



Evolution of Appropriate Crossover and Mutation Operators in a Genetic Process*

TZUNG-PEI HONG

Department of Information Management, I-Shou University, Kaohsiung, 84008, Taiwan, Republic of China
tphong@isu.edu.tw

HONG-SHUNG WANG

Institute of Electrical Engineering, Chung-Hua University, Hsin-Chu, 30067, Taiwan, Republic of China

WEN-YANG LIN

Department of Information Management, I-Shou University, Kaohsiung, 84008, Taiwan, Republic of China
wylin@isu.edu.tw

WEN-YUAN LEE

Institute of Information Engineering, I-Shou University, Kaohsiung, 84008, Taiwan, Republic of China
m883322m@isu.edu.tw

Abstract. Traditional genetic algorithms use only one crossover and one mutation operator to generate the next generation. The chosen crossover and mutation operators are critical to the success of genetic algorithms. Different crossover or mutation operators, however, are suitable for different problems, even for different stages of the genetic process in a problem. Determining which crossover and mutation operators should be used is quite difficult and is usually done by trial-and-error. In this paper, a new genetic algorithm, the dynamic genetic algorithm (DGA), is proposed to solve the problem. The dynamic genetic algorithm simultaneously uses more than one crossover and mutation operators to generate the next generation. The crossover and mutation ratios change along with the evaluation results of the respective offspring in the next generation. By this way, we expect that the really good operators will have an increasing effect in the genetic process. Experiments are also made, with results showing the proposed algorithm performs better than the algorithms with a single crossover and a single mutation operator.

Keywords: genetic algorithms, evolution, crossover, mutation

1. Introduction

Genetic algorithms (GAs) [1, 2] have become increasingly important for researchers in solving difficult problems since they could provide feasible solutions

in a limited amount of time [3]. They were first proposed by Holland in 1975 [2] and have been successfully applied to the fields of optimization [1, 4, 5], machine learning [1, 4], neural network [5], fuzzy logic controllers [6], and so on. GAs are developed mainly based on the ideas and the techniques from genetic and evolutionary theory [7]. According to the principle of *survival of the fittest*, they generate the next population by several operations, with each

*This is a modified and expanded version of the paper "Simultaneously applying multiple crossover and mutation operators," presented at the Genetic and Evolutionary Computation Conference, 1999.

individual in the population representing a possible solution. There are three principal operations in a genetic algorithm:

1. The *crossover* operation generates offspring from two chosen individuals in the population, by exchanging some bits in the two individuals. The offspring thus inherit some characteristics from each parent.
2. The *mutation* operation generates offspring by randomly changing one or several bits in an individual. Offspring may thus possess different characteristics from their parents. Mutation prevents local searches of the search space and increases the probability of finding global optima.
3. The *selection* operation chooses some offspring for survival according to predefined rules. This keeps the population size within a fixed constant and puts good offspring into the next generation with a high probability.

In the earliest genetic algorithms, the related research on Holland's original GA [2] usually used a single crossover operator and a single mutation operator to produce successive generations. There is evidence showing that for many applications the crossover and mutation operators adopted are the key to the success of the genetic algorithms [2, 8–10]. Determining which crossover and mutation operators should be used is quite difficult and is usually done by trial-and-error. In [11] and [12], we respectively discussed the effect of dynamically adjusting multiple crossover operators and multiple mutation operators. In this paper, we propose the dynamic genetic algorithm (DGA) to simultaneously apply multiple crossover and mutation operators in the genetic process. Their ratios are dynamically adjusted according to the evaluation results of the respective offspring in the next generation. The dynamic genetic algorithm can be thought of as a generalization of the traditional genetic algorithms by setting the initial ratios of the applied crossover and mutation operators as the assigned values and the initial ratios of the other operators (not applied) as zero.

Experiments are also made to show the effectiveness of the proposed dynamic genetic algorithm. The test suite includes some simple functions, the functions used in [13], the optimization problems in [14], and the 0-1 knapsack problem [15]. The experimental results show that the proposed algorithm performs better than those with a single crossover and a single mutation

operators. They also show that the proposed algorithm is better than those in [11] and [12].

The content of this paper is organized as follows. A brief review of genetic algorithms and related work is introduced in Section 2. The dynamic genetic algorithm is proposed in Section 3. An example is also given there to illustrate the proposed algorithm. Section 4 describes the experiments we made. Finally, conclusions and future work are given in Section 5.

2. Review of Genetic Algorithms and Related Work

On applying genetic algorithms to solving a problem, the first step is to define a representation that describes the problem states. The most common way used is the bit string. An initial population is then defined and three genetic operations (crossover, mutation, and selection) are performed to generate the next generation. Traditional genetic algorithms use a single crossover operator and a single mutation operator throughout the entire evolutionary process. This procedure is repeated until the termination criterion is satisfied. This so called simple genetic algorithm (SGA) [1] is described as follows.

The simple genetic algorithm.

Step 1: Create an initial population of N individuals for evolution.

Step 2: Define a suitable fitness function for the individuals.

Step 3: Perform genetic operations (crossover and mutation) to generate offspring.

Step 4: Evaluate the fitness of each individual.

Step 5: Select N superior individuals according to their fitness values to form the next generation.

Step 6: If the termination criterion is not satisfied, go to Step 3; otherwise, stop the algorithm.

The simple genetic algorithm uses several parameters such as population size (N), crossover probability (p_c) and mutation probability (p_m). A number of guidelines existed in the literature for setting the values of p_c and p_m [7, 13, 16]. These general guidelines were drawn from empirical studies on a fixed set of test problems and were inadequate since optimal use of p_c and p_m is specific to the problem under consideration. Some studies particularly focused on finding optimal mutation rates [17–19]. These heralded the need for

self-adaption of the crossover or mutation rates. In [20], Fogarty used a varying mutation rate, demonstrating that a mutation rate that decreased exponentially over generations had superior performance. An analogous way of cyclically varying the mutation rate reported in [21] exhibited similar effect. In [22], an adaptive genetic algorithm was proposed in which p_c and p_m were varied according to the fitness values of the solutions.

There were also works on devising adaptive crossover operators instead of varying the crossover rates [23]. Examples included Punctuated Crossover [24], Masked Crossover [25], Adaptive Uniform Crossover [26], Selective Crossover [23], and Simulated Binary Crossover [27]. Besides, in 1989, Davis first proposed the concept of *adapting probabilities of operators selection* [28, 29]. Several operators were employed and the probabilities of applying each operator were adapted according to the performance of the offsprings generated by the operator. Since then, several similar works have also been done [11, 30–33, 35–37].

3. The Dynamic Genetic Algorithm

Traditional genetic algorithms use only a single crossover operator and a single mutation operator to produce offspring. Different crossover operators can however produce different styles of offspring and different mutation operators can traverse different search directions in state space, thus affecting the performance of the applied genetic algorithm. Designing a new genetic algorithm to automatically choose the appropriate crossover and mutation operators are then necessary. In this section, we propose a new genetic algorithm, the dynamic genetic algorithm (DGA), to achieve this purpose.

3.1. Description

The dynamic genetic algorithm simultaneously applies multiple crossover and mutation operators in each genetic process and automatically adjusts their ratios. Thus, the dynamic genetic algorithm can be thought of as a generalization of the conventional genetic algorithms by setting the initial ratios of the applied crossover and mutation operators as the assigned values and the initial ratios of the other operators (not applied) as zero. The flow chart of the dynamic genetic algorithm is shown in Fig. 1.

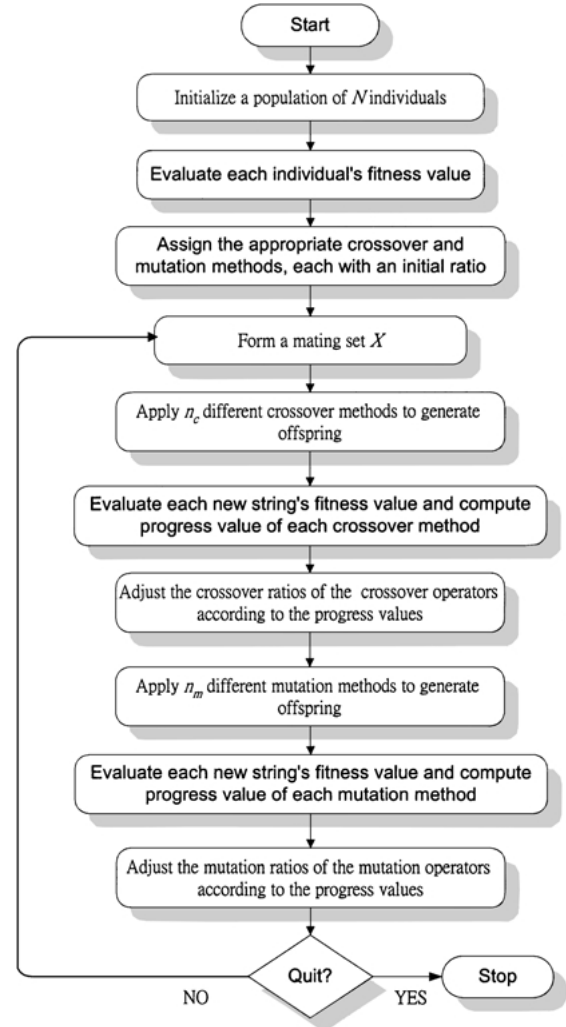


Figure 1. The flow chart of the dynamic genetic algorithm.

The detailed algorithm is described as follows.

The dynamic genetic algorithm.

Step 1: Create an initial population set of N individuals for evolution.

Step 2: Evaluate the fitness of each individual.

Step 3: Assign n_c candidate crossovers and n_m mutations, each with an initial ratio.

Step 4: Form a mating set X . The parents are then picked up at random from the mating set X .

Step 5: Apply the crossover operators to generate the offspring (each crossover operator is applied proportionally to its ratio).

Step 6: Evaluate the fitness values of each individual and calculate the average progress value generated by each crossover operator as follows.

Assume parents p and q are chosen by a crossover operation to produce two offspring a and b . The progress value of the crossover operator is calculated as follows:

(The sum of first two biggest fitness values among $f(p), f(q), f(a), f(b)$) $-(f(p) + f(q))$.

Assume that r pairs of parents are chosen by this crossover operator in this generation. The average progress value $Progress(C_i)$ of this crossover operator C_i is the average of the r values.

Step 7: Adjust the crossover ratios of the candidate crossover operators according to the average progress values.

Step 8: Apply the mutation operators to generate the offspring.

Step 9: Evaluate the fitness values of each individual and calculate the average progress value generated by each mutation operator as follows.

Assume that a new string p is chosen by a mutation operator to produce a new offspring a . The progress value of the mutation operator is calculated as follows:

$$\max\{f(p), f(a)\} - f(p).$$

If r old strings are chosen by this mutation operator in this generation, the progress value $Progress(M_i)$ of this mutation operator M_i is then the average of the r values.

Step 10: Adjust the mutation ratios of the candidate mutation operators according to the average progress values.

Step 11: Select N individuals to form the new population.

Step 12: If the termination criteria are not satisfied, go to Step 4; otherwise, stop the algorithm.

After Step 12, the individual with the highest fitness value is then output as the best solution. Below, an example is given to demonstrate the algorithm.

3.2. An Example

Assume that we want to find a value of t to maximize the following function by the dynamic genetic algorithm:

$$f(t) = t^4 |\sin(5\pi t)|, \quad t \in [0.000, 1.024],$$

find the max.

Table 1. An example of initial population after Step 2.

No.	Bit string	t	$f(t)$
1	1101100000	0.864	0.4705
2	0110110000	0.432	0.0168
3	1111000001	0.961	0.4904
4	1100100010	0.802	0.0130
5	0111011010	0.474	0.0463
6	1000110111	0.567	0.0512
7	0000110001	0.049	0.0000
8	1000001011	0.523	0.0700

Each step of the algorithm is illustrated as follows.

Step 1: Create an initial population of N individuals for evolution.

Assume that the value of t is represented by a bit string of length = 10. Each individual in the population is then composed of 10 bits. For example, the genetic representation for $t = 0.738$ is 1011100010. Let $N = 8$. Eight individuals are thus randomly generated for evolution.

Step 2: Evaluate the fitness of each individual.

In this example, the fitness function is $f(t)$ itself. An example of the initial population after this step is shown in Table 1.

Step 3: Assign several appropriate or common candidate crossover and mutation operators, each with an initial ratio.

Assume that $p_c = 1$ and $p_m = 0.04$. Furthermore, assume that four crossover operators C_1 to C_4 and four mutation operators M_1 to M_4 are used. The initial ratio for each crossover operator C_i is $p_c/n_c = 1/4 = 0.25$, and the initial ratio for each mutation operator M_i is $p_m/n_m = 1/4 = 0.01$, for $i = 1$ to 4.

Step 4: Form a mating set X .

There are several methods to construct the mating set. For example, a chromosome can be put into the mating set by a probability proportional to its fitness value. The parents can be picked up at random from the mating set X to generate offspring.

Step 5: Apply the crossover operators to generate the offspring (each crossover operator is applied proportionally to its crossover ratio).

Table 2. The fitness values of eight individuals before operation.

No.	Old string	t	$f(t)$
1	1011100010	0.738	0.2453
2	1100001100	0.78	0.1144
3	1100101010	0.81	0.0673
4	0110101101	0.429	0.0149
5	0110100111	0.423	0.0113
6	0100100111	0.295	0.0076
7	0011111001	0.249	0.0027
8	0010010101	0.149	0.0004

In this step, the applied four crossover operators do crossover according to their crossover ratios.

Step 6: Evaluate the fitness value of each individual and calculate the average progress value generated by each crossover operator.

Assume that the eight individuals before operation are shown in Table 2 and the offspring generated by the crossover operators are shown in Table 3. The average progress value for a crossover operator is calculated as follows.

For example, parents No. 4 (p) and No. 8 (q) are chosen by crossover operator C_1 to produce two offspring N3 (a) and N4 (b). According to the algorithm in Section 3, the average progress value of the crossover operator C_1 is thus calculated as follows:

$$(0.0149 + 0.0021) - (0.0149 + 0.0004) = 0.0017$$

Repeating the same calculation for all the other crossover operators, the average progress values are shown in Table 4.

Table 3. Eight offspring generated by the crossover operators.

No.	Parents	Offspring	t	$f(t)$	Crossover method
N1	(7, 8)	0011010001	0.209	0.0003	C_4
N2	(7, 8)	0010111101	0.189	0.0002	C_4
N3	(8, 4)	0110010101	0.405	0.0021	C_1
N4	(8, 4)	0010101101	0.173	0.0004	C_1
N5	(4, 6)	0100101101	0.301	0.0082	C_3
N6	(4, 6)	0110100111	0.423	0.0113	C_3
N7	(1, 8)	1010100011	0.675	0.1918	C_2
N8	(1, 8)	0011010100	0.212	0.0004	C_2

Table 4. The average progress values of the crossover operators.

Crossover method	Progress value	Rank
C_1	0.0017	3
C_2	0.0186	1
C_3	0.0037	2
C_4	0.0000	4

Step 7: Adjust the crossover ratios of the candidate crossover operators according to the average progress values.

There are several methods that can be used in adjusting the ratios of the crossover operators [11]. The geometric progression method [12], which adjusts the ratios of the crossover operators in a geometric progression, is explained here as an example. The crossover operators are first sorted by their progress values in each generation. The crossover operators ranked in the first half then increase their ratios by a geometric progression series, and those ranked in the second half then decrease their ratios by a geometric progression series. By this way, the best crossover method will gradually get the largest control rate and dominate the main crossover action in the genetic process.

Step 8: Apply the mutation operators to generate the offspring.

In this step, the adopted four mutation operators are executed according to their mutation ratios.

Step 9: Evaluate the fitness value of each individual and calculate the average progress value generated by each mutation operator.

The average progress value for a mutation operator is calculated in a way a little different from that for a crossover operator. Assume that string No. 3 (p) with fitness value 0.0673 in Table 2 is chosen by a mutation operator M_1 to produce a new offspring (a) with fitness value 0.0083. According to the proposed algorithm in Section 3, the average progress value of the mutation operator is $0.0673 - 0.0083 = 0.0590$.

The average progress values for the other mutation operators can be similarly calculated.

Step 10: Adjust the mutation ratios of the candidate mutation operators according to their average progress values.

This can be done in a way similar to that in Step 7.

Step 11: Select N individuals to form the new population.

The best N individuals will be picked up as the new population.

Step 12: If the termination criteria are not satisfied, go to Step 4; otherwise, stop the algorithm.

For this example, we stop the genetic algorithm at the 100th generation.

4. Experiments

In this section, we report the experiments made on showing the performance of the proposed dynamic genetic algorithm. We also compare the execution time of the proposed algorithm with that of the simple genetic algorithm. All programs were run on an Intel PC. The experiments consisted of two parts. In the first part, we tested fourteen function optimization problems that were classified into five groups; in the second part, we used the proposed algorithm to solve the 0-1 knapsack problem.

4.1. Experiment 1

In this experiment, we tested fourteen problems classified into five groups. The functions in Group A were simple functions with only one variable. The functions in Group B were simple functions with more than one variable. The problems in Group C were linear programming problems with multiple constraints. Group D included the problems used by De Jong [13] in 1975, and widely seen in the GA literature. Group E included the two special functions in [14]. These functions are described as follows.

(A) Single-variable functions:

Function 1: $f = x^3$, $0.000 \leq x \leq 16.384$, find the maximum.

Function 2: $f = x^4 |\sin(\pi x)|$, $0 < x < 16$, find the maximum.

(B) Multiple-variable functions:

Function 3: $f = x_1 - x_2 + x_3$, $0 \leq x_i \leq 1023$, find the maximum.

Function 4: $f = x_1 x_2 + x_3$, $0 \leq x_i \leq 1023$, find the maximum.

Function 5: $f = x_1/(x_2 + 1) + x_3$, $0 \leq x_i \leq 1023$, find the maximum.

Function 6: $f = x_1 x_2 x_3 - 100 x_1 x_2$, $-512 \leq x_i \leq 512$, find the maximum.

(C) Linear Programming Problems

Function 7:

$$4x_1 + 5x_2 + x_3 \leq 1000$$

$$x_1 + 4x_2 + 4x_3 \leq 1000$$

$$x_1 - 8x_2 + 15x_3 \leq 1200$$

$$0 \leq x_i \leq 1023$$

$$\text{Maximize } f = x_1 + 2x_2 + 10x_3.$$

Function 8:

$$4x_1 + 5x_2 + x_3 \leq 1000$$

$$5x_1 + 2x_2 + 4x_3 \leq 1000$$

$$3x_1 - 8x_2 + 5x_3 \leq 1200$$

$$0 \leq x_i \leq 1023$$

$$\text{Maximize } f = x_1 + 2x_2 + 10x_3.$$

(D) De Jong's Test Functions:

Function 9: $f = \sum_{i=1}^3 (x_i)^2$, $-5.12 \leq x_i \leq 5.12$, find the minimum.

Function 10: $f = 100(x_1^2 - x_2)^2 + (1 - x_1)^2$, $-2.048 \leq x_i \leq 2.048$, find the minimum.

Function 11: $f = \sum_{i=1}^5 \text{int}(x_i)$, $-5.12 \leq x_i \leq 5.12$, find the minimum.

Function 12: $f = \sum_{i=1}^{30} i x_i^4 + \text{Gauss}(0,1)$, find the minimum.

(E) Other Problems:

Function 13: $f = \sum_{i=1}^{10} -x_i \sin(\sqrt{|x_i|})$, $-512 \leq x_i \leq 12$, find the minimum.

Function 14: $f = \sum_{i=1}^{10} (x_i)^2/4000 - \prod_{i=1}^{10} \cos(x_i/\sqrt{i}) + 1$, $-512 \leq x_i \leq 512$, find the minimum.

The performance of the dynamic genetic algorithm by four crossover operators and four mutation operators is compared with that of the simple genetic algorithms, each with a single crossover operator and a single mutation operator. The following four crossover operators

and four mutation operators are used in the dynamic genetic algorithm:

Crossover 1. Single-point crossover: This method defines a mask to determine which bits should be exchanged between two individuals. However there is only one bit with value 1 on the mask. For the position of bit value 1 on the mask, the parents exchange the corresponding bits.

Crossover 2. Multi-point crossover [34]: This method defines a mask to determine which bits should be exchanged between two individuals. For the positions that are 1 on the mask, the parents exchange the corresponding bits.

Crossover 3. Substring crossover: This method changes arbitrary substrings between two individuals. Length and positions of these substrings are chosen at random, but are the same for both individuals.

Crossover 4. Uniform crossover [33]: This method defines a mask to determine which bits should be exchanged between two individuals. Bit values of 1 and 0 however alternative with each other on the mask. For the positions that are 1 on the mask, the parents exchange the corresponding bits.

Mutation 1. 0, 1 change: This method changes one bit of a string.

Mutation 2. Swapping: This method exchanges arbitrary two bits in a single chromosome.

Mutation 3. Inversion: This method changes the order of the bits in an arbitrary interval of the string.

Mutation 4. Bit-change: This method changes the bit value 0 to 1 and 1 to 0.

Traditionally, most genetic algorithms perform the 0, 1 change mutation method with a very small mutation probability. This mutation method is adopted in the following SGA1~SGA4, for comparison.

The relationship between the best fitness values (average 500 runs for each problem) and the generations can then be found from these experiments. In all the following figures, the notations are illustrated as follows:

SGA1: The simple genetic algorithm using crossover operator 1 and the 0,1 change mutation method;

SGA2: The simple genetic algorithm using crossover operator 2 and the 0, 1 change mutation method;

SGA3: The simple genetic algorithm using crossover operator 3 and the 0, 1 change mutation method;

SGA4: The simple genetic algorithm using crossover operator 4 and the 0, 1 change mutation method;

SGA5: The simple genetic algorithm using mutation operator 1 and the substring crossover method.

SGA6: The simple genetic algorithm using mutation operator 2 and the substring crossover method.

SGA7: The simple genetic algorithm using mutation operator 3 and the substring crossover method.

SGA8: The simple genetic algorithm using mutation operator 4 and the substring crossover method.

DGA: The dynamic genetic algorithm using crossover operators 1 ~ 4 and mutation operators 1 ~ 4.

Due to the limitation of paper length, only the experimental results for Function 10 are shown here. Similar results are observed for the other problems. The results for Function 10 are shown in Figs. 2 and 3.

$$f = 100(x_1^2 - x_2)^2 + (1 - x_1)^2, \\ -2.048 \leq x_i \leq 2.048, \text{ find the minimum.} \\ \text{String length: 24} \\ \text{Crossover rate: 1.0}$$

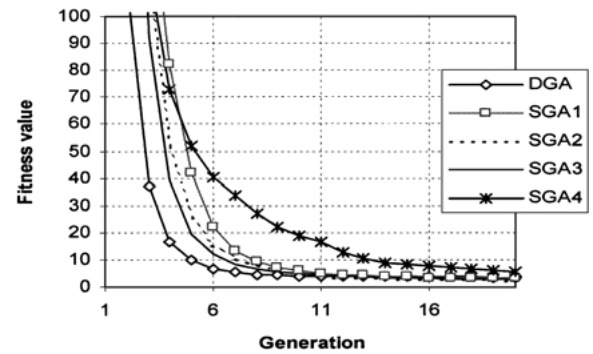


Figure 2. Experimental results of DGA and SGA1 to SGA4 for Function 10.

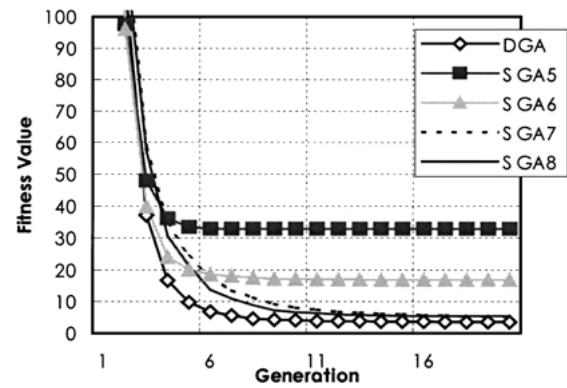


Figure 3. Experimental results of DGA and SGA5 to SGA8 for Function 10.

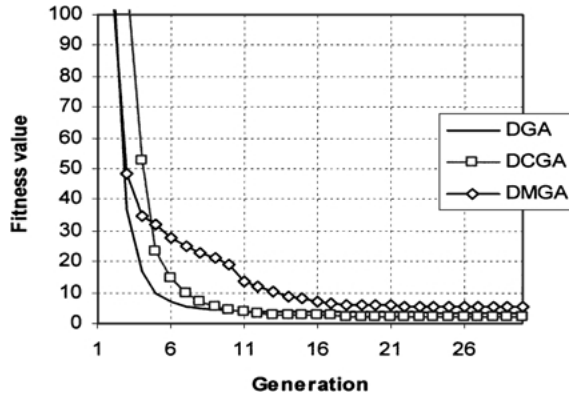


Figure 4. Experimental results of DGA, DCGA and DMGA for Function 10.

Mutation rate: 0.24
 Population size: 40
 Generation #: 40
 Experiment #: 500

From Figs. 2 and 3, it is observed that DGA has a better performance than the others, quite consistent with our discussion.

Next, experiments were made for comparing the performance of the proposed dynamic genetic algorithm with that of DCGA (which applied a fixed mutation operator and simultaneous applied multiple crossover operators) and DMGA (which applied a fixed crossover operator and simultaneous applies multiple mutation crossover operators). Results are shown in Fig. 4.

From Fig. 4, we find that DGA is better than DCGA and DMGA. DGA combines the advantages of these two methods and therefore converges faster and gets better fitness values than DCGA and DMGA.

Experiments were finally made for comparing the cpu time (in seconds) of using DGA and SGAs. The time is averaged over 500 runs for each algorithm. The experimental results for Function 10 are shown in Table 5, where SGA time is the average time of a simple GA, SGA16 is the average total time of 16 different SGAs, each with a different crossover or mutation operator, and DGA is the average time of a dynamic GA.

From Table 5, we can observe that the time of DGA is a little more than a single SGA. It is however much less than the summation of time in 16 SGAs. Since the performance of DGA is better than all the 16 SGAs, the extra time paid by using DGA is worth, rather than applying the 16 different SGAs to determine which combination yields the best result.

Table 5. The average CPU time spent for Function 10.

Gen#	Pop. Size	SGA time	SGA16 time	DGA time	DGA/SGA	DGA/SGA16
30	40	1.39	22.21	2.10	1.51	0.09

4.2. Experiment 2—The 0/1 Knapsack Problem

In this subsection, we examine the performance of the dynamic genetic algorithm on the 0/1 knapsack problem, which belongs to the class of knapsack-type problems and is well known as NP-hard [4]. The problem of 0/1 knapsack is, given a set of objects, a_i , for $1 \leq i \leq n$, together with their profits P_i , weights W_i , and a capacity C , to find a binary vector $\mathbf{x} = \langle x_1, x_2, \dots, x_n \rangle$, such that

$$\sum_{i=1}^n x_i \cdot W_i \leq C \quad \text{and} \quad \sum_{i=1}^n x_i \cdot P_i \text{ is maximal.}$$

Since the difficulty of the knapsack problems is greatly affected by the correlation between profits and weights [15], we followed the three randomly generated sets of data used in [4]:

- (1) uncorrelated:
 W_i and P_i : random(1..v);
- (2) weakly correlated:
 W_i : random(1..v),
 P_i : $W_i + \text{random}(-r..r)$;
- (3) strongly correlated:
 W_i : random(1..v),
 P_i : $W_i + r$.

The data were generated with the following parameter settings: $v = 10$, $r = 5$, and $n = 250$. Following a suggestion from [15], two different types of capacity C were adopted: 1) $C = 2v$, for which the optimal solution contained very few items; and 2) $C = 0.5 \sum W_i$, in which about half of the items were in the optimal solution.

To be consistent with the crossover and mutation operators considered, we used the simple binary string encoding scheme: each bit represented the inclusion or exclusion of an object. It was however possible to generate infeasible solutions with this representation. That was, the total weights of the selected objects would exceed the knapsack capacity. In the literature, there were two ways of handling the constraint violation [4]. One

way used a penalty function to penalize the fitness of the infeasible candidate so as to diminish its survival chance. The second way used a repair mechanism to correct the representation of the infeasible candidate. As indicated in [4], the repair method was more effective than the penalty approach. We hence adopted the repair approach in our implementation for both of the simple genetic algorithm and the dynamic genetic algorithm. The repair scheme we used was a greedy approach. All objects in a knapsack represented by an overfilled bit string were sorted in the decreasing order of their profit weight ratios. The last object was then selected for elimination (change the corresponding bit of “1” to “0”). This procedure was executed until the weight of the remaining objects was less than the total capacity. The parameters in this experiment were set as below:

Crossover rate: 1.0,
Mutation rate: 0.24,
Population size: 100,
Generation #: 500,
Experiment #: 20.

The greedy-repair DGA (GDGA) was compared with the eight greedy-repair SGAs (GSGAs) as chosen in Experiment 1. The results are shown in Figs. 5 to 10.

The test suit we used represents two different types of problem instances. Those data sets in Group 1 under constraint $C = 2v$ is quite simpler than those in Group 2 under constraint $C = 0.5 \sum W$, because the number of candidate solutions in Group 1 is much less than that in Group 2. For the simpler problems in Group 1 we can

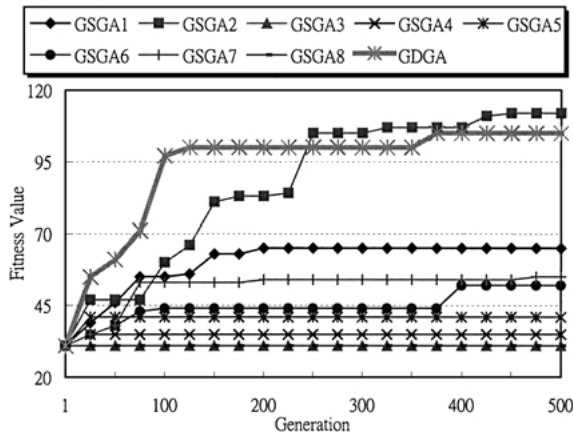


Figure 5. Experimental results on uncorrelated 0/1 knapsack for $C = 2v$.

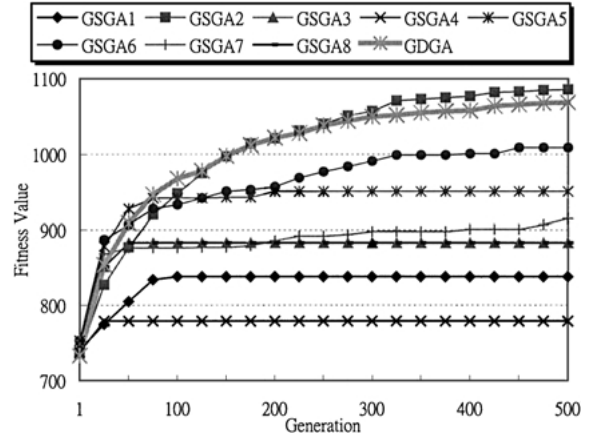


Figure 6. Experimental results on uncorrelated 0/1 knapsack for $C = 0.5 \sum W_i$.

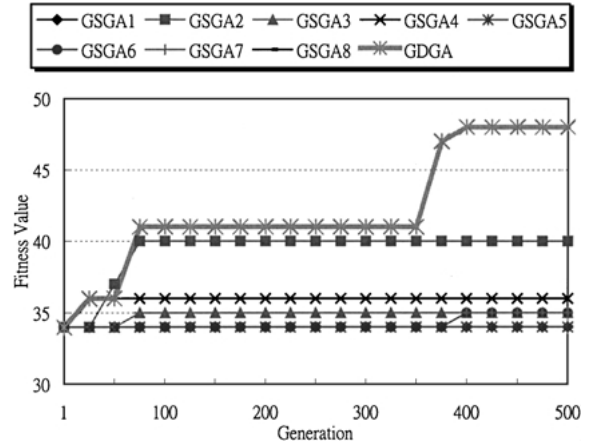


Figure 7. Experimental results on weakly correlated 0/1 knapsack for $C = 2v$.

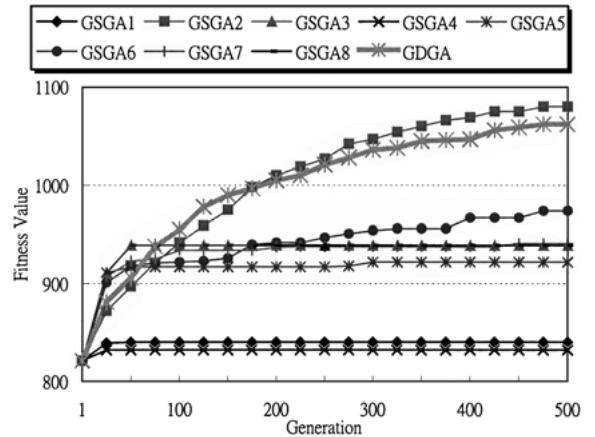


Figure 8. Experimental results on weakly correlated 0/1 knapsack for $C = 0.5 \sum W_i$.

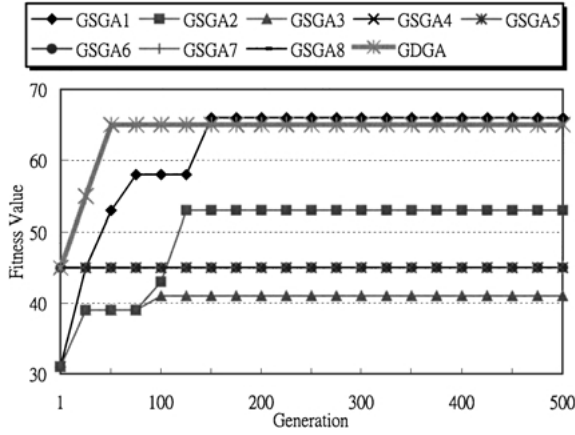


Figure 9. Experimental results on strongly correlated 0/1 knapsack for $C = 2v$.

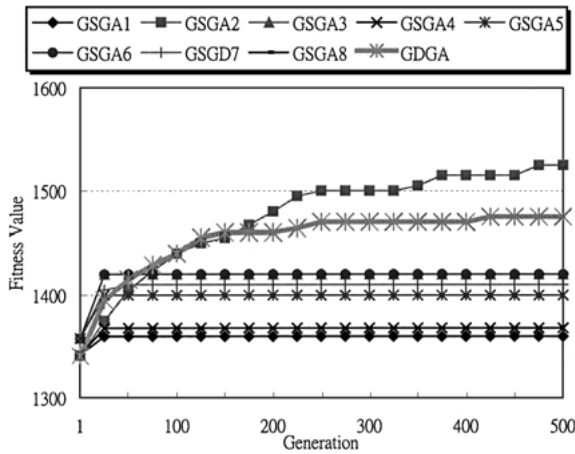


Figure 10. Experimental results on strongly correlated 0/1 knapsack for $C = 0.5 \sum W_i$.

observe from Figs. 5, 7 and 9 that GDGA and GSGA2 outperform the others and alternate the leading place both in searching speed and output quality. In the other group, we observe from Figs. 6, 8 and 10 that GDGA and GSGA2 again outperform the other methods in the resulting quality but evolve more slowly, and GDGA performs quite similar to GSGA2.

These observations show that GSGA2 is the best combination of concern for the 0-1 knapsack problem. However, this is concluded from 16 different trials. On the other hand, through the capability of automatically adjusting the appropriate operators, GDGA can ultimately find the most suitable operator combination, e.g., the multi-point crossover and 0,1 change muta-

tion in this experiment, thus behaves like GSGA2. And, sometimes, as in Fig. 7, GDGA can retain the population diversity to explore further to find better solution. This experiment illustrates again the capability of automatically adjusting the appropriate crossover and mutation operators eliminating the effort for determining the best combinations via trial-and-error.

5. Conclusions and Future Work

Genetic algorithms (GAs) have become increasingly important for researchers in solving difficult problems since they could provide feasible solutions in a limited amount of time. This paper has presented a new enhanced genetic algorithm for automatically applying suitable crossover and mutation operators to generate offspring for solving an arbitrary problem. The new genetic algorithm can save much time in searching for an appropriate crossover or mutation operator than applying the simple genetic algorithms one by one. The performance of the proposed genetic algorithm is also better than the simple genetic algorithms. Also, the simple genetic algorithms are the special cases of the proposed dynamic genetic algorithms by appropriately setting the adjusting ratios. The dynamic genetic algorithm thus makes a good trade-off between time-complexity and performance. In the future, we will attempt to design other sophisticated algorithms to automatically adapt the other parameters.

References

1. D.E. Goldberg, *Genetic Algorithms in Search, Optimization & Machine Learning*, Addison-Wesley: Reading, MA, 1989.
2. J.H. Holland. *Adaptation in Natural and Artificial Systems*, University of Michigan Press: Ann Arbor, MI, 1975.
3. A. Homaifar, S. Guan, and G.E. Liepins, "A new approach on the traveling salesman problem by genetic algorithms," in *Proceedings of the Fifth International Conference on Genetic Algorithms*, 1993.
4. Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs*, Springer-Verlag: Berlin, 1994.
5. M. Mitchell, *An Introduction to Genetic Algorithms*, MIT press: Cambridge, MA, 1996.
6. G.E. Sanchez, T. Shibata, and L.A. Zádach, "Genetic Algorithms and Fuzzy Logic Systems: Soft Computing Perspectives", World-Scientific: Singapore, 1997.
7. J.J. Grefenstette, "Optimization of control parameters for genetic algorithms," *IEEE Trans. Systems Man, and Cybernetics*, vol. 16(1), pp. 122–128, 1986.
8. Y. Davidor, "Analogous crossover," in *Proceedings of the Third International Conference on Genetic Algorithms*, 1989.

9. K. Deb and S. Agrawal, "Understanding interactions among genetic algorithm parameters," *Foundations of Genetic Algorithms* vol. 5, pp. 265–286, 1998.
10. D. Jong, "Adaptive system design: A genetic approach," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 10, pp. 566–574, 1980.
11. T.P. Hong and H. S. Wang, "Automatically adjusting crossover ratios of multiple crossover operators," *Journal of Information Science and Engineering*, vol. 14(2), pp. 369–390, 1998.
12. T.P. Hong and H.S. Wang, "A dynamic mutation genetic algorithm," in *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*, 1996, vol. 3, pp. 2000–2005.
13. D. Jong, "An analysis of the behavior of a class of genetic adaptive systems," University of Michigan, Ph.D. Thesis, 1975.
14. H. Muhlenbein, M. Schomisch, and J. Born, "The parallel genetic algorithm as function optimizer," in *Proceedings of the Fourth International Conference on Genetic Algorithms*, 1991.
15. S. Martello and P. Toth, *Knapsack Problems*, John Wiley: Chichester, UK, 1990.
16. J.D. Schaffer, R.A. Carvana, L.J. Eshelman, and R. Das, "A study of control parameters affecting online performance of genetic algorithms for function optimization," in *Proceedings of the Third International Conference on Genetic Algorithms*, 1989.
17. T. Bäck, "Optimal mutation rates in genetic search," in *Proceedings of the Fifth International Conference on Genetic Algorithms*, 1993, pp. 2–8.
18. J. Hesser and R. Männer, "Towards on optimal mutation probability for genetic algorithms," in *Proceedings of Parallel Problem Solving from Nature Conference*, 1991.
19. G. Ochoa, I. Harvey, and H. Buxton, "On recombination and optimal mutation rates," in *Proceedings of Genetic and Evolutionary Computation Conference*, 1999, pp. 488–495.
20. T.C. Fogarty, "Varying the probability of mutation in genetic algorithms," in *Proceedings of the Third International Conference on Genetic Algorithms*, 1989, pp. 104–109.
21. T.P. Hoehn and C.C. Petey, "Parental and cyclic-rate mutation in genetic algorithms: An initial investigation," in *Proceedings of Genetic and Evolutionary Computation Conference*, 1999, pp. 297–304.
22. M. Srinivas and L.M. Patnaik, "Adaptive probabilities of crossover and mutation in genetic algorithms," *IEEE Transactions on System, Man and Cybernetics*, vol. 24(4), pp. 17–26, 1994.
23. K. Vekaria and C. Clark, "Biases introduced by adaptive recombination operators," in *Proceedings of Genetic and Evolutionary Computation Conference*, 1999, pp. 670–677.
24. J.D. Schaffer and A. Morishima, "An adaptive crossover distribution mechanism for genetic algorithms," in *Proceedings of the Second International Conference on Genetic Algorithms*, 1987, pp. 36–40.
25. S.J. Louis and G.J.E. Rawlins, "Designer genetic algorithms: Genetic algorithms in structure design," in *Proceedings of the Fourth International Conference on Genetic Algorithms*, 1991, pp. 53–60.
26. T. White and F. Oppacher, "Adaptive crossover using automata," in *Proceedings of Parallel Problem Solving from Nature Conference*, 1994, pp. 229–238.
27. K. Deb and H. Beyer, "Self-adaptation in real-parameter genetic algorithms with simulated binary crossover," in *Proceedings of Genetic and Evolutionary Computation Conference*, 1999, pp. 172–179.
28. L. Davis, "Adapting operator probabilities in genetic algorithms," in *Proceedings of the Third International Conference on Genetic Algorithms*, 1989, pp. 61–69.
29. L. Davis, *Handbook of Genetic Algorithms*, Van Nostrand-Reinhold: Princeton, NJ, 1991.
30. B.A. Julstrom, "What have you done for me lately? Adapting operator probabilities in a steady-state genetic algorithm," in *Proceedings of the Sixth International Conference on Genetic Algorithms*, 1995, pp. 81–87.
31. H.-P. Schwefel, *Evolution and Optimum Seeking*, John Wiley & Sons: New York, 1995.
32. J.T. Stanczak, J.J. Mulawka, and B.K. Verma, "Genetic algorithms with adaptive probabilities of operators selection," in *Proceedings of the Third International Conference on Computational Intelligence and Multimedia Applications*, 1999, pp. 464–468.
33. G. Syswerda, "Uniform crossover in genetic algorithms," in *Proceedings of the Third International Conference on Genetic Algorithms*, 1989.
34. W.M. Spears and K.A. Dejong, "An analysis of multipoint crossover," *Foundations of Genetic Algorithms*, vol. 2, pp. 301–315, 1991.
35. I. Ono, H. Kita, and S. Kobayashi, "A robust real-coded genetic algorithm using unimodal normal distribution crossover augmented by uniform crossover: Effects of self-adaptation of crossover probabilities," in *Proceedings of Genetic and Evolutionary Computation Conference*, 1999, pp. 496–503.
36. W.M. Spears, "Adapting crossover in evolutionary algorithms," in *Proceedings of the Fourth Annual Evolutionary Programming Conference*, 1995, pp. 367–384.
37. A. Tuson and P. Ross, "Cost based operator rate adaptation: An investigation," in *Proceedings of Parallel Problem Solving from Nature Conference*, 1996, pp. 461–469.