An improvement of the standard genetic algorithm fighting premature convergence in continuous optimization

Siarry Patrick

Advances in Engineering Software

ELSEVIER

Advances in Engineering Software 32 (2001) 49–60

ADVANCES IN ENGINEERING SOFTWARE

www.elsevier.com/locate/advengsoft

An improvement of the standard genetic algorithm fighting premature convergence in continuous optimization

J. Andre[a], P. Siarry[b,*], T. Dognon[a]

[a]*E.S.I.E.E., Département des Sciences Mathématiques et Physiques, 2 boulevard Blaise Pascal, 93162 Noisy-le-Grand, France*
[b]*Fac. des Sciences et de Technologie, Université de Paris 12, Laboratoire d'Etude et de Recherche en Instrumentation Signaux et Systèmes, 61 avenue du Général de Gaulle, 94010 Créteil, France*

Received 24 November 1998; received in revised form 24 February 2000; accepted 13 March 2000

Abstract

This paper discusses the trade-off between accuracy, reliability and computing time in the binary-encoded genetic algorithm (GA) used for global optimization over continuous variables. An experimental study is performed on a large set of analytical test functions. We show first the limitations of the "standard GA", which mostly requires a high computing time, though exhibiting a low success rate, due to premature convergence. We then point out the disappointing effect of carefully choosing and tuning the "classical" GA parameters, such as the code and mutation or crossover operators. Indeed, Gray coding and double crossover helped improving on speed, but did not answer the problem of a too homogeneous population. To fight the premature convergence of GA, we emphasize at last two deciding alterations made to the algorithm: an adaptive reduction of the definition interval of each variable and the use of a scale factor in the calculation of the crossover probabilities. The enhanced GA so achieved is discussed in detail and intensively tested on more than 20 test functions of 1–20 variables. © 2000 Elsevier Science Ltd. All rights reserved.

Keywords: Binary-encoded genetic algorithm; Global optimization; Gray coding

1. Introduction

A genetic algorithm (GA) may be described as a mechanism that mimics the genetic evolution of a species. The underlying principles of GAs were first exposed by Holland [1]. An overview about GAs and their implementation in various fields was given by Goldberg [2]. GAs were successfully applied mainly to combinatorial problems (see for instance Refs. [3,5]).

When searching for the global optimum of complex problems, especially for problems with many local optima, traditional optimization methods fail to provide efficiently reliable results. De Jong showed that the standard binary-encoded GA could constitute an interesting alternative to perform the global minimization of a function—called the objective function—depending on several continuous variables [4]. Unfortunately, being faced with the usual conflict between accuracy, reliability and computation time, this type of GA often results in an unsatisfactory compromise, characterized by a slow convergence and a lack of accuracy, when an exact solution is required.

Various approaches have been proposed to overcome these difficulties relating to binary-encoding for real-valued optimization. In particular, a new version of GA has been recently developed [6,7], consisting of using real-valued encodings of the parameters to be optimized. This "real-coded GA" offers the advantages of speeding up the search and making the development of approaches "hybridized" with other methods easier [8]. But it requires the development of new appropriate operators, and to date it lacks mathematical foundations, currently established only for "discrete GA".

In the work presented here, we propose to enhance the efficiency of binary-encoded GAs devoted to the global optimization of complex continuous real-world problems. Our task was achieved empirically using a large panel of more than 20 published analytical test functions of which global and local minima are known.

Owing to their "implicit parallelism" GAs potentially are well adapted to the search for a set of optima when handling multimodal problems. However, the purpose of the algorithm studied in this paper is restricted to the search for the global optimum of an objective function, which allows

* Corresponding author. Tel.: + 33-1-4517-1567; fax: + 33-1-4517-1492.
E-mail address: siarry@univ-paris12.fr (P. Siarry).

a further comparison with other "metaheuristics", such as Simulated Annealing (SA) and Tabu Search (TS). A multimodal version of our GA could be elaborated to solve multiple optima problems.

The paper is divided into three more sections. In Section 2, we first recall the general structure of a standard binary-encoded GA; we then define the criteria used to evaluate the performances of an optimization algorithm; at last we present experimental results, which point out the difficulties of the standard GA relating to global optimization. In Section 3, we propose some improvements. We first evaluate the limited effects of changing the type of the binary code used, and perfecting mutation and crossover operators. To efficiently fight premature convergence of binary-encoded GAs, we then introduce two significant adjustments to the program: an adaptive reduction of the definition interval of each variable and the use of a scale factor in the calculation of the crossover probabilities. In Section 4 the relevant characteristics of the enhanced GA are outlined in short.

2. Description and experimental study of a standard genetic algorithm

The notions developed in this section form the basis of the GA, on which we had to work to increase the reliability of our program. This basis sets the general structure of a GA [2], and we thought it might be helpful to describe them in slight detail, for this paper to be as clear as possible.

We can first point out four important particularities:

- GA uses a code of parameters instead of the parameters.
- GA works on a population of points and not on one single point.
- GA uses probabilistic evolution rules to obtain a satisfying result, according to the laws of natural evolution.
- GA only uses simple operations on the value of the objective function: no complex calculation, like derivation, is necessary.

Knowing this, we can now say that a standard GA is organized around the following elements:

- a code of the population elements;
- a set of genetic operators—Selection, Crossover and Mutation.

We shall now describe those different elements.

2.1. Coding a population

One of the main characteristics of GA is the use of a population of points initially randomly chosen, evolving to comprise the final optimal point. Each point has several components, each of them varying in a specified interval. So one representation of point X could be:

$$X = x_1 x_2 \dots x_k$$

where x_i represents the ith component of X.

Each component is a number that can be coded (for example according to its binary value), if we quantify the definition interval of this variable. If the codes of the different variables are concatenated, X finally becomes a series of bits (a binary string) that can be compared to a "chromosome", each bit representing a "gene" [2,9].

So, to choose an initial population of N points means to generate randomly N binary strings.

Supposing that the binary code has a length of M bits, the process of creating an initial population can be described in the following way:

- Generate k random real numbers (the k components of one point).
- Code each of those k numbers with a M-bit length binary code.
- Repeat those operations N times.

2.2. Selection

Once the initial population is created, we have to define the rules according to which it will be possible to evolve, with a high probability, towards the desired result (the global optimum of the function). The first of those operators is Selection.

The principle of the GA is simple: the best points in the population (i.e. the ones which are the closest to the optimum) have to survive and to reproduce themselves more than the worst points. This is the purpose of the Selection operator.

Several types of Selection operators have been described in the literature. The two most popular ones are the proportional selection and the tournament selection. In this work, we used the selection operator implemented in the standard GA, namely the "roulette wheel selection", which is the simplest form of the proportional selection. This selection technique is briefly described below.

According to the value given by the objective function, each point in the population is assigned a probability of reproduction [4]. The closest point to the optimum must have a higher probability of reproduction than the others. Supposing that we always look for a minimum of the objective function (simple modifications can always turn the problem into such a search), the probability of reproduction for each point is given by:

$$P(X) = F(X) \Big/ \sum_{i \in I} F(X_i)$$

where $F(X) = -(F_{\text{obj}}(X) - \max_{i \in I}(F_{\text{obj}}(X_i)) + \epsilon)$, I the set of points of the population and ϵ a low positive value allowing a no null probability to the worst point.

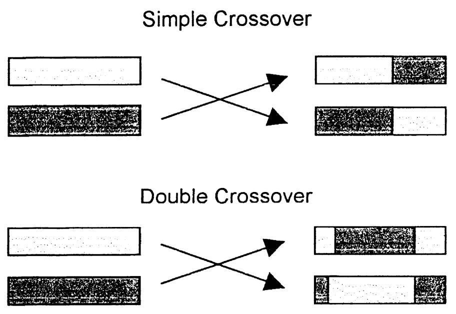

Fig. 1. Simple and double crossover.

We can notice that P is a law of probability ($\sum_{i \in I} P(X_i) = 1$). According to this method, the best points will have the higher probability of being chosen.

One easy way to understand the process of reproduction in the standard GA is to imagine a 'roulette wheel' [10], where each chain of the population occupies a slice proportional to its adaptation (i.e. the probability calculated before). To know which chains are going to reproduce, you just have to spin the wheel as many times as there are individuals in the population. Then each chain is selected a certain number of times and allowed to participate to the crossover operation the same number of times. This process is explained by Goldberg in Ref. [2].

To reduce the stochastic errors associated with the roulette wheel selection, several researchers proposed alternative selection schemes, such as "remainder stochastic sampling" and "stochastic sampling", both of them without or with replacement (see Ref. [2] for more details). We did not implement these techniques to keep as close as possible to the standard GA.

2.3. Crossover

Once the population is created and the best elements selected, we need an operator to make this population evolve. This is the role of the Crossover operator.

After the Selection operation, each element is given a probability of being chosen. The crossover operator chooses two strings according to those probabilities. These two "parents" will give birth to two "children" strings, made of one part of parent1's string and one part of parent2's string. An illustration helps to understand this process (see Fig. 1).

This is called the simple crossover, because the crossing point is unique. Other kinds of crossover operators can be used (double crossover, generalized crossover,…), as we will see later on.

The new strings have new characteristics and will be added to the population. Selection and crossover are then repeated until the size of the population reaches a fixed maximum. At this point, the weakest elements are removed from the population, until it gets back to its initial size. The new population shows better genetic characteristics than the former one, in other words it contains elements that are closer to the optimum.

The three operations we have just described are the basic components of a standard genetic algorithm. This is the first version we have tested on a series of functions, in order to have an element of comparison for the future modifications we would make. Before analyzing the results, we have to say some words on the criteria we used to evaluate the performances of this standard GA.

2.4. Mutation

The principle of mutation developed in Ref. [11] is simple: it consists of the modification of one gene (i.e. one bit of the string). For each element of the population, the probability of mutation, P_m, is tested. If a randomly selected number is superior to a fixed threshold (i.e. if the condition is true), one bit is chosen at random and its value is changed (0 becomes 1 for example). As for crossover, we can imagine other kinds of mutation operators, for example on several genes (one for each component), or only on localized genes on the string, depending on the effects we want to observe.

2.5. Criteria of evaluation

Three main criteria appeared to be really important when trying to determine the performances of an algorithm: convergence, speed and robustness. To clarify the analysis we have done below, we have to define what these words mean to us.

2.5.1. Convergence

The aim of this study is the search for the global optimum of a function. So it is of major importance that our GA effectively converges to this optimum! To do so, we must avoid premature convergence in a local optimum, as well as give a final result precise enough to be used.

So the test for convergence we used is the evaluation of the error between the result given by the algorithm and the exact value of the optimum, for each test function. The relative error was used each time it was possible:

$$E_{\text{rel}} = \frac{|X_{\text{algo}} - X_{\text{exact}}|}{X_{\text{exact}}}$$

When the optimum to reach was 0, it was no longer possible to use this expression so we calculated the absolute error:

$$E_{\text{abs}} = |X_{\text{algo}} - X_{\text{exact}}|.$$

2.5.2. Speed

This second criterion tests the speed with which the optimum is obtained. For the GA to be competitive, it is important to be able to compare its speed with the other existing methods. As time is a relative notion depending on the

Table 1
Results with the standard AG

Name of the function	Number of variables	Theoretical minimum	Minimum found		Relative error average %	Absolute error average	Number of evaluation of the function	Success %
			Min	Max				
F1	1	−1.12323	−1.12323	−1.2153	0.01	0.000	5566	100
F3	1	−12.03125	−12.03120	−11.95370	0.01	0.001	5347	100
Branin	2	0.39789	0.39789	0.43755	0.76	0.003	8125	81
Camelback	2	−1.03163	−1.03163	−1.02138	0.44	0.005	1316	98
Goldprice	2	3	3.00000	13.06440	7.62	0.229	8185	59
PShubert1	2	−186.73091	−186.73000	−147.11800	2.44	4.563	7192	63
PShubert2	2	−186.73091	−186.73100	−137.82700	2.56	4.772	7303	59
Quartic	2	−0.35239	−0.35239	−0.29590	0.94	0.003	8181	83
Shubert	2	−186.73091	−186.73100	−175.90800	0.32	0.596	6976	93
Hartman1	3	−3.86278	−3.86249	−3.76495	0.64	0.025	1993	94
Shekel1	4	−10.15320	−10.13490	−1.42461	59.76	6.067	7495	1
Shekel2	4	−10.40294	−10.16770	−1.60922	46.68	4.856	8452	0
Shekel3	4	−10.53641	−10.40340	−1.82221	48.65	5.126	8521	0
Hartman2	6	−3.32237	−3.30652	−2.51791	4.33	0.144	19452	23
Hosc45	10	1	1.99506	2.00000	99.98	1.000	11140	0
Brown1	20	2	43.62810	465.51300	9520.59	190.412	6844	0
Brown3	20	0	1.30600	16.75060	–	6.072	8410	5
Chainsing	20	0	214.82600	39011.40000	–	5962.169	7084	4
F5n	20	0	0.47353	9.93800	–	2.583	8725	0
F10n	20	0	7.83515	91.35410	–	33.518	9298	0
F15n	20	0	0.52117	7.82703	–	2.253	9541	0

computer used, we preferred to measure the speed by counting the number of evaluations of the tested function. On the one hand, this criterion is independent from the type of computer used, and on the other hand the evaluation of the function is the operation taking the longest time in the program.

2.5.3. Robustness

By this we mean the adaptation of the algorithm to the widest range of functions. The less the algorithm is specialized, the better it is, that is the reason why we chose a large number of test functions, representative for the various problems that may be treated in reality.

2.6. Experimental results

For each of these criteria, and for each evolution of our algorithm, the tests were carried out in a statistic method: 100 tests for each function, so that the result found can be reliable.

We can notice that a very important problem of optimization algorithms may be the memory used. In our case, the population tested does not exceed 400 individuals per generation, so the size of the memory used is reduced and does not appear as a major problem.

The results obtained with the standard GA are given in Table 1. All the test functions used are listed in Appendix A. The algorithm parameters were fixed as follows: population size equal to 200; crossover rate equal to 100%; no mutation operator implemented. These tunings were obtained empirically through a set of preliminary experiments handling all the test functions. Of course, all results discussed below were obtained while keeping these GA parameters constant. It can be pointed out that the retained tunings are near from that usually given in the literature, like in Ref. [2]. In particular, a high crossover rate allows us to reduce the CPU time and the role of mutation is generally weak.

The criterion of success is the percentage of trials (out of the 100 tests for each function) that managed to reach the global optimum within 500 iterations of the algorithm. This means that the optimum found differed of less than 1% from the exact optimum in relative error (or 0.1 in absolute error, if the optimum was 0).

The results are contrasted, for some functions the optimum is reached almost every time, and for others almost never. When looking at the failing functions more carefully, we can notice that most of the time the problem is due to premature convergence, the algorithm being blocked in a local optimum without being able to get out of it. Most of the time, these functions appear to be multimodal ones, showing a lot of sharp peaks, and the population after a few iterations only contains identical elements in a wrong valley. Therefore, it is impossible for the algorithm to make a "jump" to another valley, and the global minimum can never be reached because of the lack of diversity of the population after a few iterations. So new ideas have to be implemented in order to overcome those difficulties, and the first one is the creation of a mutation operator.

All the studies carried out on the mutation operator show that its role is secondary in the evolution of a GA. Nevertheless, the introduction of mutation in the genetic process can be of some interest for the two following reasons: first, a model including mutation is closer to the real genetic process, as long as the probability of mutation, P_m, remains low (0.1% typ.). Moreover, it can constitute a remedy against an important problem we met with the standard algorithm: premature convergence. Indeed, the mechanisms of reproduction and crossover can sometimes be too efficient, and the population soon contains only identical strings. So the evolution is stopped whatever far the optimum is, for the mixing of the strings has no more influence on the characteristics of the population. This can be a problem if the algorithm reaches a local optimum and cannot move away from it. By introducing randomly new genetic information, mutation gives a solution to this major problem of premature convergence.

The effects of mutation on the evolution of the population are not significant, as we have already said. The tests performed with the simple mutation (modification of one bit of the string) show no amelioration, but no deterioration either.

Other modifications and improvements of the standard GA will be developed further, concerning the speed and the reliability of our algorithm. The number of evaluations given by the standard algorithm will therefore be a major point of comparison for the future algorithms, to check how the improvement obtained relating to the convergence criterion influences the speed of the algorithm.

3. Improvements of the standard GA

Considering the results obtained with the standard GA, we can point out a major problem: the low success rate, specially for the highly multimodal functions, is due to premature convergence, which is paradoxically a consequence of the performance of the algorithm. Indeed, the role of a GA is to make an initial population of points evolve to a population of 'perfect' points, presenting the same genetic characteristics: the code of the global optimum. But in several cases, this evolution is too fast and the final population does not contain the global optimum (it has focused on a local one, and the possibilities to 'jump' into another valley are limited, because of a too homogeneous population). Another problem is the lack of stability of the standard GA when treating with functions presenting narrow valleys.

The aim of our search was, therefore, to avoid those two main problems, and several modifications have been tested to eliminate them.

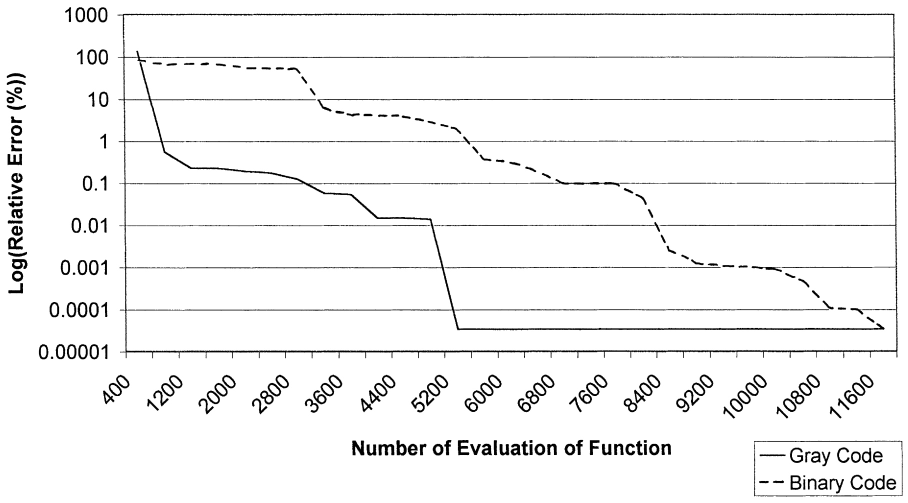

Fig. 2. Function Goldprice—speed increase.

3.1. Changing the code

The first stage of the creation of a GA is the choice of a code to represent the parameters. The code we used above in the standard algorithm was the natural binary code. Following the work of Hollstien [12], we tried to point out the interest of using a Gray code instead, presuming that the characteristics of such a code (adjacent integers only differ of one bit) would improve the convergence speed.

The results obtained (Fig. 2) show the increase in the speed of the algorithm, but no significant improvement is achieved concerning robustness: indeed, the homogeneity of the population is not altered by the use of such a code.

3.2. Tests on mutation

One way to prevent the population becoming too homogeneous could be to improve the Mutation operator. Tests carried out by De Jong [4] showed that increasing the probability of mutation (fixed at the beginning of the program) had bad effects on the convergence of the algorithm: the higher this probability is, the closer we are to a purely random search. Therefore, we chose to work on the nature of the mutation operator rather than on the probability. Two types of mutations were implemented.

1. *A 'multiple' mutation:* each gene of the string is tested for mutation. This could allow greater 'jumps' to other valleys.
2. *A localized mutation:* when a point is chosen for mutation, the change only affects the lowest bits of each component. This type of mutation could help entering narrow valleys by slightly changing the position of the points.

The results obtained show no major amelioration, confirming the secondary role of the mutation operator in GAs.

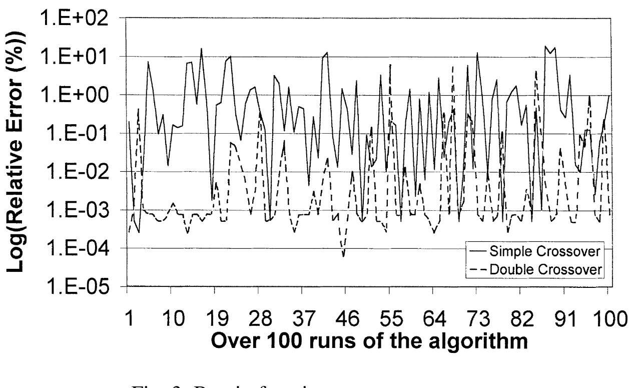

Fig. 3. Branin function—test on crossover.

3.3. Tests on crossover

Another way to improve the performances of our algorithm is to modify the crossover operator. Instead of mixing the strings at one point, we tried to implement a double crossover, where the strings of the parents were mixed twice in each component of the child's code. The results (Fig. 3) did not show any amelioration as far as premature convergence is concerned. But when considering the speed criterion, we observed two important issues:

- for the functions in which the former algorithm was reaching the global optimum, the new algorithm reached the same optimum faster with a higher precision;
- for the functions in which the former algorithm failed and returned a local optimum, the new algorithm found the same optimum, but faster once again.

So the work on the traditional operators of GAs had a positive influence on the performances of our algorithm: Gray coding and double crossover helped improving on speed, which was one of the major evaluation criteria we have already defined. But the main problem is still

Table 2
Results with the improved AG

Name of the function	Number of variables	Theoretical minimum	Minimum found		Relative error average %	Absolute error average	Number of evaluation of the function	Success %
			Min	Max				
F1	1	−1.12323	−1.12323	−1.11390	0.03	0.000	784	100
F3	1	−12.03125	−12.03120	−11.92697	0.12	0.014	744	100
Branin	2	0.39789	0.39791	0.40183	0.48	0.002	2040	100
Camelback	2	−1.03163	−1.03163	−1.02138	0.44	0.005	1316	100
Goldprice	2	3	3.00028	3.02959	0.43	0.013	4632	100
PShubert1	2	−186.73091	−186.68574	−184.95544	0.53	0.983	8853	100
PShubert2	2	−186.73091	−186.70469	−184.92950	0.53	0.986	4116	100
Quartic	2	−0.35239	−0.35238	0.34887	0.46	0.002	3168	100
Shubert	2	−186.73091	−186.72802	−184.87532	0.49	0.921	2364	100
Hartman1	3	−3.86278	−3.86114	−3.82457	0.51	0.020	1680	100
Shekel1	4	−10.15320	−10.14866	−9.69071	0.71	0.072	36388	97
Shekel2	4	−10.40294	−10.38253	−5.12876	1.58	0.165	36774	98
Shekel3	4	−10.53641	−10.51404	−10.43177	0.70	0.074	36772	100
Hartman2	6	−3.32237	−3.31383	−3.20000	1.01	0.033	53792	92
Hosc45	10	1	1.00943	1.74531	39.21	0.392	126139	2
Brown1	20	2	8.55162	111.29142	2692.67	53.853	128644	0
Brown3	20	0	0.67464	5.91223	–	2.324	106859	5
Chainsing	20	0	0.64463	62.53152	–	10.293	104725	4
F5n	20	0	0.00221	0.59064	–	0.067	99945	100
F10n	20	0	0.04960	4.06598	–	1.197	113929	49
F15n	20	0	0.00342	0.73614	–	0.075	102413	100

Table 3
Global optimization methods used for performance analysis

Method	Name	Reference
PRS	Pure random search	[14]
MS	Multistart	[15]
SD	Simulated diffusion	[16]
SA	Simulated annealing	[17]
TS	Tabu search	[18]
GA	Binary-encoded genetic algorithm	This work

remaining: when considering robustness, the results in Table 1 showed that a standard GA is not adapted to a large panel of functions, despite the modifications we made. In fact, all those modifications did not answer the problem of a too homogeneous population.

3.4. Fighting premature convergence

After those tests on classical operators, and noting that no progress was made in robustness, we directed our search into other domains, to solve the problem of premature convergence.

3.4.1. Changing the study interval

The idea is close to a dichotomy search: each time the algorithm has done a crossover operation, it finds an optimum and gets rid of the worst individuals in the population. So each time we get this 'transition' optimum, we can reduce the definition interval of each component around this optimum, without changing the code, just by modifying the application which associates a value to each code. Doing this, we get a more precise value for each component (a code of 32 bits corresponds to approximately 4 billion points in each interval, so the smaller the domain is, the more precise is the value).

The process consists in letting the algorithm run 50 times (50 crossovers) and collect then the 10 best 'transition' optima found. Then the domain is reduced around those 10 points and the algorithm is run 50 more times, and so on (the total number of runs should be limited, 500 runs for example).

But even so, we cannot be sure that we found the right valley: so if the optimum found after 50 runs is not better than the former one, the domains of each component are reset to their initial width.

Another problem is that, after 50 runs, the population obtained is very homogeneous, and as the codes of the individuals are not changed, the distribution of the points in the new interval is similarly homogeneous. To avoid this phenomenon (and therefore to prevent premature convergence), the population is reset each time the interval is changed (a new population is created) to ensure a random distribution in the new domain.

This technique increases the performance of our algorithm in terms of robustness, for it allows a better precision in the value of the optimum, and avoids oscillations around narrow valleys. The counterpart is a reduced speed in some cases, because of the possible reinitializations of the population. But the problem of premature convergence is not completely eliminated.

3.4.2. Scale factor

The problem of premature convergence comes from the fact that, after a few runs of the algorithm, the population tends to be too homogeneous, and the action of crossover cannot be sensible anymore, so, if the optimum found is not the global one, the algorithm fails. To avoid this phenomenon, we introduced the use of a scale factor (SF) [13] in the calculation of the crossover probabilities: in the first iterations, the best individuals get a lower probability than they should, and the worst ones see their probability increased. The resulting population is maintained to be more heterogeneous at the beginning of the algorithm, which decreases the risks to be trapped into a local optimum. Then, in the last iterations, the scale factor is increased to make sure that the algorithm effectively converges ($SF = 1$ in the last iteration).

The results presented in Table 2 with both the Scale Factor and the change of study interval show the best performances. We can first notice that the success rate is improved in almost every case. When comparing it with the one obtained with the standard GA (Table 1), three categories of functions appear quite clearly.

In a first group, we could place the functions for which the global optimum was already reached by the former algorithm, with a rather satisfying rate (over 80%). In these cases, if the success rate is improved (all of these functions show a 100% success rate in Table 2), we can also point out that the speed is increased significantly: the examples of the F1, F3, Branin or Quartic functions are representative of this trend (the number of function evaluations is at least divided by 2). It is also important to observe the relative error average rate for these functions, which tends to be closer to zero with the improved GA.

A second group could be the one of the functions presenting unsatisfying rates with the standard GA (between 0 and 60% in Table 1), and of which the results have been turned "good" by the improved algorithm. For these functions, we can notice a spectacular "jump" in the success rate (from 0 to 100% success for the F5n and F15n functions, for example), but the major drawback is a much slower speed: the number of evaluations is sometimes multiplied by more than 10.

In a last group we can put the unsuccessful functions, which all have in common a high number of variables (between 10 and 20). Despite the lack of success in finding the global optimum, we can observe a decreasing of the error rate, always balanced by an increasing number of evaluations (the example of the Hosc45 function is significant: the relative error rate has been divided by almost 3, when the number of evaluations has been multiplied by 10).

Table 4
Number of function evaluations in global optimization of five functions by the six different methods defined in Table 3

Method	Function				
	Goldprice	Branin	Hartman1	Hartman2	Shubert
PRS	5125	4850	5280	18 090	6700
MS	4400	1600	2500	6000	–
SA1	5439	2700	3416	3975	241215
SA2	563	505	1459	4648	780
TS	486	492	508	2845	727
AG	4632	2040	1680	53792	2364

For those functions, our improvement of the standard GA has a strong effect, but not strong enough to be considered as successful.

3.5. Comparison with other methods of iterative improvement

Lastly we have performed a comparison of our GA with five other methods of iterative improvement listed in Table 3: Pure Random Search (PRS), Multistart (MS), Simulated Diffusion (SD), SA and TS. The efficiency was quantified in terms of the number of function evaluations necessary to find the global optimum. Each program was stopped as soon as the relative error between the best point found and the known global optimum was less than 1%. The numbers of function evaluations used by the various algorithms to optimize five test functions are listed in Table 4. It can be pointed out that we did not program ourselves the competitive algorithms used for the comparison, but we reported the results published by Cvijovic and Klinowski [18]. We can see that results achieved with our GA are satisfactory, except for Hartman 2. Not surprisingly, our numbers of evaluations are higher than that of SA and TS: GAs generally are more expensive in CPU time, but potentially capable of yielding a complete set of optima when one is faced with multimodal problems. In addition our results were the average outcome of 100 independent runs; for some published methods, the number of runs was equal to 4 or unspecified.

4. Conclusion

The two main changes made in the enhanced GA—the introduction of the Scale Factor and the adaptive study interval—have a direct influence on two of the major evaluation criteria: speed and convergence. First of all, we noticed a far better convergence for a larger number of functions, which was our primary objective in improving the algorithm. But with regard to speed, two opposite effects have been pointed out: whenever dealing with functions that already had a satisfying convergence rate, speed is increased in a sensible way. Indeed, the examples of the F1 and F3 functions are representative for this tendency, the number of evaluations of the function being in both cases divided by more than 7. On the contrary, when dealing with functions presenting unsatisfying rates, speed is decreased significantly. This event can be easily explained by the nature of the changes we made: when convergence is not a problem, the change of the study interval reduces the search area and the optimum is found more easily, so the number of evaluations is decreased. But when convergence is hard, the Scale Factor slows the process (its role is to avoid premature convergence) and more, there might be several tries of study interval before finding the "good" valley.

So the search for a more robust algorithm has a drawback, as speed can also be a major problem when using common computing resources. A choice has to be made between a wide range of functions and a reasonable calculation time.

Appendix A. List of test functions used

- F1:

$$f(x) = 2(x - 0.75)^2 + \sin(5\pi x - 0.4\pi) - 0.125$$

where $0 \le x \le 1$

- F3:

$$f(x) = -\sum_{j=1}^{5} [j \sin[(j+1)x + j]]$$

where $-10 \le x \le 10$

- Branin:

$$f(x,y) = a(y - bx^2 + cx - d)^2 + h(1-f)\cos(x) + h$$

where $a = 1,\ b = 5.1/4\pi^2,\ c = 5/\pi,\ d = 6,\ h = 10,\ f = 1/8\pi,\ -5 \le x \le 10\ \ 0 \le y \le 15$

- Camelback:

$$f(x,y) = \left(4 - 2.1x^2 + \frac{x^4}{3}\right)x^2 + xy + (-4 + 4y^2)y^2$$

where $-3 \le x \le 3\ \ -2 \le y \le 2$

- Goldprice:

$$f(x,y)$$

$$= [1 + (x + y + 1)^2(19 - 14x + 3x^2 - 14y + 6xy + 3y^2)].$$

$$[30 + (2x - 3y)^2(18 - 32x + 12x^2 + 48y - 36xy + 27y^2)]$$

where $-2 \le x \le 2 \quad -2 \le y \le 2$

- Pshubert 1 and 2:

$$f(x,y) = \left\{\sum_{i=1}^{5} i\cos[(i+1)x + i]\right\} \times \left\{\sum_{i=1}^{5} i\cos[(i+1)y + i]\right\} + \beta[(x + 1.42513)^2 + (y + 0.80032)^2]$$

where $-10 \le x \le 10 \quad -10 \le y \le 10 \quad \beta = 0.5$ for Pshubert1, $\beta = 1$ for Pshubert2

- Quartic:

$$f(x,y) = \frac{x^4}{4} - \frac{x^2}{2} + \frac{x}{10} + \frac{y^2}{2}$$

where $-10 \le x \le 10 \quad -10 \le y \le 10$

- Shubert:

$$f(x,y) = \left\{\sum_{i=1}^{5} i\cos[(i+1)x + i]\right\}\left\{\sum_{i=1}^{5} i\cos[(i+1)y + i]\right\}$$

where $-10 \le x \le 10 \quad -10 \le y \le 10$

- Hartman 1:

$$f(x_1, x_2, x_3) = -\sum_{i=1}^{4} c_i \exp - \sum_{j=1}^{3} a_n(x_i - p_{ij})^2$$

where $0 \le x_i \le 1$ for $i = 1, \ldots, 3$ with $x = (x_1, \ldots, x_3)$,

$p_i = (p_{i1}, \ldots, p_{i3}),\ a_i = (a_{i1}, \ldots, a_{i3})$.

I	a_n			c_i	p_n		
1	3.0	10.0	30.0	1.0	0.36890	0.1170	0.2673
2	0.1	10.0	35.0	1.2	0.46990	0.4387	0.7470
3	3.0	10.0	30.0	3.0	0.10910	0.8732	0.5547
4	0.1	10.0	35.0	3.2	0.03815	0.5743	0.8828

- Shekel 1, 2 and 3:

$$f(x) = -\sum_{i=1}^{m} \frac{1}{(x - a_i)^{\mathrm{T}}(x - a_i) + c_i}$$

where $0 \le x_j \le 10$ with $m = 5$ for Shekel 1, $m = 7$ for Shekel 2, $m = 10$ for Shekel 3 and $x = (x_1, x_2, x_3, x_4)^{\mathrm{T}}$, $a_i = (a_{i1}, a_{i2}, a_{i3}, a_{i4})^{\mathrm{T}}$.

i	a_i				c_i
1	4.0	4.0	4.0	4.0	0.1
2	1.0	1.0	1.0	1.0	0.2
3	8.0	8.0	8.0	8.0	0.2
4	6.0	6.0	6.0	6.0	0.4
5	3.0	7.0	3.0	7.0	0.4
6	2.0	9.0	2.0	9.0	0.6
7	5.0	5.0	3.0	3.0	0.6
8	8.0	1.0	8.0	1.0	0.7
9	6.0	2.0	6.0	2.0	0.5
10	7.0	3.6	7.0	3.6	0.5

- Hartman 2:

$$f(x_1, x_2, x_3) = -\sum_{i=1}^{4} c_i \exp - \sum_{j=1}^{3} a_n(x_i - p_{ij})^2$$

where $0 \le x_i \le 1$ for $i = 1, \ldots, 3$ with

$x = (x_1, \ldots, x_6), \qquad p_i = (p_{i1}, \ldots p_{i6}),$

$a_i = (a_{i1}, \ldots, a_{i6})$.

i	a_{ij}						c_i
1	10.00	3.00	17.00	3.50	1.70	8.00	1.0
2	0.05	10.00	17.00	0.10	8.00	14.00	1.2
3	3.00	3.50	1.70	10.00	17.00	8.00	3.0
4	17.00	8.00	0.05	10.00	0.01	14.00	32

i	p_{ij}					
1	0.1312	0.1696	0.5569	0.0124	0.8283	0.5886
2	0.2329	0.4135	0.8307	0.3736	0.1004	0.9991
3	0.2348	0.1451	0.3522	0.2883	0.3047	0.6650
4	0.4047	0.8828	0.8732	0.5743	0.1091	0.0381

- Hosc45:

$$f(x) = 2 - \prod_{i=1}^{10} \frac{x_i}{n!}$$

where $x = (x_1, \ldots, x_{10})$ and $0 \le x_i \le i$

- Brown1:

$$f(x) = \left[\sum_{j\in J}(x_i - 3)\right]^2 + \sum_{j\in J}[10^{-3}(x_i - 3)^2 - (x_i - x_{i+1}) + e^{20(x_i - x_{i+1})}]$$

where $J = \{1, 3, \ldots, 19\}$

$-1 \le x_i \le 4$ for $1 \le i \le 20$ and $x = [x_1, \ldots, x_{20}]^T$

- Brown3:

$$f(x) = \sum_{i=1}^{19}[(x_i^2)^{(x_{i+1}^2+1)} + (x_{i+1}^2)^{(x_i^2+1)}]$$

where $x = [x_1, \ldots, x_{20}]^T$ and $-1 \le x_i \le 4$ for $1 \le i \le 20$

- Chainsing:

$$f(x) = \left[\sum_{j\in J}(x_i - 3)\right]^2 + \sum_{j\in J}[10^{-3}(x_i - 3)^2 - (x_i - x_{i+1}) + e^{20(x_i - x_{i+1})}]$$

where $J = \{1, 3, \ldots 19\}$

$-1 \le x_i \le 4$ for $1 \le i \le 20$ and $x = [x_1, \ldots, x_{20}]^T$

- Chainwood and Genwood:

$$f(x) = 1 + \sum_{j\in J}[100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2 + 90(x_{i+3} - x_{i+2}^2)^2 + (1 - x_{i+2})^2 + 10(x_{i+1} + x_{i+3} - 2)^2 + 0.1(x_{i+1} - x_{i+3})^2]$$

where $x = [x_1, \ldots, x_{20}]^T$ and $\begin{cases} J = \{1, 3, \ldots, 17\} \text{ for Chainwood} \\ J = \{1, 5, \ldots, 17\} \text{ for Genwood} \end{cases}$

- Degensing:

$$f(x) = \sum_{j\in J}[(x_i + 10x_{i+1})^2 + 5(x_{i+2} - x_{i+3})^2 + (x_{i+1} - 2x_{i+2})^4 + 10(x_i - 10x_{i+3})^4]$$

where $x = [x_1, \ldots, x_{20}]^T$ and $J = \{1, 5, \ldots, 17\}$ $\begin{cases} x_i \le 0 \text{ for all } i \text{ such that } i \bmod 3 = 0 \text{ and } i \bmod 4 = 2 \\ x_i \ge 0 \text{ for all } i \text{ such that } i \bmod 3 = 0 \text{ and } i \bmod 4 \ne 2 \end{cases}$

- F5n:

$$f(x) = (\pi/20)\times\left\{10\sin^2(\pi y_1) + \sum_{i=1}^{19}[(yi - 1)^2 \times(1 + 10\sin^2(\pi y_{i+1}))] + (y_{20} - 1)^2\right\}$$

where $x = [x_1, \ldots, x_{20}]^T$ and $-10 \le x_i \le 10$ $yi = 1 + 0.25(x_i - 1)$

- F10n:

$$f(x) = (\pi/20)\left\{10\sin^2(\pi x_1) + \sum_{i=1}^{19}[(x_i - 1)^2 \times(1 + 10\sin^2(\pi x_{i+1}))] + (x_{20} - 1)^2\right\}$$

where $x = [x_1, \ldots, x_{20}]^T$ and $-10 \le x_i \le 10$

- F15n:

$$f(x) = (1/10)\left\{\sin^2(3\pi x_1) + \sum_{i=1}^{19}[(x_i - 1)^2(1 + \sin^2(3\pi x_{i+1}))] + (1/10)(x_{20} - 1)^2[1 + \sin^2(2\pi x_{20})]\right\}$$

where $x = [x_1, \ldots, x_{20}]^T$ and $-10 \le x_i \le 10$

- Chainrose and Degenrose:

$$f(x) = 1 + \sum_{i=2}^{25}[4\alpha_i(x_i - x_{i-1}^2)^2 + (1 - x_{i-1})^2]$$

where $x = [x_1, \ldots, x_{25}]^T \begin{cases} -10 \le x_i \le 10 \text{ for Chainrose} \\ xi \le 1 \text{ for Degenrose} \end{cases}$

and $\alpha_i = i + 1$

- Broyden1a and Broyden1b:

$$f(x) = 1 + \sum_{i=1}^{30} |(3 - 2x_i)x_i - x_{i-1} - 2x_{i+1} + 1|^p$$

$$\text{with } x_0 = x_{31} = 0 \text{ where } x = [x_1, \ldots, x_{30}]^{\mathrm{T}}$$

$$\text{and} \begin{cases} p = 7/3 \text{ for Broyden1a} \\ p = 2 \text{ for Broyden1b} \end{cases}.$$

References

[1] Holland JH. Adaptation in natural and artificial systems. University of Michigan, Ann Arbor, MI, Internal report, 1975.

[2] Goldberg DE. Genetic algorithms in search, optimization and machine learning. Reading, MA: Addison-Wesley, 1989.

[3] Grefenstette JJ, editor. Genetic algorithms and their applications Proceedings of the Second International Conference on Genetic Algorithms and their Applications. Hillsdale, NJ: Lawrence Erlbaum, 1987.

[4] De Jong KA. An analysis of the behavior of a class of genetic adaptive systems. Doctoral dissertation, University of Michigan, Ann Arbor, MI. Dissertation abstracts international 1975;36(10):5140B.

[5] Arunkumar S, Chockalingam T. Genetic search algorithms and their randomized operators. Computers Math Applic 1993;35(5):91–100.

[6] Davis L. Handbook of genetic algorithms. New York: Van Nostrand Reinhold, 1991.

[7] Janikow CZ, Michalewicz Z. An experimental comparison of binary and floating point representation in genetic algorithms, Proceedings of the Fourth International Conference on Genetic Algorithms. San Francisco: Morgan Kaufman, 1991 (p. 31–6).

[8] Renders JM, Flasse SP. Hybrid methods using genetic algorithms for global optimization. IEEE Trans Systems, Man Cybernetics—part B: cybernetics 1996;26(2):243–58.

[9] Koza JR. Genetic programming, on the programming of computers by means of natural selection. Cambridge: MIT Press, 1993.

[10] Frantz DR. Non-linearities in genetic adaptive search. Doctoral dissertation, University of Michigan. Dissertation abstracts international 1972;33(11):5240B–5241B.

[11] Foo, Bosworth. Algebraic, geometric and stochastic aspects of genetic operators (CR-2099). Washington, DC: National Aeronautics and Space Administration, Internal report, 1972.

[12] Hollstien RB. Artificial genetic adaptation in computer control systems. Doctoral dissertation, University of Michigan. Dissertation abstracts international 1971;32(3):1510B.

[13] De Jong KA. Adaptive system design: a genetic approach. IEEE Trans Systems, Man Cybernetics 1980;SMC-10(9):566–74.

[14] Anderssen RS. In: Anderssen RS, Jennings LS, Ryan DM, editors. Optimisation, St. Lucia, Australia: University of Queensland Press, 1972. p. 27–34.

[15] Rinnoy Kan AHG, Timmer GT. Stochastic global optimization methods, Part II: Multi-level methods. Mathematical Programming 1987;39:57–78.

[16] Aluffi-Pentini F, Parisi V, Zirilli F. Global optimization and stochastic differential equations. J Opt Theor Appl 1985;47:1–16.

[17] Dekkers A, Aarts E. Mathematical Programming 1991;50:367.

[18] Cvijovic D, Klinowski J. Taboo Search: an approach to the multiple minima problem. Science 1995;267:664–6.