EDAAC is shown in Figure 3. The algorithm begins by creating the first generation of compressed chromosomes, P. The compressed chromosome is decompressed using Arithmetic Coding Decompression algorithm. Then, the fitness of an uncompressed chromosome is evaluated. After all chromosomes are evaluated, highly fit chromosome are selected (Q) and modeled (Π). The model Π is then used to generate offspring R, which will be decompressed and evaluated. The algorithm repeats the process of selecting and modeling highly fit individuals, and generating offspring until the termination criterion is met. The algorithm terminates when a solution is found or a maximum generation is reached.

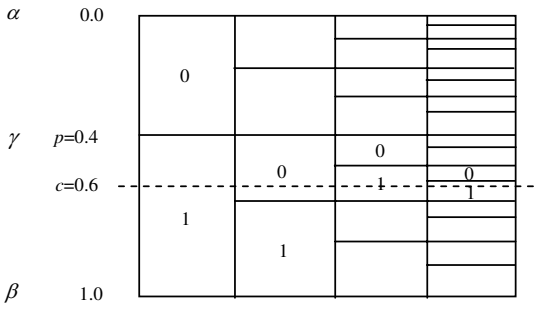

Figure 1. Decompressing 4 bits from (p, c)=(0.4,0.6). The output is 1011.

TABLE I
Interval and output for each iteration

α	γ	β	Output
0.000000	0.400000	1.000000	1
0.400000	0.640000	1.000000	0
0.400000	0.496000	0.640000	1
0.496000	0.553600	0.640000	1

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

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

Figure 2. Arithmetic Coding Decompression pseudo code

```
Algorithm: EDAAC
Π is the model
input:    P, Q, R are population
output:   the best individual in P
1:       create n individuals as the first generation, P
2:       decompress and evaluate all individuals
3:       while not terminate
4:           select n/2 individuals, P → Q
5:           model selected individuals, Π
6:           generate n/2 offspring, Π → R
7:           decompress and evaluate all offspring
8:           integrate population, Q ∪ R → P
9:       end while
```

Figure 3. EDAAC pseudo code

A. *Creating the First Generation*

Unlike a conventional GA, a chromosome in EDAAC is encoded as two double precision variables (64 bits). The first variable is p and the second variable is c. The value for p and c of the first generation chromosome are uniformly random from the range [0, 1).

B. *Decompressing and Evaluating Individuals*

Because the chromosome in EDAAC is compressed, it has to be decompressed to a binary string before its fitness can be evaluated. A compressed chromosome is decompressed using Arithmetic Coding Decompression algorithm. The de-compression stops when it outputs a binary string with the length equals to that of the problem size. For example, for 100-bit OneMax problem, the decompression stops when the algorithm outputs a 100-bit binary string. However, some value of p and c cannot be decompressed to the desired number of bits. In that case, the remaining bits are filled with 0's.

C. *Modeling Highly Fit Individuals*

Highly fit chromosomes are selected using any traditional genetic algorithm selection method. The algorithm selects $n/2$ individuals and computes μ_p, μ_c, σ_p, σ_c, and ρ of those individuals, where

- μ_p, μ_c are the means of p's and c's of the selected individuals,
- σ_p, σ_c are the standard deviation of p's and c's of the selected individuals,
- ρ is the correlation coefficient of p's and c's of the selected individuals.

D. *Generating Offsprings*

An offspring (p_g, c_g) is generated by sampling from the model Π. p_g is a normally distributed random variable with the mean μ_p and the standard deviation σ_p. c_g is a normally distributed random variable with the mean μ_g and the standard deviation σ_c. μ_g is obtained from the following formula.

$$\mu_g = \mu_c + \rho \frac{\sigma_c}{\sigma_p}(p_g - \mu_p) \qquad (1)$$

$n/2$ offspring are generated, evaluated, and integrated with $n/2$ previously selected individuals. These n individuals will be the population of the next generation.

IV. Solving One-Million-Bit Problems

One-million-bit well known problems in genetic algorithm literature are used to test the proposed algorithm. Those problems are OneMax, Royal Road, and Trap problems. EDAAC was executed with the parameters shown in Table II. The population size is 200 individuals. Please notice that this size is very small considering the size of the problems.

The algorithm is implemented in Java language. It was compiled and run using JDK 1.6 on Pentium 4 HT 3GHz with 1 GByte of RAM. The program required a great deal of memory especially in solving the one-billion-bit problem. A one-billion-bit chromosome required 125,000,000 bytes. In order to allocate such amount of memory, we run the Java program with the option –Xmx1000m to set the maximum heap size to 1000 MB. The experiment is repeated for 30 times and the average figures are reported.

A. *OneMax*

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

$$F_k(b_1 \ldots b_k) = \sum_{i=1}^{k} b_i \qquad (2)$$

where b_i is in {0,1}.

Figure 4 shows the fitness curve of EDAAC in solving OneMax problem. On average, EDAAC can solve the one-million-bit OneMax in 12.4 generations or 2,480 fitness evaluations. The number of fitness evaluation is much smaller than the theoretical prediction of using compact GA to solve OneMax [2]. The theoretical prediction is $\Theta(l \log l)$ fitness evaluations. The number of fitness evaluation is much smaller than that of LZWGA. LZWGA solves the same problem using 15,400 fitness evaluations [1].

EDAAC can find a solution in about 3.4 seconds while LZWGA can solve the same problem in about 18 minutes [1].

TABLE II
PARAMETERS OF EDAAC

Parameter	Value
Population size	200
Selection method	Tournament (size=4)
Number of best individual to keep	1
Maximum generations	500

B. *Royal Road*

A simple Royal Road functions [9] denoted by R are defined as:

$$R(x) = \sum_i c_i \, \delta_i(x), \text{ where } \delta_i(x) = \begin{cases} 1 \text{ if } x \in s_i \\ 0 \text{ otherwise.} \end{cases} \qquad (3)$$

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.

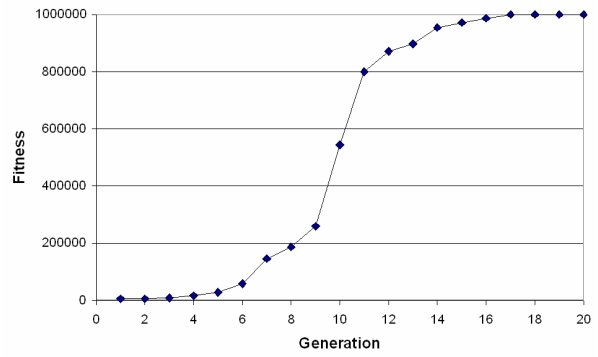

Figure 4. Fitness curve of EDAAC in solving one-million-bit OneMax problem.

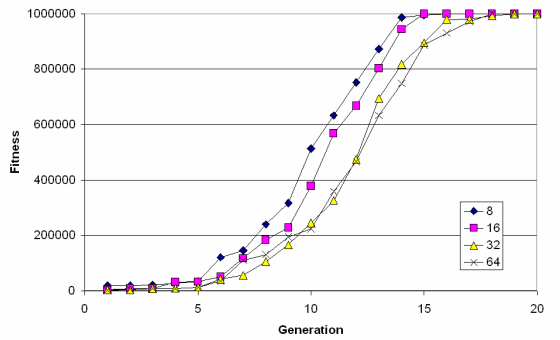

Figure 5. Fitness curves of EDAAC in solving one-million-bit Royal Road functions with various block sizes ranging from 8 to 64 bits.

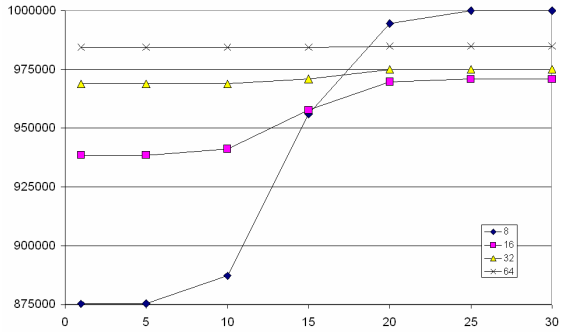

Figure 6. Fitness curves of EDAAC in solving one-million-bit Trap function with various trap sizes ranging from 8 to 64 bits.

.

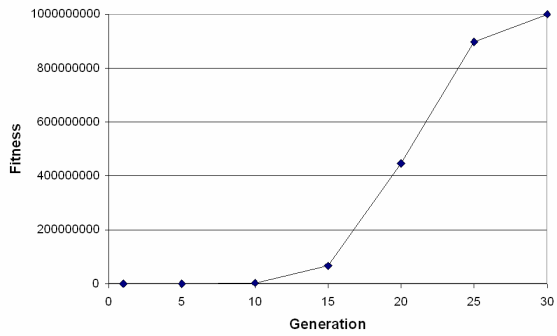

Figure 7 Fitness curve of EDAAC in solving one-billion-bit OneMax problem.

We conducted the experiment on various block sizes: 8, 16, 32, and 64. The fitness curves of EDAAC experiment on solving the Royal Road problem are shown in Figure 5. The percentage of successful run, the execution time, and the number of generation that EDAAC used to solve Royal Road problems shows in Table III. EDAAC can find a solution for all runs in about 13-15 generations.

C. *Trap*

The general k-bit trap functions are defined as:

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

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:

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

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 sizes: 8, 16, 32, and 64 bits. The chromosome length is one million bits. The fitness curves of EDAAC experiments on solving trap functions are shown Figure 6. The success rate, running time, and the average generations for successful runs are shown in Table IV. Larger trap size reduces the success rate and takes longer time for the problem to be solved. Notice that the algorithm will terminate after the 500$^{\text{th}}$ generation. However, from 30 runs, the maximum generation of successful run is only 28 generations.

TABLE III
SUCCESS RATE AND AVERAGE GENERATION IN SOLVING ROYAL ROAD PROBLEM OF VARIOUS SIZES IN MILLION BITS PROBLEM

Block size	Success	Time (seconds)	Generation
8	100%	5.6	13.0
16	100%	4.6	13.2
32	100%	4.6	14.4
64	100%	6.0	15.1

TABLE IV
SUCCESS RATE AND AVERAGE GENERATION IN SOLVING TRAP PROBLEM OF VARIOUS SIZES IN MILLION BITS PROBLEM. THE TIME AND THE GENERATION ARE AVERAGED FROM SUCCESSFUL RUNS.

Trap size	Success	Time (seconds)	Generation
8	100%	9.9	16.9
16	53%	14.1	18.9
32	20%	16.2	19.3
64	3%	11.7	18.0

V. SOLVING ONE-BILLION-BIT ONEMAX PROBLEM

Sastry et. al. implemented parallel Compact Genetic Algorithm to solve one-billion-bit Noisy OneMax problem [2]. They conduct the experiment on 128- and 256- processor partitions of 1280-processor cluster.

We conduct the experiment on one-billion-bit problem using a normal PC-class machine (the same machine used in one-million-bit problem). The memory requirement for a one-billion-bit chromosome is 125,000,000 bytes. In fact, to solve OneMax problem using EDAAC, it is not necessary to allocate such large amount of memory. Arithmetic Coding Decompression can easily be modified to be a stream decompression. With a stream decompression (on-line one-bit-at-a-time), very small amount of memory can be used to evaluate a one-billion-bit OneMax chromosome. However, in this paper, we did not use the stream decompression. We allocate the amount of data that can store a one-billion-bit chromosome to get a better understanding of the execution time of large scale problems.

EDAAC can find a solution to one-billion-bit OneMax in about 22.5 generations on average. The program can solve the problem in 92.2 minutes using one PC-class computer. Figure 7 shows the fitness curve of EDAAC in solving the problem.

VI. DISCUSSION

In theory, Arithmetic Coding can compress a binary string of any length using two real numbers without losing any information. This is because we can partition the interval forever. However, in an actual implementation, a real number is represented by a limited number of bits. Therefore, those numbers cannot produce binary string of arbitrary length.

We conduct an experiment to measure the length of binary string that can be decompressed from two double precision (64 bits) variables before the interval cannot be any smaller. We run a program that decompresses random numbers in various precisions. During the decompression, the program checks whether it can partition the interval into a smaller one. (The split is in the lines 7, 8 and 11, 12 of the pseudo code in Figure 2.) This can be done by checking whether the new interval is not smaller than the previous interval. Figure 8 shows the average number of bits that can be decompressed from p in various ranges. The result shows that, on average, when p is closer to the bound of the range [0, 1), the algorithm can decompressed longer binary string. Moreover, p is a bias toward more zero's or one's in a decompressed binary string. When p is close to zero, the decompression algorithm will produce more 1's. For example, when $(p, c) \approx (4.42 \times 10^{-7}, 0.61)$ and $\approx (8.62 \times 10^{-10}, 0.70)$, the decompression algorithm will produce all 10^6 and 10^9 one's respectively. Those values are the best chromosome obtained from two EDAAC runs.

In addition, we conducted an experiment to show the average number of bits that can be decompressed from decimal numbers with various precisions. The precisions we tested in the experiment are 10, 20, 40, 80 digits and a double precision. The result is shown in Figure 9. The white rectangle in the figure is a data from a double precision variable. The graph shows that the number of bit that can be decompressed is about 10 times the precision of the decimal number.

VII. CONCLUSION AND FUTURE WORK

EDAAC can solve one-million-bit OneMax, Royal Road, and Trap functions very fast. It can solve one-billion-bit OneMax in reasonable amount of time using a normal PC-class computer. We believe that our result on computation time is a significant improvement over other method to solve large scale problems. However, one might argue that such compressed encoding GA performed well on those problems because they have a high regularity solution. It would be interesting to investigate how EDAAC solves other large scale problems that the solution is not all 1's or all 0's.

ACKNOWLEDGEMENT

We would like to thank Thotsaphon Thanatipanonda for giving the idea on how to show the average number of bits that can be decompressed from double precision variables.

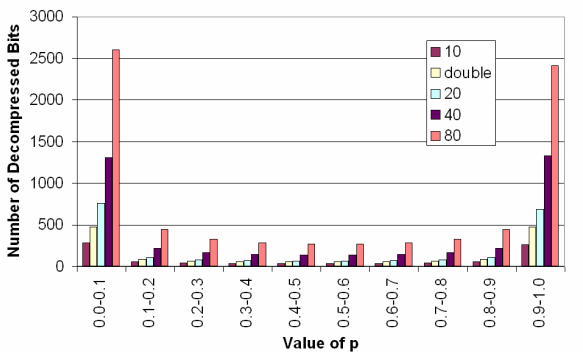

Figure 8. Average number of bits that can be decompressed from various values of *p*.

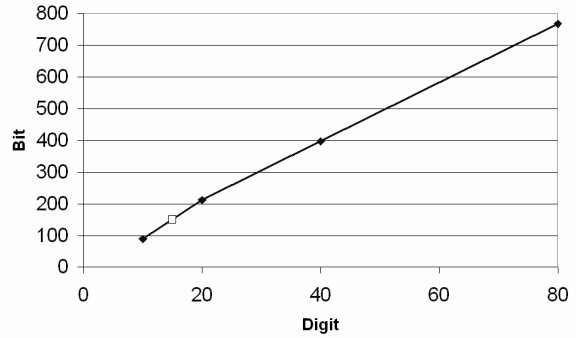

Figure 9. Average number of bits that can be decompress from a decimal number with various digits.

REFERENCES

[1] N. Kunasol, W. Suwannik, P. Chongstitvatana, "Solving One-Million-Bit Problems Using LZWGA," Proceedings of International Symposium on Communications and Information Technologies (ISCIT), October 18-20, 2006.

[2] K. Sastry, D. E. Goldberg, D. E., X. Llorà, "Towards billion bit optimization via efficient genetic algorithms," IlliGAL Report No. 2007007. University of Illinois at Urbana-Champaign, Urbana IL, 2007.

[3] 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.

[4] F.G. Lobo, K. Deb, D.E. Goldberg, G. Harik, L. Wang, "Compressed Introns in a Linkage Learning Genetic Algorithm," Genetic Programming : Proceedings of the Third Annual Conference, Madison, Wisconsin, pages 551-558, 1998.

[5] 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)

[6] 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)

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

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

[9] 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.