

https://dspace.org/

Some Population Set-Based Methods for Unconstrained Global Optimization

Kaelo, Professor

2006-11-16T08:40:11Z
http://hdl.handle.net/10539/1771

Downloaded from a DSpace repository. https://dspace.org/

Integrated crossover rules in real coded genetic algorithms

P. Kaelo and M. M. Ali
School of Computational and Applied Mathematics,
Witwatersrand University, Wits - 2050, Johannesburg

Abstract

Modifications in crossover rules and localization of searches are suggested to the real-coded genetic algorithms for continuous global optimization. Central to our modifications is the integration of different crossover rules within the genetic algorithm. Numerical experiments using a set of 50 test problems indicate that the resulting algorithms are considerably better than the previous version considered and offer a reasonable alternative to many currently available global optimization algorithms, especially for problems requiring 'direct search type' methods.

Keywords: Genetic algorithms, global optimization, direct search methods, probabilistic adaptation.

1 Introduction

We consider the problem of finding the global minimum of the unconstrained optimization problem

$$\text{minimize } f(x) \text{ subject to } \ x \in \Omega, \tag{1}$$

where $f(x) : \Omega \subset R^n \rightarrow R$ is a continuous real-valued function and x is a n-dimensional continuous variable vector. The search region Ω is assumed to be either a box or some other region easy to sample. A point x_{opt} is said to be a global minimizer of f if $f_{opt} = f(x_{opt}) \leq f(x), \forall x \in \Omega$. In many applications, for example, in applied sciences and engineering, the function of interest may be non-linear, non-convex or non-differentiable. It is with this view in mind that some global optimization methods that do not require much information about the function were developed. They include differential evolution (DE) (Storn and Price, 1997; Kaelo and Ali, 2005), genetic algorithms (GA) (Michalewicz, 1996; Ali and Törn, 2004) and controlled random search (CRS) (Price, 1983; Ali et. al., 1997) amongst others. Unlike gradient based methods, these methods use no properties of the function being optimized. The only requirement on the problem is that $f(x)$ can be computed for any $x \in \Omega$. They are also easy to implement.

In recent years, real coded genetic algorithms have been used to solve continuous optimization problems (Michalewicz, 1996; Herrera and Lozano, 2000; Herrera et. al., 2003; Ali and Törn, 2004). Traditionally, candidate points (chromosomes) in genetic algorithms are represented by binary coded strings. In real coded GA, real-valued genes are used instead of the conventional bit-string encoding, i.e. it treats chromosomes as vectors of real valued numbers. It then adapts the genetic operators of the binary coded GA accordingly. The real-valued genes have shown to offer advantages over the conventional bit-valued genes in that they are better adapted to numerical optimization of continuous problems. They also make GA to be easily hybridized with other methods. However, real coded genetic algorithms often suffer premature convergence due to lack of population diversity. On the other hand, they could also suffer from slow convergence. This is due to the fact that GAs do not exploit local information of points in the population set.

To overcome these problems of slow convergence and premature convergence, some researchers have hybridized GA with other optimization methods, for example, Chelouah and Siarry (2003) hybridized GA with the Nelder-Mead simplex algorithm (Nelder and Mead, 1965). In their hybridization, they use GA to locate a promising search region and then apply the Nelder-Mead algorithm from the best point found within this promising search region. Yun et. al. (2003) used a rough search technique to generate a diverse initial population and then hybridized GA with a local descent technique. A diverse initial population generation using some quasi-random sequences is also suggested in Maaranen et. al. (2004). On the other hand, other researchers have worked on improving the crossover operators of the real coded GA (Tsutsui and Goldberg, 2001; Hrstka and Kučerová, 2004). Population diversity in real coded genetic algorithms is needed throughout the search process–not just at the initial stages. This depends on how the set evolves with each generation. Hence, it depends on the ability of the point generation operators, e.g. the crossover operator, to explore the search region.

Although a number of crossover rules have been suggested in the literature to improve the population diversity, they were tailored towards solving certain problems. For example, Tsutsui and Goldberg (2001) suggested a crossover operator that performs well on functions that have their global minimizers at the boundary of the feasible search region. Also, these modifications were justified using small sets of low dimensional problems. The objective of this paper is to investigate the efficiency and robustness of a real coded GA that was suggested in Michalewicz (1996) and studied by Hu et. al. (1997), and to suggest modifications to improve its robustness and efficiency. To achieve this, we first carried out an extensive numerical study of GA (using a large set of 50 test problems) with various crossover rules, some of which are suggested in the literature. From the numerical study, it was clear that efficiency and robustness of GA are largely problem dependent. Some versions were superior to others on some problems while on other problems the converse statement was true. This means a particular crossover rule favoured some problems over others. Thus, in this paper we propose some real coded GA algorithms that use the complementary strengths of various crossover rules by integrating them suitably. We also propose a scheme that probabilistically adapts crossover rules for a given problem. In addition, we suggest a local technique for exploration of search at early stages and faster convergence at later stages. We demonstrated that the combined effects of these changes diversify the population for exploration of

searches and intensify the searches at later stages.

The organisation of the paper is as follows. In section 2 we briefly describe GA. Section 3 contains the full description of our proposed modifications. In section 4 numerical results are presented and comparisons made. Section 5 contains the concluding remarks based on the results obtained.

2 Brief introduction of real coded genetic algorithms

Genetic algorithm is a direct search algorithm that has attracted a lot of interest in recent years. It is a global optimization technique based on natural selection and the genetic reproduction concept. Like CRS and DE, real coded GA maintains a set S of candidate points. At each generation of GA, the set S is updated by new offsprings obtained through the reproduction (crossover and mutation) process. A simple GA consists of the following four steps:

Algorithm 1: A real coded genetic algorithm

Step 1 **Initialize**. Generate N uniformly distributed random points from the search region Ω and store the points and their corresponding function values in S.

Step 2 **While not stopping condition**. Do Steps 3 and 4.

Step 3 **Generate offsprings**

- Selection: select $m \leq N$ points from S as parents.
- Crossover: pair the parents and create m new points (offsprings).
- Mutation: mutate an element (gene) of each offspring with probability p_ν. Mutation is repeated if the mutated element is infeasible.

Step 4 **Update** S. Replace m least fittest points in S with the m offsprings.

In real coded GA, for example, a crossover can be defined as follows: if $x = (x_1, x_2, \cdots, x_n)$ and $y = (y_1, y_2, \cdots, y_n)$ denote two parents and $\tilde{x} = (\tilde{x_1}, \tilde{x_2}, \cdots, \tilde{x_n})$ and $\tilde{y} = (\tilde{y_1}, \tilde{y_2}, \cdots, \tilde{y_n})$ denote the offsprings, then the offsprings are arithmetically represented by

$$\begin{array}{rcl} \tilde{x_i} & = & \alpha_i x_i + (1-\alpha_i) y_i \\ \tilde{y_i} & = & \alpha_i y_i + (1-\alpha_i) x_i, \end{array} \tag{2}$$

where α_i are uniform random numbers, say in $[-0.5, 1.5]$ (Michalewicz, 1996; Hu et. al., 1997). This crossover rule is referred to as c_{r1} in this paper. c_{r1} does not maintain the convexity property, but it is exploratory. We used this crossover rule in our experiments. As for mutation, the non-uniform mutation (Michalewicz, 1996) can be used. That is, if $x = (x_1, \cdots, x_i, \cdots, x_n)$ is a chromosome and $x_i \in [l_i, u_i]$, where l_i and u_i are the lower and upper bounds of x_i respectively, is the element to be mutated in a generation k, the resulting chromosome will be $x' = (x_1, \cdots, x_i^{'}, \cdots, x_n)$ where $x_i^{'}$ is obtained by

$$x_i^{'} = \begin{cases} x_i + \triangle(k, u_i - x_i) & \text{if} \quad \tau = 0 \\ x_i - \triangle(k, x_i - l_i) & \text{if} \quad \tau = 1, \end{cases} \tag{3}$$

where τ is a random digit that takes either the value zero or one, and

$$\triangle(k, y) = y\left(1 - r^{(1-\frac{k}{T})^b}\right), \tag{4}$$

where r is a random number from $[0, 1]$, T is the maximal generation number, and b is a parameter provided by the user which determines the degree of non-uniformity. This operation returns a value in the range $[0, y]$ such that the probability of returning a number close to zero increases as k increases. This property causes this operator to search uniformly in the initial stages (when k is small) and locally at the final stages. We have used the mutation rule (3) for our experiments. We found the non-uniform mutation much superior to the mutation rule suggested in Hu et. al. (1997). In their mutation, a randomly selected element x_i of x is perturbed using

$$x_i^{'} = x_i + \nu(u_i - l_i), \tag{5}$$

where $\nu = 0.1$. More crossover and mutation rules for real coded GA can be found in Mühlenbein and Schlierkamp-Voosen (1993), Michalewicz (1996), Haslinger et. al. (2000), Herrera and Lozano (2000) and Chelouah and Siarry (2003).

After the creation of the offsprings, the population set S is updated for the next generation. There are different ways of doing this. Some researchers have employed a tournament selection in updating the set S. In this tournament selection, the offsprings and the old population are combined to increase the population size. Then two random points are picked at a time from this extended set, compared and the worst rejected. Thus the new population size is reduced by 1 each time. This process continues until the original population size is reached. Another updating strategy is to replace the least fitted m points in S with the m offsprings, giving the new population. In this strategy, therefore, $N - m$ best points from the old set are automatically copied to the new set. These updating strategies are done in order to enrich the future generations with specific information of the points with the best fitness from the current generation. They also ensure that the best found points so far are not lost. This process of copying the best individuals to the new population is known as elitism. We used the second updating strategy.

In order to facilitate the understanding and to make the difference between the methods more explicit, we append to each individual algorithm name the appropriate parameter containing '(crossover rule, local technique)'. Using this notation, we can write GA that implements the crossover rule c_{r1} and mutation rule (3) as GA($c_{r1}, \cdot$).

3 Proposed modifications to GA

We propose five new versions of the real coded GA. They use either a crossover rule or a combination of crossover rules (or probabilistically combine two rules), and/or a local technique. Our modifications are based on three new features. These are a local technique, an integration of crossover rules and a probabilistic adaptation of crossover rules. We begin by defining the local technique. The local technique (ℓ) is implemented whenever a generation of the GA procedure produces at least one offspring which is better than the best point in S from the previous generation. The local technique is implemented at the end of a

generation that produces such an offspring. Our second feature is based on an integration of two different crossover rules. We refer to this combined crossover rule as c_{r2}. Thirdly, we probabilistically combine the crossover rule c_{r1} with the rule c_{r2}. We refer to this combined crossover rule as c_{r12}. These features, as will be shown later, considerably improve the robustness and efficiency of GA. In all our implementations of GA, the tournament selection is used in selecting the parents to produce offsprings (Ali and Törn, 2004). In particular, we use the tournament selection using two players. That is, two points are chosen at random from the set S and the better of the two is taken as a parent for crossover.

3.1 A genetic algorithm with local technique

In the first version, we introduce the local technique in the GA($c_{r1}, \cdot$) algorithm. One of the recent real coded GA that uses a local technique is reported in Yun et. al. (2003). This local technique explores the neighbourhood of the current best point x_l in S at each iteration. The best point generated in the neighbourhood of x_l by the local technique then competes with x_l. Hence, if this best point is better than x_l then it replaces x_l.

The local technique that we introduce here is different from the local technique suggested by Yun et. al. (2003). In particular, we invoke a local technique if a generation of GA produces a point that is better than the current best point x_l in S. This local technique generates a small number of points, say $\ell_r = 3$. Each of these points attempts to replace the current worst point in S. The technique generates a new trial point $\bar{x}$ using the current best point, x_l, in S and a randomly drawn point y from S, different from x_l. The point $\bar{x}$ then competes with the current worst point in S. We propose the following rule for the local technique:

$$\bar{x}^i = (1 + \gamma^i){x_l}^i - \gamma^i y^i, \tag{6}$$

where $\bar{x}^i$ is the i-th component of $\bar{x}$, and γ^i are uniform random numbers, say in $[-0.5, 1.5]$, different for each i. The point $\bar{x}$ is biased towards the best point x_l. When the points in S are scattered, $\bar{x}$ is not necessarily local to either of the points x_l and y. However, as the points in S gradually form a cluster, $\bar{x}$ becomes local to x_l. Thus, this local search technique is not local in the classical sense. It is introduced for search exploration at early stages and for rapid convergence at later stages. It is therefore clear that the diversity of the population set will not be compromised. Notice that the worst point in S is updated and the best point may also be updated each time the current worst point is replaced by $\bar{x}$. This interesting feature expedites the convergence when the points in S form a cluster around the global minimum. This version of GA is referred to as the genetic algorithm with local technique or GA(c_{r1}, ℓ). We now present the GA(c_{r1}, ℓ) algorithm.

Algorithm 2 : the GA(c_{r1}, ℓ) algorithm

Step 1 **Initialize**. Same as in Algorithm 1.

Step 2 **Stopping rule based on the best and the worst point**. Find the best and worst points in S, x_l and x_h, where the best point x_l has the lowest function value f_l and the worst point x_h has the highest function value f_h. If the stopping condition (e.g., $|f_h - f_l| \leq \epsilon$) is achieved, then stop.

Step 3 **Generate offsprings**

- Tournament selection using two players: Repeat the tournament selection twice to create two parents.
- Crossover and mutation : Create two offsprings using the crossover rule c_{r1} and mutation rule (3).

Repeat Step 3 until m offsprings are generated.

Step 4 **Update** S. Replace the least fittest m points in S with the m offsprings. Set $k = k + 1$. If a point is found which is better than the best point in the previous generation then go to Step 5. Else go to Step 2.

Step 5 **Local technique**. Select a point y at random from the set S, different from the current best x_l. Find a trial point $\bar{x}$ using (6). If $f(\bar{x}) < f_h$ then replace x_h and f_h by $\bar{x}$ and $f(\bar{x})$ in S respectively. Repeat Step 5 ℓ_r times. Go to Step 2.

3.2 A genetic algorithm with integrated crossover rule

We present a genetic algorithm that uses the integrated crossover rule c_{r2}. In this integrated crossover rule two offsprings are produced at a time using three parents x_1, x_2 and x_3. Each of these three parents is selected using tournament of two players. We then mate these parents to create the first offspring y_1 using :

$$y_1^i = \frac{x_1^i + x_2^i}{2} + \sigma_i \left(\frac{x_1^i + x_2^i}{2} - x_3^i \right), \tag{7}$$

where y_1^i is the i-th component of y_1, σ_i is a uniform random number in $[0, 1]$ for each i, and x_3 is the worst parent. This Nelder and Mead (1965) 'simplex-like' crossover rule was used in Kalganova and Strechen (1997), where a single offspring was generated from the three parents. Notice that unlike in Kalganova and Strechen (1997) where σ_i is constant, we use it as a uniform random variable for each component y_1^i of y_1. We also mated the same parents to create the second offspring y_2 using :

$$y_2^i = x_1^i + F_i(x_2^i - x_3^i), \tag{8}$$

where F_i are uniform random numbers in $[0.4, 1]$. The crossover rule (8) was used to generate trial points in the differential evolution (DE) algorithm (Kaelo and Ali, 2005). Thus, in our modification, we combine the above two rules to have a new integrated crossover rule c_{r2} that generates two offsprings at a time. We refer to the algorithm that uses c_{r2} as a real coded genetic algorithm with integrated crossover rule or GA($c_{r2}, \cdot$). We now present the GA($c_{r2}, \cdot$) algorithm.

Algorithm 3 : the GA($c_{r2}, \cdot$) algorithm

Step 1 **Initialize**. Same as in Algorithm 1.

Step 2 **Stopping rule based on the best and the worst point**. Same as in Algorithm 2.

Step 3 **Generate offsprings**

- Tournament selection using two players : repeat the tournament selection three times to create three parents x_1, x_2 and x_3.
- Crossover: Apply crossover rule c_{r2} defined by (7) and (8).
- Mutation: Same as in Algorithm 2.

Repeat Step 3 until m offsprings are generated.

Step 4 **Update** S. Replace the least fittest m points in S with the m offsprings. Set $k = k + 1$ and go to Step 2.

3.3 A genetic algorithm with integrated crossover rule and local technique

Motivated by the results of GA(c_{r1}, ℓ), we incorporated the local technique ℓ in GA($c_{r2}, \cdot$). We refer to this version of GA as a genetic algorithm with integrated crossover rule and local technique or GA(c_{r2}, ℓ). The algorithm for GA(c_{r2}, ℓ) is presented below :

Algorithm 4 : the GA(c_{r2}, ℓ) algorithm

Step 1 **Initialize**. Same as in Algorithm 1.

Step 2 **Stopping rule based on the best and the worst point**. Same as in Algorithm 2.

Step 3 **Generate offsprings**

- Selection: Same as in Algorithm 3.
- Crossover: Same as in Algorithm 3.
- Mutation: Same as in Algorithm 2.

Repeat Step 3 until m offsprings are generated.

Step 4 **Update** S. Replace the least fittest m points in S with the m offsprings. Set $k = k + 1$. If a point is found which is better than the best point in the previous generation then go to Step 5. Else go to Step 2.

Step 5 **Local technique**. Same as in Algorithm 2.

3.4 A genetic algorithm based on probabilistic adaptation of crossover rules

We now propose a genetic algorithm that generates offsprings using some probability distribution over the set $\{c_{r1}, c_{r2}\}$. That is, offsprings are either generated using c_{r1} or c_{r2} at each generation. Initially equal probabilities are assigned to both crossover rules and these

probabilities are updated according to some rules based on reward (for being successful) and penalty (for being unsuccessful). A probabilistic adaptation in the algorithm guides the algorithm in deciding on which crossover rule to use most in generating offsprings. This adaptation procedure allows the algorithm to bias the crossover rule that solves a given problem in an efficient and robust manner. The probabilistic adaptation in the new algorithm follows the concept of a learning system used in learning automata (Bressloff, 1995; Najim et. al., 1990).

In learning automata, a learning system is composed of a stochastic automaton and a performance evaluation unit. The stochastic automaton consists of a finite set

$$U = \{s_1, s_2, \cdots, s_{m_1}\}$$

with m_1 actions. A probability $\alpha_i(k)$ is associated with each action s_i $(i = 1, 2, \cdots, m_1)$ at each iteration k and $\sum_{i=1}^{m_1} \alpha_i(k) = 1$. Initially, all the actions have equal probabilities. At each iteration k, the stochastic automaton selects an action s_i with probability $\alpha_i(k)$ and modify the overall probability distribution based on the response of the performance evaluation unit. The performance evaluation unit responds to the action s_i by giving an evaluation signal $a_k \in \{0, 1\}$, where $a_k = 0$ corresponds to a reward if the selected action s_i meets the intended target, and $a_k = 1$ for penalty otherwise. A reinforcement-learning scheme for the automaton is then used to reward or penalize the action s_i. A number of reinforcement - learning schemes have been proposed, both linear and non-linear (see Bressloff (1995), Najim et. al. (1990) and references therein).

In this paper, we use the non-linear reinforcement-learning scheme that was used in Najim et. al. (1990). Initially, equal probabilities (0.5) are assigned to each of the crossover rules c_{r1} and c_{r2} and these probabilities are then updated, using the reinforcement learning scheme, at each iteration depending on whether the rule used was successful or not. Thus, the probability α_k is assigned to c_{r1} and $\gamma_k = 1 - \alpha_k$ to c_{r2} at iteration k. If the rule c_{r1} proves successful then α_k is increased using the rule

$$\alpha_k = \alpha_{k-1} + \beta_0 \alpha_{k-1}(1 - \alpha_{k-1}), \tag{9}$$

where $0 \leq \beta_0 \leq 1$ is a user provided parameter that controls the increments. On the other hand, if c_{r1} does not prove successful, that is, the offsprings are not better in comparison to the current m least fittest points, then the probability α_k of c_{r1} can either be left unchanged or decreased using the rule:

$$\alpha_k = \alpha_{k-1} - \beta_1 \alpha_{k-1}(1 - \alpha_{k-1}), \tag{10}$$

where $0 \leq \beta_1 \leq 1$ is also a user provided parameter that controls the reductions. We can also reward or penalize the probability γ_k in a similar way. However, rewarding γ_k means penalizing α_k and penalizing γ_k means rewarding α_k. We can therefore work with only one probability, say α_k.

Our motivation for this modification is purely based on numerical experiments with the genetic algorithm using 50 test problems. We observed that GA($c_{r1}, \cdot$), that uses only c_{r1}, performed well in terms of the number of function evaluations for some problems, while for others it performed poorly. The same is true for GA($c_{r2}, \cdot$), which uses c_{r2}. This motivated us

to introduce a scheme that combines these two offsprings generation rules (crossover rules c_{r1} and c_{r2}) probabilistically. Hence we used the above scheme to probabilistically combine the crossover rules c_{r1} and c_{r2}. The purpose of the probabilistic scheme is to design an information-bearing crossover rule that is able to find new promising search regions. The use of this adaptation in an algorithm is to guide the algorithm in deciding on which crossover rule to use most in generating offsprings for any given problem. This allows an algorithm to bias the offsprings generation to a crossover rule that solves a given problem efficiently.

To determine whether to reward or penalize a crossover rule, we propose a scheme that assigns a weight to each offspring created by the crossover rule. The offspring i is assigned a weight p_i, $i = 1, 2, \cdots, m$. We rank order the offsprings in increasing order of function values and assign the highest weight p_1 to the offspring with the smallest function value, and the smallest weight p_m to the offspring with the largest function value. The other offsprings have weights ranging from $p_{(m-1)}$ to p_2 which are obtained using the following scheme:

$$p_i = p_m + \left[\frac{(m-i)}{(m-1)}\right](p_1 - p_m),\ i = 2, \cdots, (m-1). \tag{11}$$

If p_m and p_1 are chosen as $p_m = \frac{c}{m}$ and $p_1 = \frac{2-c}{m}$, $0 \leq c < 1$, respectively, then the above scheme ensures that the higher weights are assigned to offsprings with lower function values. The constant c determines the amount of weight assigned to each offspring. In Yang and Douglas (1998), a similar scheme was used for parent selections for the mating pool. We used this scheme to decide when to reward or penalize the probabilities defined by (9) and (10). We calculated the average function value of the offsprings, and added all the weights of the offsprings that have function values lower than this average. If the total weight of these offsprings is greater than or equal to 0.75 then we reward the crossover rule that created the offsprings, if it falls between 0.25 and 0.75 then we neither reward nor penalize the crossover rule. If, however, the total weight is less than or equal to 0.25, then we penalize the crossover rule. In any of these cases, i.e. whether the probability of a crossover rule is increased, decreased or kept unchanged, the m offsprings always replace the m least fittest points in S.

Using the above mentioned schemes, we came up with an adaptive process which tends to let the algorithm decide which crossover rule to use most in creating offsprings for any given problem. This is done by increasing the value of α_k whenever c_{r1} gives more favourable offsprings, thus reducing the probability of c_{r2}. On the other hand, if c_{r2} gives better offsprings then α_k is reduced. In this way, the algorithm adapts itself to a combination of crossover rules that solves a given problem in a more robust and efficient way. Discussions on the parameters β_0, β_1 and c are given in section 4. We refer to this algorithm as a genetic algorithm based on probabilistic adaptation of crossover rules or GA($c_{r12}, \cdot$). The GA($c_{r12}, \cdot$) algorithm is presented below.

Algorithm 5 : the GA($c_{r12}, \cdot$) algorithm

Step 1 **Initialize**. Same as in Algorithm 1.

Step 2 **Stopping rule based on the best point and the worst point**. Same as in Algorithm 2.

Step 3 **Crossover rule selection**. Go to Step 3a with probability α_k else go to Step 3b with probability γ_k.

Step 3a **Generate offsprings**

- Selection: Same as in Algorithm 2.
- Crossover: Same as in Algorithm 2.
- Mutation: Same as in Algorithm 2.

Repeat Step 3a until m offsprings are generated. Go to Step 4.

Step 3b **Generate offsprings**

- Selection: Same as in Algorithm 3.
- Crossover: Same as in Algorithm 3.
- Mutation: Same as in Algorithm 2.

Repeat Step 3b until m offsprings are generated. Go to Step 5.

Step 4. **Update** α_k **and** S: Set $k = k + 1$.

Step 4a **Reward** α_k. If the offsprings make less progress than the m least fittest points then go to Step 4b. $\alpha_k = \alpha_{k-1} + \beta_0\alpha_{k-1}(1 - \alpha_{k-1})$. Go to Step 4d.

Step 4b **Unchanged**. If these offsprings are worse than the m least fittest points then go to Step 4c. $\alpha_k = \alpha_{k-1}$. Go to Step 4d.

Step 4c **Penalize** α_k. $\alpha_k = \alpha_{k-1} - \beta_1\alpha_{k-1}(1 - \alpha_{k-1})$. Go to Step 4d .

Step 4d **Replace**. Replace the least fittest m points by the m offsprings. Go to Step 2.

Step 5. **Update** α_k **and** S: Set $k = k + 1$.

Step 5a **Penalize** α_k. If the offsprings make less progress than the m least fittest points then go to Step 5b. $\alpha_k = \alpha_{k-1} - \beta_1\alpha_{k-1}(1 - \alpha_{k-1})$. Go to Step 5d.

Step 5b **Unchanged**. If these offsprings are worse than the m least fittest points then go to Step 5c. $\alpha_k = \alpha_{k-1}$. Go to Step 5d.

Step 5c **Reward** α_k. $\alpha_k = \alpha_{k-1} + \beta_0\alpha_{k-1}(1 - \alpha_{k-1})$. Go to Step 5d.

Step 5d **Replace**. Replace the least fittest m points by the m offsprings. Go to Step 2.

3.5 A genetic algorithm based on probabilistic adaptation of crossover rules with local technique

Following the successful incorporation of local technique in GA($c_{r1}, \cdot$) and GA($c_{r2}, \cdot$), we also incorporated the local technique ℓ in GA($c_{r12}, \cdot$). We refer to this new algorithm as genetic algorithm based on probabilistic adaptation of crossover rules with local technique or GA(c_{r12}, ℓ). The algorithm for GA(c_{r12}, ℓ) is presented below.

Algorithm 6 : the GA(c_{r12}, ℓ) algorithm

Step 1 **Initialize**. Same as in Algorithm 1.

Step 2 **Stopping rule based on the best and the worst point**. Same as in Algorithm 2.

Step 3 **Crossover rule selection**. Go to Step 3a with probability α_k else go to Step 3b with probability γ_k.

Step 3a **Generate offsprings**

- Selection: Same as in Algorithm 2.
- Crossover: Same as in Algorithm 2.
- Mutation: Same as in Algorithm 2.

Repeat Step 3a until m offsprings are generated. Go to Step 4.

Step 3b **Generate offsprings**

- Selection: Same as in Algorithm 3.
- Crossover: Same as in Algorithm 3.
- Mutation: Same as in Algorithm 2.

Repeat Step 3b until m offsprings are generated. Go to Step 5.

Step 4. **Update α_k and S**:

Step 4a **Reward** α_k. Same as in Algorithm 5.

Step 4b **Unchanged**. Same as in Algorithm 5.

Step 4c **Penalize** α_k. Same as in Algorithm 5.

Step 4d **Replace**. Same as in Algorithm 5. If a point is found which is better than the best point in the previous generation then go to Step 6. Else go to Step 2.

Step 5. **Update α_k and S**:

Step 5a **Penalize** α_k. Same as in Algorithm 5.

Step 5b **Unchanged**. Same as in Algorithm 5.

Step 5c **Reward** α_k. Same as in Algorithm 5.

Step 5d **Replace**. Same as in Algorithm 5. If a point is found which is better than the best point in the previous generation then go to Step 6. Else go to Step 2.

Step 6. **Local technique**. Same as in Algorithm 2.

Remark:
It can be seen that when $\alpha_k = 0$ then GA($c_{r12}, \cdot$)=GA($c_{r2}, \cdot$) and GA(c_{r12}, ℓ)=GA(c_{r2}, ℓ). Similarly, when $\alpha_k = 1$ then GA($c_{r12}, \cdot$)=GA($c_{r1}, \cdot$) and GA(c_{r12}, ℓ)= GA(c_{r1}, ℓ).

4 Numerical results and discussion

In this section numerical results of all the real coded genetic algorithms described in the previous section are presented. We use 49 test problems from Ali et. al. (2005) and one problem (see the problem ZAK in Table 1) from Chelouah and Siarry (2003), giving 50 problems. All the problems are of continuous variables and a detailed description of the

problems can be found in Ali et. al. (2005) and in Chelouah and Siarry (2003). We compare the new algorithms (Algorithms 2-6) with the simple real coded genetic algorithm using crossover rule c_{r1}, GA($c_{r1}, \cdot$), to assess their robustness in finding the global minimum and their efficiency in terms of the number of function evaluations. We answer the following research questions:

- Does the new local technique improve the real coded GAs, i.e. GA($c_{r1}, \cdot$), GA($c_{r2}, \cdot$) and GA($c_{r12}, \cdot$)?
- Does the new integrated crossover rule benefit the real coded GA, i.e. GA($c_{r1}, \cdot$)?
- How does the new probabilistic combination of crossover rules benefit the real coded GA, i.e. GA($c_{r1}, \cdot$)?

The algorithms were run 100 times on each of the 50 test problems to determine the success rate (SR) (or percentage success) of each algorithm. There were 5000 runs in total. We calculated the average number of function evaluations (FE) and cpu times (CPU) for those runs for which the global minima were found. We used SR, FE and CPU as the criteria for comparison. We note that none of the algorithms succeeded in finding the global minimum for five 10 dimensional functions, namely Rastrigin (RG), Salomon (SAL), Odd Square (OSP), Price's Transistor Modelling (PTM) and Shekel's Foxholes (SF) and the 9 dimensional Storn's Tchebychev function (ST). Except for these six problems, all other problems were solved by at least one of the algorithms. Therefore, results for these six problems are not reflected in our presentation.

4.1 Parameter selection

All the algorithms have some parameter values that are to be provided by the user. We first discuss the parameters that are common to all algorithms. For example, the size N of the population set S, the number of offsprings m, the mutation rate p_ν, the parameter b in the non-uniform mutation in (3), the maximal generation parameter T. We took $N = 100$ and $m = 10$. In the literature, the mutation probability p_ν is a value between 0.001 and 0.1 and a small value is often preferred. We set $p_\nu = 0.001$ for all our experiments. All the values given above are a heuristic choice and the values of N and p_ν can always be increased for obtaining the global minimum with higher probability. However, increasing these values may also increase the computational cost. The parameter T was set to $T = 10000$. The parameter b was set to $b = 5$ as suggested by Michalewicz (1996). Our numerical study also suggested that this is a good value to choose. A run was terminated when either the function values of all points in S were identical to an accuracy of four decimal places, i.e,

$$|f_h - f_l| \leq \epsilon = 10^{-4} \tag{12}$$

or the maximum number of generations (T) was reached. If the functions in the test problem set have a wide spectrum of function values, then the condition (12) may be replaced with another stopping condition, e.g.

$$\left| 1 - \frac{f_l}{f_h} \right| \leq \epsilon, \tag{13}$$

to ensure the same level of accuracy. However, our test problems set consists of 41 functions with $-5 \leq f_{opt} \leq 5$, 4 functions with optimum value close to -10.5. The remaining 5 functions have their global minima far from zero. For example, the global minimum of Dekkers and Aarts (DA), Neumaier 3 (NF3), Paviani (PP), Schubert (SBT) and Schwefel (SWF) functions are

$$-24776.518, -210, -44.778, -186.73 \text{ and } 4189.829$$

respectively. Our numerical experiment suggested (see Table 1) that the stopping condition (12) is sufficient to achieve the same level of accuracies. Hence, a success was counted when the following condition was met

$$f_l - f_{opt} \leq 0.01, \tag{14}$$

where f_{opt} is the known global minimum of the problem and f_l is the best value obtained by an algorithm.

As a result of probabilistic combination of crossover rules, three parameters were introduced in GA($c_{r12}, \cdot$) and GA(c_{r12}, ℓ). These are β_0 in (9), β_1 in (10) and c in (11). Different combinations of β_0 and β_1 values can be used. We carried out numerical experiments using various combinations of values for these parameters. Our numerical experiments with β_0 and β_1 suggested that these parameters are problem dependent. However, for almost all problems in the test set, we observed that combinations of values that had β_0 smaller than β_1 gave good results. This means rewarding a scheme less for a success and penalizing it more for failure. We also observed that smaller values for these parameters gave better results than those obtained using higher values. Hence, for our experiments, we used combinations of smaller values for both β_0 and β_1 (Kaelo, 2005). Although the results for other combinations used were also good, the results presented here are the best results obtained for $\beta_0 = 0.05$ and $\beta_1 = 0.065$. For results using other combinations of β_0 and β_1, see Kaelo (2005). We forced $\alpha_k, k \geq 1$, to lie in $[0.05, 0.95]$ in all our experiments. For example, if α_k goes below 0.05, we set $\alpha_k = 0.05$ by clipping. This was done in order to avoid the algorithm switching entirely to one procedure. Thus α_k and γ_k lie in $(0, 1)$ to allow the algorithm to be able to switch from one crossover rule to the other that may be suited to a problem at hand. The value of c was taken to be 0.1. We chose a small value of c so that more weight is assigned to the best offsprings. From (11), it is clear that as the value of c increases from 0 to 1, the weight assigned to the worst offsprings increases and as c approaches 1, all the offsprings will have equal weighting. Thus, a higher value of c would lead to a case where there is no reward or penalty for α_k and the algorithm would then generate offsprings using a constant α_k.

4.2 Comparisons and discussions

The results of GA($c_{r1}, \cdot$) along with the results of the new algorithms are presented in Table 1, where TR in the last row represents the total FE and SR. Each problem in this table was solved by at least one algorithm. The results in Table 1 show that both GA($c_{r12}, \cdot$) and GA(c_{r12}, ℓ) solved 43 problems out of 50 while the other algorithms solved either 41 or 42 problems.

Table 1: Comparison of GA($c_{r1}, \cdot$), GA(c_{r1}, ℓ), GA($c_{r2}, \cdot$), GA(c_{r2}, ℓ), GA($c_{r12}, \cdot$) and GA(c_{r12}, ℓ)

		GA($c_{r1}, \cdot$)		GA(c_{r1}, ℓ)		GA($c_{r2}, \cdot$)		GA(c_{r2}, ℓ)		GA($c_{r12}, \cdot$)		GA(c_{r12}, ℓ)	
P	n	FE	SR	FE	SR	FE	SR	FE	SR	FE	SR	FE	SR
ACK	10	5609	95	5265	99	7008	98	6784	98	6532	98	6291	100
EM	10	0	0	0	0	0	0	0	0	7863	4	0	0
GRP	3	1443	4	2581	15	2157	100	2054	100	2309	88	2136	96
H6	6	1870	85	1709	83	2017	94	1876	93	1909	96	1806	91
HV	3	5270	15	10001	99	2725	100	2615	100	3017	93	2638	100
KL	4	1033	38	928	47	789	91	757	86	839	87	764	84
MR	3	1548	17	1575	23	1767	77	1669	78	1860	72	1637	77
ML	10	8548	4	7105	2	7760	1	6894	2	7674	7	8726	8
MRP	2	1530	50	1465	71	1643	63	1414	66	1481	72	1399	72
MGP	2	2329	78	2333	71	1594	67	1593	52	1695	64	1542	68
NF3	10	0	0	38036	100	9562	100	6931	100	9874	100	7123	100
PP	10	3879	99	3234	100	4153	100	3638	100	3928	100	3600	100
PRD	2	2540	64	3200	66	2573	100	2345	100	2225	93	2107	94
PQ	4	4150	67	3674	100	2665	100	2488	100	2785	100	2603	100
RB	10	0	0	0	0	0	0	30667	93	0	0	31941	92
SF1	2	8019	8	8460	7	5130	60	4664	65	4446	59	4668	60
SBT	2	2094	98	2225	100	3367	100	3326	100	2565	100	2521	100
SWF	10	6458	6	5907	6	0	0	0	0	8217	3	6686	4
S5	4	3559	47	2866	44	2441	81	2268	72	2497	71	2191	74
S7	4	2720	70	2481	71	2228	92	2158	92	2203	90	2117	86
S10	4	2404	79	2351	73	2347	91	2213	93	2215	93	2094	87
SIN	20	5415	66	5469	100	8779	100	7942	100	8058	100	7226	100
WP	4	9381	7	12299	7	7621	100	6124	100	7875	95	5974	98
ZAK	10	13477	41	10442	100	6201	100	6137	100	6767	100	6239	100
AP	2	992	100	929	100	932	100	897	100	923	100	894	100
BL	2	1119	100	1061	100	1229	100	1176	100	1144	100	1124	100
B1	2	1341	100	1252	100	1335	100	1275	100	1302	100	1252	100
B2	2	1389	100	1292	100	1320	100	1266	100	1303	100	1254	100
BR	2	1181	100	1046	100	1130	100	1124	100	1104	100	1041	100
CB3	2	931	100	881	100	878	100	860	100	884	100	857	100
CB6	2	1029	100	979	100	1137	100	1094	100	1089	100	1019	100
CM	4	1420	100	1305	100	1600	100	1563	100	1489	100	1403	100
DA	2	1671	100	1547	100	1577	100	1494	100	1559	100	1507	100
EP	2	1085	100	1040	100	1003	100	981	100	1013	100	982	100
EXP	10	2127	100	2051	100	2552	100	2530	100	2415	100	2342	100
GP	2	1279	100	1163	100	1094	100	1059	100	1117	100	1076	100
GW	10	3371	100	3111	100	4064	100	3832	100	3745	100	3659	100
H3	3	1117	100	973	100	985	100	954	100	1015	100	966	100
HSK	2	804	100	765	100	753	100	730	100	762	100	728	100
LM1	3	1222	100	1143	100	1314	100	1243	100	1238	100	1177	100
LM2	10	2713	100	2617	100	3554	100	3401	100	3224	100	3165	100
MC	2	860	100	812	100	768	100	750	100	794	100	768	100
MCP	4	1205	100	1072	100	715	100	671	100	832	100	781	100
SF2	2	2810	100	2592	100	2607	100	2511	100	2623	100	2501	100
TR		122942	3038	161237	3384	115074	3815	135968	3890	128409	3785	142525	3891

To answer our first research question, we compare the results of the algorithms that implement the local technique with those that do not. From the total results in the last row of Table 1, it can be seen that GA(c_{r1}, ℓ) is superior to GA($c_{r1}, \cdot$) by 346 successes. It is inferior to GA($c_{r1}, \cdot$) by 32% in terms of FE. However, the function NF3 was solved by GA(c_{r1}, ℓ) while GA($c_{r1}, \cdot$) was unsuccessful. Hence, if we compare these two algorithms by excluding the results for this function then GA(c_{r1}, ℓ) is superior to its counterpart by 246 in terms of SR, while FE incurred by both algorithms remain almost the same. The same trend can be observed for GA($c_{r2}, \cdot$) and GA(c_{r2}, ℓ) and also for GA($c_{r12}, \cdot$) and GA(c_{r12}, ℓ). We summarise these results in Table 2.

To address our second and third research questions, we compare the algorithms on the basis of various crossover rules. In particular, we make a comparison of the algorithms that do not implement local technique. By looking at the TR in Table 1, it can be seen that GA($c_{r2}, \cdot$) is superior to the other two algorithms both in terms of SR and FE. It can also be seen that GA($c_{r12}, \cdot$) is much superior to GA($c_{r1}, \cdot$) in terms of SR. However, GA($c_{r12}, \cdot$) solved two more difficult problems (EM and SWF) than GA($c_{r2}, \cdot$). If we compare FE of these two algorithms excluding the results for EM and SWF, we see that the former is superior to the latter.

Next we compare all the algorithms using the total results of those functions for which all algorithms succeeded in finding the global minima. These results are presented in Table 2 where we have also presented the total CPU. Using the total results from Table 2, we rank order the algorithms in Table 3. From this table it can be seen that there is no overall best performer in terms of FE, SR and CPU. In terms of FE, GA(c_{r12}, ℓ) is the best algorithm. In terms of SR, GA($c_{r2}, \cdot$) is the best while in terms of CPU, GA($c_{r12}, \cdot$) is the best performer. By looking at Table 2, we see that the algorithms that have the integrated crossover rule c_{r2} are comparable and are superior to the algorithms that have c_{r1} only. This shows that the integrated crossover rule c_{r2} and local technique both greatly improve the real coded GA.

Table 2: Comparison of GA($c_{r1}, \cdot$),GA(c_{r1}, ℓ),GA($c_{r2}, \cdot$),GA(c_{r2}, ℓ), GA($c_{r12}, \cdot$) and GA(c_{r12}, ℓ)

	GA($c_{r1}, \cdot$)	GA(c_{r1}, ℓ)	GA($c_{r2}, \cdot$)	GA(c_{r2}, ℓ)	GA($c_{r12}, \cdot$)	GA(c_{r12}, ℓ)
FE	112018	112718	97501	91179	95221	90301
SR	3066	3278	3715	3697	3678	3695
CPU	3.00	3.66	2.74	2.89	2.70	2.76

Table 3: Rank order of algorithms

Rank	1	2	3	4	5	6
FE	GA(c_{r12}, ℓ)	GA(c_{r2}, ℓ)	GA($c_{r12}, \cdot$)	GA($c_{r2}, \cdot$)	GA($c_{r1}, \cdot$)	GA(c_{r1}, ℓ)
SR	GA($c_{r2}, \cdot$)	GA(c_{r2}, ℓ)	GA(c_{r12}, ℓ)	GA($c_{r12}, \cdot$)	GA(c_{r1}, ℓ)	GA($c_{r1}, \cdot$)
CPU	GA($c_{r12}, \cdot$)	GA($c_{r2}, \cdot$)	GA(c_{r12}, ℓ)	GA(c_{r2}, ℓ)	GA($c_{r1}, \cdot$)	GA(c_{r1}, ℓ)

We now compare all the algorithms graphically using SR from Table 1. We exclude all those problems for which all six algorithms achieved 100% success, as these problems will

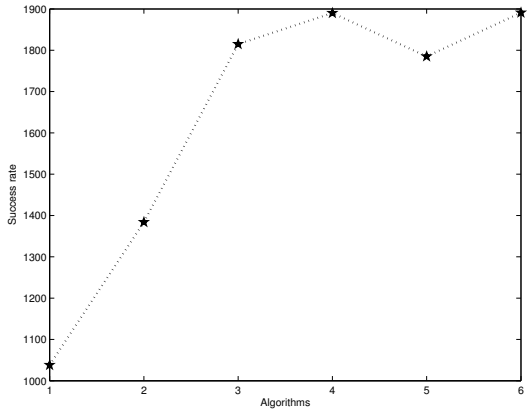


Figure 1: Total SR for the algorithms on 24 functions

not add any value to our graphic presentation. We, therefore, compare the algorithms on 24 problems. These problems are given in the first 24 rows of Table 1. We plot the graph using the algorithms as the x - axis and the total SR as the y - axis. The six algorithms in Table 1, from left to right, are numbered as 1 to 6 respectively. This is shown in figure 1. Figure 1 clearly shows that GA(c_{r2}, ℓ) and GA(c_{r12}, ℓ) are the best performers. The runners-up are GA($c_{r2}, \cdot$) and GA($c_{r12}, \cdot$). It is, therefore, clear that the integrated crossover rules c_{r2} and c_{r12} are much superior to the original crossover rule c_{r1}. Moreover, the incorporation of the local technique in GA($c_{r2}, \cdot$) and GA($c_{r12}, \cdot$) have further improved their performances. A similar comparison of the algorithms can also be presented using FE.

Table 4: List of other algorithms used in the comparison

Algorithm	References
Continuous genetic algorithm (CGA)	Chelouah and Siarry (2000)
Continuous hybrid algorithm (CHA)	Chelouah and Siarry (2003)
Continuous tabu simplex search (CTSS)	Chelouah and Siarry (2005)
Simplified atavistic DE (SADE)	Hrstka and Kučerová (2004)

The performance of the overall best algorithms, GA(c_{r2}, ℓ) and GA(c_{r12}, ℓ), in terms of success rate, is further compared with other competing algorithms in the literature. These algorithms are presented in Table 4 together with the references in which they appear. We compare the success rate of the algorithms on a common set of test functions. This comparison is made in Table 5. Results missing in Table 5 suggest that they were not available in the corresponding references. We see from Table 5 that the SR of the new GAs are very competitive with other continuous meta-heuristics proposed in the literature. GA(c_{r2}, ℓ) and GA(c_{r12}, ℓ) outperform CHA, CGA and CTSS in the shekel family and the 10 dimensional RB problem. However, GA(c_{r2}, ℓ) and GA(c_{r12}, ℓ) are outperformed by CGA and CHA in the Hartman 6 problem. A comparison of GA(c_{r2}, ℓ) and GA(c_{r12}, ℓ) with SADE shows that the new GAs are outperformed by SADE in the shekel family. They, however, achieved better success for the Hartman 6 (H6) problem.

Table 5: Comparison of 6 algorithms on some common functions

function	GA(c_{r2}, ℓ)	GA(c_{r12}, ℓ)	CGA	CHA	CTSS	SADE
BR	100	100	100	100	100	100
B2	100	100	100	100	100	-
EP	100	100	100	100	100	-
GP	100	100	100	100	100	100
SBT	100	100	100	100	100	100
H3	100	100	100	100	100	100
H6	93	91	100	100	-	67
S5	72	74	76	85	69	99
S7	92	86	83	85	68	100
S10	93	87	81	85	65	99
RB	93	92	80	83	-	-
ZAK	100	100	100	100	-	-

We also present results of these algorithms obtained using tournament of 3 players. These results are presented in Table 6. Like in Table 2, these results are for only those problems that were solved by all the algorithms. As can be seen from Table 6, the tournament of 3 players is a bit greedy when compared to the tournament of 2 players in that though it reduces the number of function evaluations greatly and also improves CPU, it has a negative effect in the success rate. Also, from Tables 2 and 6, it can be seen clearly that the local technique has an effect of reducing FE. Tables 2 - 6 show that the deterministic integration of crossover rules, i.e. c_{r2}, has an effect of increasing SR and the probabilistic integration of crossover rules, i.e. c_{r12}, has an effect of reducing FE. These crossover rules, when equipped with local technique, have further effects of improving FE. We also carried out numerical studies with GA using the offspring generation rules (7) and (8) separately and combining them probabilistically. The results of these studies are inferior to the results presented here. From the analysis of the results using tables 1 - 6 and figure 1, we see that the new algorithms greatly improved the reliability, in terms of locating the global minimum, and also improved the efficiency, in terms of FE and CPU, of the real coded genetic algorithm GA($c_{r1}, \cdot$). Therefore, the introduction of the new features, i.e. local technique, integrated crossover rule and probabilistic combination of crossover rules, is fully justified.

Table 6: Comparison of GA($c_{r1}, \cdot$),GA(c_{r1}, ℓ),GA($c_{r2}, \cdot$),GA(c_{r2}, ℓ),GA($c_{r12}, \cdot$) and GA(c_{r12}, ℓ) using tournament of 3 players

	GA($c_{r1}, \cdot$)	GA(c_{r1}, ℓ)	GA($c_{r2}, \cdot$)	GA(c_{r2}, ℓ)	GA($c_{r12}, \cdot$)	GA(c_{r12}, ℓ)
FE	100804	80515	71467	63982	68906	63076
SR	2863	3195	3564	3601	3537	3541
CPU	2.30	2.75	2.02	2.02	1.99	2.30

Now we show how the probabilistic adaptation takes place within an algorithm. We use the algorithm GA($c_{r12}, \cdot$) to demonstrate the probabilistic adaptation. We present three figures to illustrate these adaptations. The figures have been plotted using the number of

iterations (generations) k as the horizontal axis and values of α_k as the vertical axis. For a particular problem, we observed slight variations in plots from run to run. However, the general trend is the same for all successful runs. Therefore, we present each figure from a single run. We use figures 2 - 4 to illustrate the probabilistic adaptation of c_{r1} and c_{r2} within GA($c_{r12}, \cdot$). The figures 2 - 4 are respectively for the functions Becker and Lago (BL), Helical Valley (HV) and Neumaier 3 (NF3). The GA($c_{r12}, \cdot$) algorithm uses crossover rule c_{r1} for $\alpha_k = 1$, crossover rule c_{r2} for $\alpha_k = 0$ and a mixture of the two for any $\alpha_k \in (0, 1)$. For the Becker and Lago problem (Figure 2), GA($c_{r12}, \cdot$) mostly uses crossover rule c_{r1} for the first 20 generations, then a mixture of (slightly less) crossover rule c_{r2} and (slightly more) crossover rule c_{r1} up to 90 generations, and finally switching mostly to crossover rule c_{r2}. For the Helical Valley problem (Figure 3), the algorithm uses on average an equal probability for the two crossover rules for up to about 150 generations before gradually switching mostly to crossover rule c_{r2}. However, for the Neumaier 3 problem (Figure 4), it consistently uses crossover rule c_{r2} to solve this problem. As can be seen from Table 1, this problem was not solved by GA($c_{r1}, \cdot$) but by GA($c_{r2}, \cdot$). This clearly shows the dominance of crossover rule c_{r2} in GA($c_{r12}, \cdot$) in solving this problem.

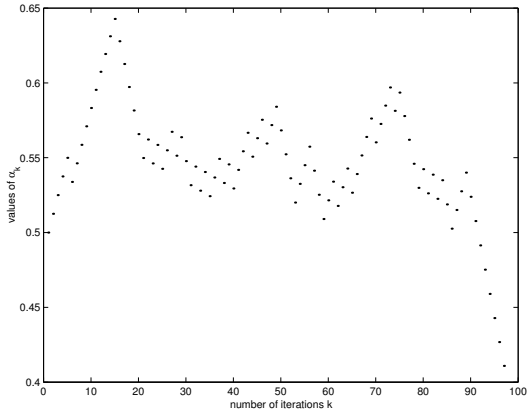


Figure 2: Probabilistic adaptation of GA($c_{r12}, \cdot$) on BL

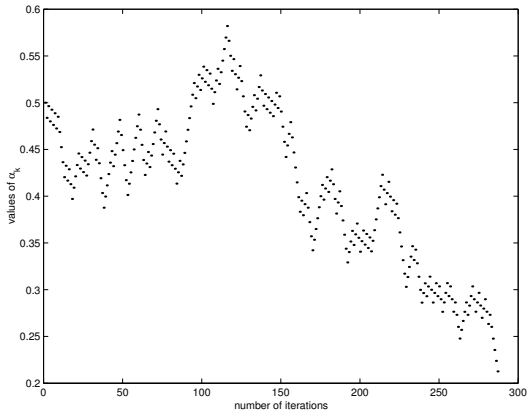


Figure 3: Probabilistic adaptation of GA($c_{r12}, \cdot$) on HV

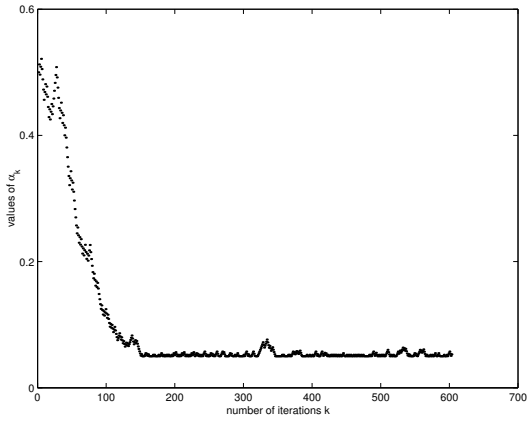


Figure 4: Probabilistic adaptation of GA($c_{r12}, \cdot$) on NF3

5 Conclusion

We have introduced and tested five new versions of the real coded genetic algorithm on 50 benchmark test problems with dimensions ranging from 2 to 20. From the comparisons in the previous section, it is quite clear that the new algorithms are much better than the original real coded GA.

The local technique improves the robustness and efficiency of GA in terms of finding the global minima and reducing the number of function evaluations respectively. The proposed integrated crossover rules showed their robustness in terms of solving more problems and a higher success rate compared to the original GA. In particular, the probabilistic crossover rule was able to solve more problems than the others. It also proved efficient in terms of the number of function evaluations and cpu times. Some graphic examples were also presented to show how the probabilistic adaptation guides an algorithm. Further research is underway in developing even more efficient GA for high dimensional problems.

References

Ali, M. M., Khompatraporn, C., Zabinsky, Z. B., 2005. A numerical evaluation of several stochastic algorithms on selected continuous global optimization test problems, Journal of Global Optimization 31 635–672.

Ali, M. M., Törn, A., Viitanen, S., 1997. A numerical comparison of some modified controlled random search algorithms, Journal of Global Optimization 11 377–385.

Ali, M. M., Törn, A., 2004. Population set based global optimization algorithms : some modifications and numerical studies, Computers and Operations Research 31(10) 1703–1725.

Bressloff, P. C., 1995. Stochastic dynamics of reinforcement learning, Network: Computation in Neural Systems 6 289-307.

Chelouah, R., Siarry, P., 2000. A continuous genetic algorithm designed for the global optimization of multimodal functions, Journal of Heuristics 6 191-213.

Chelouah, R., Siarry, P., 2005. A hybrid method combining continuous tabu search and Nelder-Mead simplex algorithms for the global optimization of multiminima functions, European Journal of Operational Research 161 636–654.

Chelouah, R., Siarry, P., 2003. Genetic and Nelder-Mead algorithms hybridized for a more accurate global optimization of continuous multiminima functions, European Journal of Operational Research 148 (2) 335–348.

Haslinger, J., Jedelský, D., Kozubek, T., Tvrdík, J., 2000. Genetic and random search methods in optimal shape design problems, Journal of Global Optimization 16 109–131.

Herrera, F., Lozano, M., 2000. Two-loop real-coded genetic algorithms with adaptive control of mutation step sizes, Applied Intelligence 13 187–204.

Herrera, F., Lozano, M., Sanchez, A. M., 2003. A taxanomy for the crossover operator for real-coded genetic algorithms: an experimental study, International Journal of Intelligent Systems 18(3) 309-338.

Hrstka, O., Kučerová, A., 2004. Improvements of real coded genetic algorithms based on differential operators preventing premature convergence, Advances in Engineering Software 35(3-4) 237-246.

Hu, Y. F., Maguire, K. C., Cokljat, D., Blake, R. J., 1997. Parallel controlled random search algorithms for shape optimization. In: Emerson D. R., Ecer A., Periaux J., Satofuka N., editors, Parallel Computational Fluid Dynamics, Amsterdam, Holland, pp. 345–352.

Kaelo, P., Ali, M. M., 2005. A numerical study of some modified differential evolution algorithms, European Journal of Operational Research, in press.

Kaelo, P., 2005. Some population set based methods for unconstrained global optimization, *PhD thesis*, submitted.

Kalganova, T., Strechen, N., 1997. Genetic algorithm approach to find the best input variable partitioning, Proceedings of the 3rd Nordic Workshop on GA, Helsinki, pp. 101–290.

Maaranen, H., Miettinen, K., Mäkelä, M. M., 2004. Quasi-random initial population for genetic algorithms, Computers and Mathematics with Applications 47 1885-1895.

Michalewicz, Z., 1996. *Genetic Algorithms + Data Structures = Evolution Programs*, Springer-Verlag, Berlin.

Mühlenbein, H., Schlierkamp-Voosen, D., 1993. Predictive models for the breeder genetic algorithms I. continuous parameter optimization, Evolutionary Computation 1 25–49.

Najim, K., Pibouleau, L., Le Lann, M. V., 1990. Optimization technique based on learning automata, Journal of Optimization Theory and Applications 64 331–347.

Nelder, J. A., Mead, R., 1965. A simplex method for function minimization, Computer Journal 7 308–313.

Price, W. L., 1983. Global optimization by controlled random search, Journal of Optimization Theory and Applications 40 333–348.

Storn, R., Price, K., 1997. Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces, Journal of Global Optimization 11 341–359.

Tsutsui, S., Goldberg, D. E., 2001. Search space boundary extension method in real-coded genetic algorithms, Information Sciences 133 229–247.

Yang, R., Douglas, I., 1998. Simple genetic algorithm with local tuning: efficient global optimization technique, Journal of Optimization Theory and Applications 2 449–465.

Yun, Y., Gen M., Seo, S., 2003. Various hybrid methods based on genetic algorithm with fuzzy logic controller, Journal of Intelligent Manufacturing 14 401–419.