GAGA: Model Building Genetic Algorithms Using Sub-population and Sub-probability Vector

Jiradej Ponsawat, Sunisa Rimcharoen, Daricha Sutivong and Prabhas Chongstitvatana

Department of Computer Engineering, Faculty of Engineering,
Chulalongkorn University, Phayathai Road
Bangkok 10330 Thailand
jiradejhm@hotmail.com, suni16@hotmail.com, daricha.s@chula.ac.th, prabhas@chula.ac.th

Abstract

The Compact Genetic Algorithm (cGA) has a distinct characteristic that it requires almost minimal memory to store candidate solutions. The probability vector is used to generate candidate solutions. This probability vector represents a structure of the population as a probability distribution over the set of solution. It has been established that the power of cGA is comparable to the standard Simple Genetic Algorithm (sGA) with uniform crossover. Hence, its limitation hinges on the assumption of the independency between each individual bit. For example, a standard difficult test problem for GA is a deceptive function, or so called Trap function. cGA fails to solve this problem. This work proposes another approach of using the probability vector as a model of the structure of solutions, named GAGA (for GA-in-GA). GAGA employs two new methods for updating the probability vector. The first-method uses a sub-genetic algorithm and the second method uses a sub-probability vector. The experimental results show that the proposed methods can solve the problem that has the dependency between bits such as Trap function.

1. Introduction

The genetic algorithm [3] is an optimization algorithm inspired by natural evolution. The GA is performed by creating a population of solutions and uses genetic operators, e.g. reproduction, crossover and mutation to produce offsprings. The solutions are gradually improved by a selection scheme which selects the survivors by their fitness values defined by users. Contrary to the GA, the cGA proposed by Harik et al. [2] represents the population as a probability distribution over the set of solution; thus, the whole population needs not to be stored. At each generation, cGA samples individuals according to the probabilities specified in the probability vector. The individuals are evaluated and the probability vector is updated towards the better individual. The cGA mimics the order-one behavior of sGA with uniform crossover using a small amount of memory and achieves comparable quality with approximately the same number of fitness evaluations as the sGA [2]. However, the cGA does not provide acceptable solutions to difficult problems such as deceptive problem or so called trap function which is a standard difficult test problem for GA.

In order to update the probability vector we should not use only the positive sample (the individual with high fitness) because in the early generation the population contains not so good individual. Our intuition is to include the negative sample. Another suggestion is to invest in improving the population before choosing the sample for update. By spending small computational effort we can have a more accurate sample which should lead to a faster convergence. This work proposes two new methods for updating the probability vector. The first-method uses a sub-genetic algorithm and the second method uses a sub-probability vector. The proposed method is named GAGA (for GA-in-GA) because it is intended to build

a model of solutions not using any heuristics but using power of another GA inside the main GA to do it.

The first technique is similar to using another GA to improve the sub-population. The probability vector is sampled with a population size of O(log n) where n is the population size of sGA. The population is evolved over small number of steps. The best and the worst individuals are compared. The different bits are more important than the similar bits in the update rule. The method is based on the belief that the identical bits give the same fitness, while those distinct bits give unequal fitness.

Another method uses a group of individuals to update the probability vector instead of one individual. The idea behind this method rests on grouping the individuals as a sub-probability vector. A small population is generated from the probability vector. They are evaluated and individuals are arranged into groups of similar fitness. A sub-probability vector is calculated from a probability of each bit in the group. The update rule uses the difference between a sub-probability of the best group and a random group selected from a top half.

The paper is organized as follows: Section 2 introduces the compact genetic algorithm. Section 3 describes the technique using sub-genetic algorithm. Section 4 describes the technique using sub-probability vector. Experiment results are provided in section 5 and a conclusion is drawn in section 6.

2. The Compact Genetic Algorithm

The pseudocode of the compact genetic algorithm [2] is shown in Figure 1. The parameters are population size(n) and chromosome length(l). First, the probability vector p is initialized to 0.5. Next, the individuals a and b are generated from p. The fitness values are then assigned to a and b. The probability vector is updated towards the better individual. The updating step size is $1/n$; the probability vector is increased or decreased by this size. The loop is repeated until the vector converges.

```
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 := compete(a, b);

4) update the probability vector towards the better one
   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;
```

Figure 1: Pseudocode of the compact genetic algorithm

Harik [2] also proposes a modification of the compact genetic algorithm with a higher selection pressure. It simulates a tournament size *s*. Figure 2 shows the modification of the compact genetic algorithm.

```
1) generate s individuals from the vector and store them in S
   for i := 1 to s do S[i] := generate(p);

2) rearrange S so that S[1] is the individual with higher fitness

3) let S[1] compare with the other individuals
   for i := 2 to s do
   begin
     winner, loser := compete(S[1], S[i]);
     update probability vector (step 4 of cGA code)
   end
```

Figure 2: Pseudocode of a modification of the compact genetic algorithm

3. Sub-Genetic Algorithm

The update rule for the probability vector in cGA is influenced mostly by the winner (see fig.2). The motivation for sub-GA method is that using only the information from strong individuals might not be the best policy. As in the early stage most individuals are bounded not to be fit, the strong individuals at this stage may lead to the wrong direction. The proposed method is to also use the information from the weak individual. A small size population is generated from the probability vector to be a sample of population, called sub-population. This sub-population is evolved by a limited number of generations, called quasi evolution. In the quasi-evolution, the Simple Genetic Algorithm is used with a single point crossover, tournament selection and no mutation. The aim here is to develop a clear winner and loser and use them in the update rule. The use of the loser will prevent the domination of the winner.

The update rule compares these two individuals and update the probability vector toward the winner. All bit-positions will be updated, and the position of the bits that are different is updated with two times the weight of the position of the bits that are the same. The pseudo code for sub-population GA is shown in Figure 3.

4. Sub-Probability Vector

This method uses a group of individuals to update the probability vector(p). A small population(m) is generated from the probability vector. The individuals are evaluated and arranged in to group of similar fitness. A sub-probability vector is calculated from all members in the group. The update rule uses a sub-probability of the best group and a random group selected from a top half. The adjustment of the probability vector is calculated from a difference of these two groups multiplied by m/n, where $1/n$ is analogous to the step size in cGA. Pseudo code of this method is shown in Figure 4.

Parameter *m* : the number generated individual
l : chromosome length
S : sub-population

1) initialize probability vector
   for *i* := 1 to *l* do *p*[*i*] := 0.5;

2) step := *m* / *n;*

3) generate *m* individuals from the vector and store them in *S*
   for *i* := 1 to *m* do *S*[*i*] := generate(*p*);

4) quasi-evolve the sub-population

5) select the best and the worst individuals
   best := the best individual in *S*
   worst := the worst individual in *S*

6) update probability p
   for *i*:=1 to *l* do
     if *best*[*i*] ≠ *worst*[*i*]
       update[*i*] := 2 * *step*;
     else
       update[*i*] := *step*;
     if best[i] = 1
       p[*i*] := *p*[*i*] + *update*[*i*];
     else
       p[*i*] := *p*[*i*] - *update*[*i*];
   end for

7) while vector *p* not converge, return to step 2)

Figure 3: Pseudocode for sub-population GA

Parameter m : the number of generated individuals
l : chromosome length
g : the number of groups
a : the number of the individuals in the best group
b : the number of the individuals in the selected group
G : a set of individual

1) initialize probability vector
for *i* := 1 to *l* do *p*[*i*] := 0.5;

2) generate m individuals from the vector
for *i* := 1 to *m* do *individual*[*i*] := *generate*(*p*);

3) evaluate
for *i* := 1 to *m* do *individual*[*i*].*fitness* := *evaluate*(*individual*[*i*]);

4) grouping the individuals by their fitness
for *j* := 1 to *g* do
for *i* := 1 to *m* do
if *individual*[*i*].*fitness* = *group*[*j*].*fitness* then
G[*j*] := *G*[*j*] U {*individual*[*i*]}

5) select the best group
winner := *G*[*i*] | max(group[*i*].fitness)

6) randomly select an another group to compare with the best group
sort *G*[*i*] by group[*i*].fitness
loser := *G*[*R*]; where *R* is uniform random *g*/2..*g*

7) compute probability vector of winner and loser

$$p_{winner}[i] := \sum^{k} winner_k[i] / a;$$

$$p_{loser}[i] := \sum^{k} loser_k[i] / b;$$

where *k* is k^{th} element in the set

8) compute the adjust value
for *i* := 1 to *l* do
adjust[*i*] := *m* * (P_{winner}[*i*] - P_{loser}[*i*]) * 1 / *n*;

9) update the probability vector
for *i* := 1 to *l* do *p*[*i*] := *p*[*i*] + *adjust*[*i*];
if *p*[*i*] > 1 then *p*[*i*] := 1;
if *p*[*i*] < 0 then *p*[*i*] := 0;

10) while vector *p* not converge, return to step 2

Figure 4: Pseudocode for sub-probability GA

5. Experiment Results

5.1 Testing problems

In the experiments, we test the algorithms using two test problems: 100 bit one-max problem and 3x10-bit trap problem. The data are averaged over 50 runs. All runs end when the vector fully converges, that is all positions are zero or one.

One-max problem is a simple test problem for GA. This problem finds a maximum value in which all bits are one. The fitness value is assigned according to the number of bits that are one in the chromosome. Thus, the maximum value is equal to chromosome length.

The trap function [1] is a difficult test problem for GA. The general *k*-bit trap function is defined as:

$$F_k(b_0 \ldots b_{k-1}) = \begin{cases} f_{\text{high}} & ; \quad \text{if } u = k \\ f_{\text{low}} - u\dfrac{f_{\text{low}}}{k-1} & ; \quad \text{otherwise} \end{cases} \tag{1}$$

Where $b_i \in \{0, 1\}$, u = $\sum_{i=0}^{k-1} b_i$, and $f_{\text{high}} > f_{\text{low}}$. Usually, f_{high} is set at k and f_{low} is set at k-1. The test function $F_{\text{k} \times \text{m}}$ is defined as:

$$F_{k \times m}(B_0 \ldots B_{m-1}) = \sum_{i=0}^{m-1} F_k(B_i),\ B_i \in \{0,1\}^k \tag{2}$$

This function fools gradient-based optimizers to favor zeroes, but the optimal solution is composed of all ones. The *k* and *m* vary to produce a number of test functions. For example, 3x5 bit trap function is shown in Table 1.

Table 1: Example of 3x5 bit trap function

Individual	$b_0b_1b_2$	$b_3b_4b_5$	$b_6b_7b_8$	$b_9b_{10}b_{11}$	$b_{12}b_{13}b_{14}$	Fitness
1	111	111	000	111	000	13.0
2	000	000	111	000	111	12.0
3	111	111	011	111	111	12.0
4	111	000	000	111	000	12.0
5	111	001	010	111	111	11.0
6	000	000	000	000	111	11.0
7	111	001	110	111	111	10.0
8	000	000	000	000	000	10.0

5.2 Parameters

The proposed methods are compared to cGA and sGA. In order to make a meaningful comparison, the population size of sGA is used as one parameter (*n*). Two measurements are collected: the number of function evaluations and the solution quality which is measured by the number of traps that are successfully solved when the algorithms stop. To calibrate the size of sub-population, two population sizes are introduced log *n* and $\sqrt{n}$.

The update rule requires two parameters, e.g. the population size of simple GA and a step size. The step size is an important parameter used in this experiment. It is related to the number of individuals generated from the probability vector which is a population sample. In the sub-probability method, the actual size of population is equal to the tournament size used in compact genetic algorithm. In the sub-GA method, the actual number of population is either log n or $\sqrt{n}$. Thus, the proposed methods use population size smaller than sGA. The number of population sizes in the experiments are compared with those in sGA in Table 2.

Table 2: Real population size

sGA	sGA	Sub-probability vector	Sub-GA log n	Sub-GA $\sqrt{n}$
8	2, 4, 8	2, 4, 8	3	2
500	2, 4, 8	2, 4, 8	8	22
1000	2, 4, 8	2, 4, 8	9	31
1500	2, 4, 8	2, 4, 8	10	38
2000	2, 4, 8	2, 4, 8	10	44
2500	2, 4, 8	2, 4, 8	11	50
3000	2, 4, 8	2, 4, 8	11	54

5.3 Results

In the 100 bit one-max problem, the results are comparable to sGA and cGA in terms of performance and solution quality. Figure 5 shows the solution quality and the number of function evaluations taken to converge.

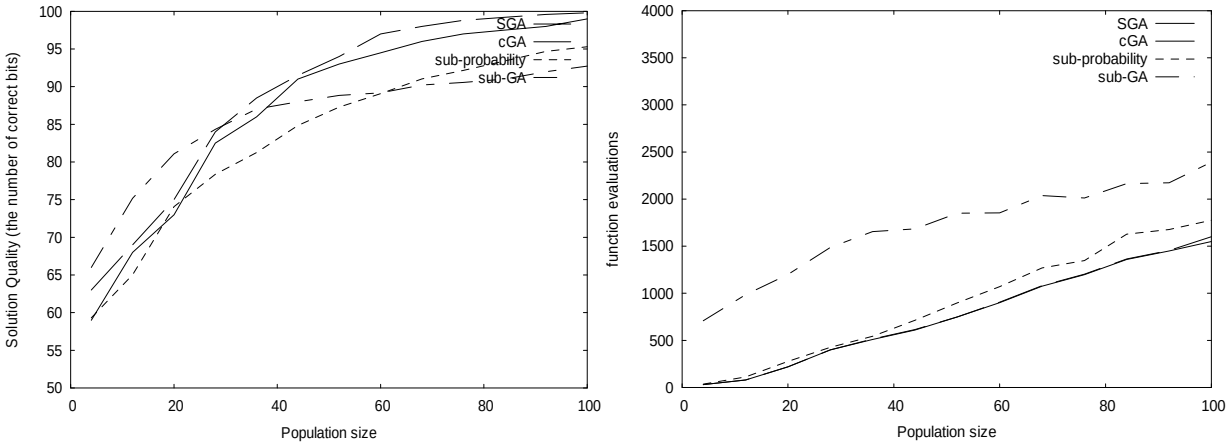
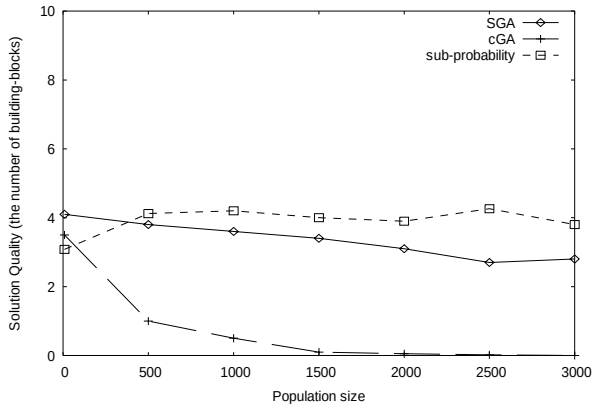
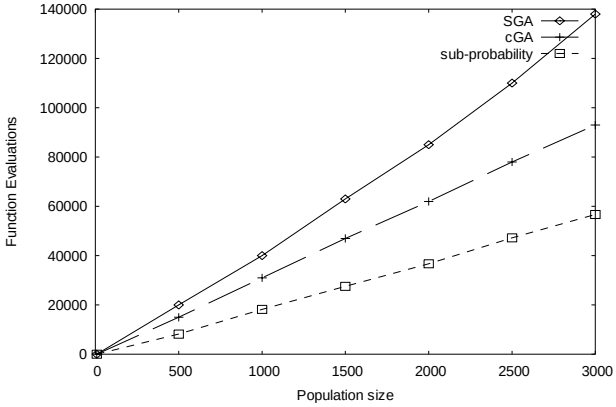


Figure 5: 100 bit one-max problem

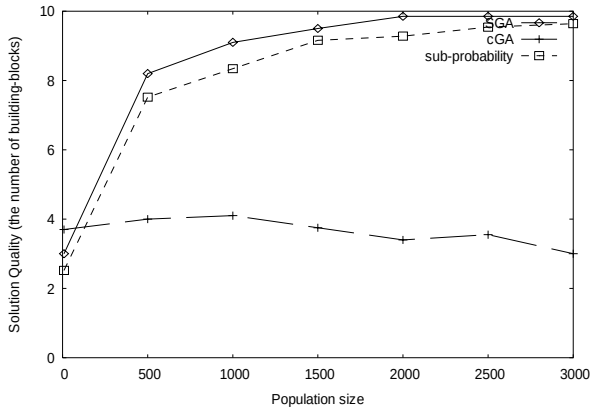
In 3x10 bit trap problem, figure 6 shows comparisons among the simple GA, the compact GA, and sub-probability algorithm using a tournament size of two, four and eight and population sizes of 8, 500, 1000, 1500, 2000, 2500 and 3000. On the left column, the graphs illustrate the number of trap-functions solved. On the right column, the graphs display the number of function evaluations.

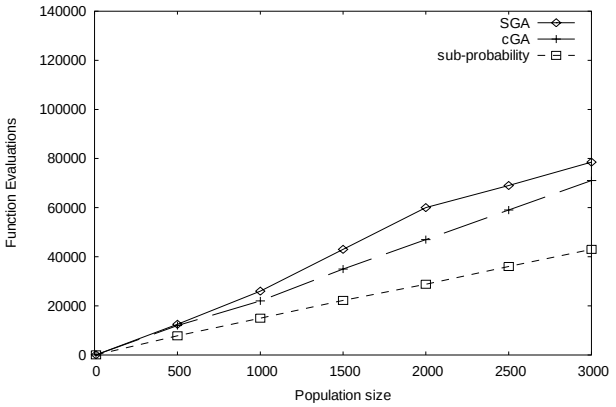
Tournament size = 2



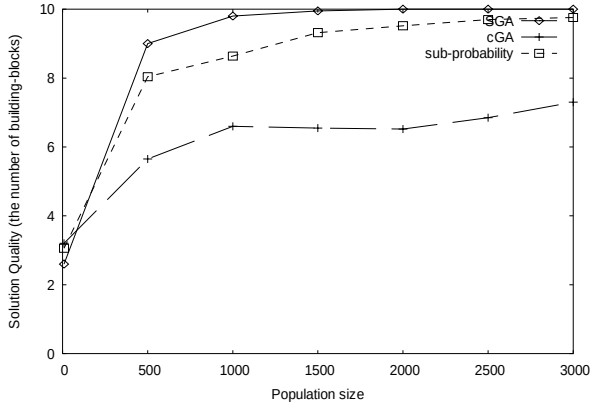


Tournament size = 4





Tournament size = 8



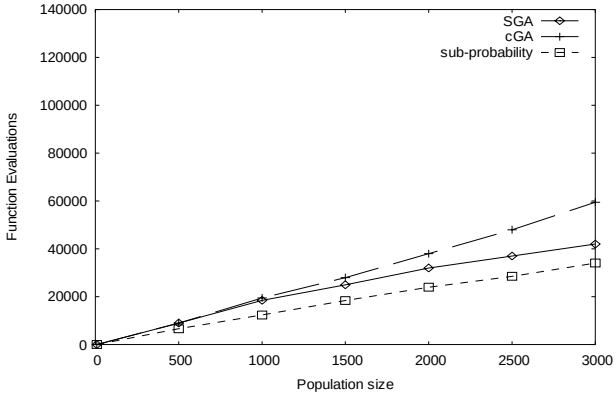


Figure 6: 3x10 bit trap problem using sGA, cGA and sub-probability algorithm

The 3x10 bit trap results for sub-GA method are shown in figure 7. The comparisons are between the sub-probability method, the sub-GA with log *n* and the sub-GA with $\sqrt{n}$. On the left side, the graph shows the number of sub-functions solved. On the right side, the graph shows the number of function evaluations.

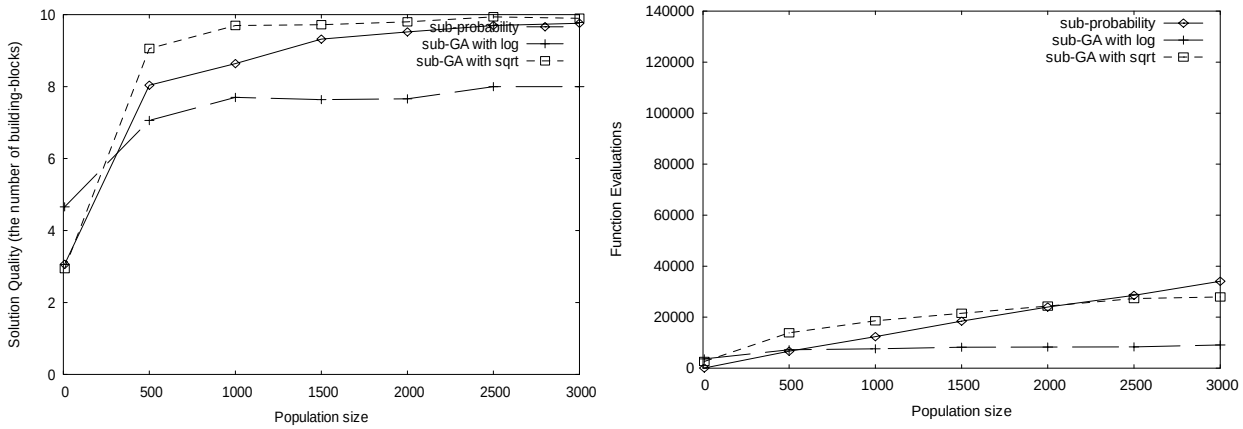


Figure 7: 3x10 bit trap function problem using sub-GA

5.4 Discussion

The sub-probability vector method requires population size smaller than the simple GA. In this experiment, the probability vector generates a small population (of size 2, 4 and 8). With the same population size, it can obtain higher quality of solution than the compact GA. The result is comparable to the simple GA that uses a large population. The advantage of this technique is that it requires a smaller population. Moreover, the number of function evaluations is less than the sGA and the cGA. Comparing with the sub-GA, the sub-probability method generates the smallest population size but the number of function evaluations is higher.

The sub-GA method with log *n* requires a minimum number of function evaluations but the solution quality also drops. In order to improve the solution quality, population size of $\sqrt{n}$ is used. With a larger population size, the sub-GA yields a better quality.

6. Conclusions

Two test problems are used: 100-bit one-max problem and 3x10-bit Trap problem. The results are compared with the simple genetic algorithm and the original compact genetic algorithm. The experiment results show that the proposed method can find the solutions of both problems with fewer number of function evaluations using very small population size.

References

[1] Ackley, D. H., "A Connectionist Machine for Genetic Hillclimbing", Kluwer Academic Publishers, Boston, MA, 1987.

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

[3] Holland, J. H., "Adaptation in Natural and Artificial Systems, University of Michigan Press, 1975.