Continuous Non-Revisiting Genetic Algorithm

Shiu Yin Yuen and Chi Kin Chow

Abstract—**The non-revisiting genetic algorithm (NrGA) is extended to handle continuous search space. The extended NrGA model, Continuous NrGA (cNrGA), employs the same tree-structure archive of NrGA to memorize the evaluated solutions, in which the search space is divided into non-overlapped partitions according to the distribution of the solutions. cNrGA is a bi-modulus evolutionary algorithm consisting of the genetic algorithm module (GAM) and the adaptive mutation module (AMM). When GAM generates an offspring, the offspring is sent to AMM and is mutated according to the density of the solutions stored in the memory archive. For a point in the search space with high solution-density, it infers a high probability that the point is close to the optimum and hence a near search is suggested. Alternatively, a far search is recommended for a point with low solution-density. Benefitting from the space partitioning scheme, a fast solution-density approximation is obtained. Also, the adaptive mutation scheme naturally avoid the generation of out-of-bound solutions. The performance of cNrGA is tested on 14 benchmark functions on dimensions ranging from 2 to 40. It is compared with real coded GA, differential evolution, covariance matrix adaptation evolution strategy and two improved particle swarm optimization. The simulation results show that cNrGA outperforms the other algorithms for multi-modal function optimization.**

I. Introduction

SEARCH history, including the performed operations, the positions of the evaluated solutions and the fitness values of the solutions, are valuable information to enhance the performance of an evolutionary algorithm (EA). Intuitively, it can be used to maintain diversity. It can also guide the search direction or suggest promising search regions of interest. In addition, when the same optimum reappears in the search history, it can warn that the search may have trapped in a local optimum. In expensive objective function optimization, one may use the history to approximate the objective function and pre-evaluate the potential optimum on this approximated function. This helps save computation cost.

Several search algorithms [1-3] employ search history in the form of memory to adaptively guide the search strategies. However, they only use partial search histories – that is, only part of the information gained from the search is retained and the rest are discarded. Recently, a non-revisiting genetic algorithm (NrGA) is proposed by Yuen and Chow [4, 5]. It memorizes all evaluated solutions. It is the first algorithm to our knowledge that advocates a non-revisiting design for EA. A binary partitioning tree (BSP) archive is used to store the evaluated solutions. Meanwhile, it divides the search space into non-overlapped rectangular partition set $H = \cup_i h_i$ of different sizes according to the cumulative distribution of the evaluated solutions.

The non-revisiting scheme of NrGA uses this archive to prevent solution re-evaluation. As a result, a solution that has been visited before will never be revisited. In addition, the scheme acts as a parameter-less adaptive mutation operator: the input $\mathbf{x}$ of this operator can be any point in the search space. If $\mathbf{x}$ is a *revisit*, this operator outputs solution $\mathbf{r}$ such that 1) $\mathbf{r} \neq \mathbf{x}$, 2) $\mathbf{x}, \mathbf{r} \in h \subseteq H$ and 3) $\mathbf{r}$ is randomly selected from h. If $\mathbf{x}$ is not a revisit, $\mathbf{r}$ is assigned as $\mathbf{x}$. Since the size of $h \subseteq H$ gradually decreases as the number of iterations increase, and $\mathbf{r}$ is selected from h where $\mathbf{x} \in h$, the expected distance between $\mathbf{x}$ and $\mathbf{r}$ (the mutation step size of $\mathbf{x}$) becomes smaller. The qualitative picture is as follows: Initially, the GA uses the mechanism of selection and crossover to explore the search space. This is a pure exploration phase. As selection and crossover finds promising hyper-rectangular boxes h, revisits begin to occur. An exploration-exploitation phase begins. h is searched using mutation steps that depends on the size of h, which in turn depends on the number of revisits. The more promising is h, the smaller is the mutation steps. Hence it implements an adaptive mutation. The exploration (selection and crossover) and exploitation (adaptive mutation) work cooperatively in a parameter-less manner. This is a worthy merit as parameter settings and control is a very hard problem in evolutionary computation [6, 7]. NrGA has been compared with 1) GA; 2) real-coded GA; 3) GA with simple diversity; 4) particle swarm optimization (PSO); 5) two improved versions of PSOs; and 6) Covariance Matrix Adaptation Evolution Strategy (CMA-ES). The comparison is done on 19 famous benchmark functions, which includes uni-modal functions, multi-modal functions, rotated multi-modal functions and hybrid composition function. The problem dimension D varies from 10 to 40 (except for four 2-dimensional functions). NrGA obtains superior performance compared with all the above methods [5].

The non-revisiting idea is a generic idea that can be applied to other evolutionary algorithms. For example, recently, it has been applied to simulated annealing [8] and particle swarm optimization [9]. In each case, significant performance gains are observed.

Theoretical investigations have also been done on simplified versions of NrGA: the (1+1) EA with memory and

Shiu Yin Yuen is with the Department of Electronic Engineering, City University of Hong Kong, Hong Kong SAR, China (E-mail: kelviny.ee@cityu.edu.hk).

Chi Kin Chow is with the Department of Electronic Engineering, City Universtiy of Hong Kong, Hong Kong SAR, China (E-mail: chowchi@cityu.edu.hk).

The work described in this paper was supported by CityU Research Grant (7002304).

978-1-4244-2959-2/09/$25.00 ©

randomized local search (RLS) with memory. It is shown that by reducing revisits, it is sometimes possible to drastically reduce the expected time complexity of finding the optimal solution from exponential to polynomial [10, 11].

The idea of NrGA advocates storing all visited solutions in memory, then organize it in some way to adaptively guide the search. At first glance, this may involve a tremendous amount of memory. In fact, it is untrue. Firstly, consider applications that involve expensive and time consuming fitness evaluations; the fitness evaluation cost is significantly higher than solution generation cost (the black box optimization scenario in computational complexity theory [12]). There are many such applications in engineering, artificial intelligence and robotics. For such problems, the total number of evaluations that can be made by an EA cannot be too large; and it is culpable to throw away any information gained from fitness evaluations. Secondly, currently, the memory that is becoming available due to advance in computer technology is increasing drastically (Moore's Law). Algorithms that are previously considered memory demanding, e.g. z buffer in computer graphics, are now standard provisions. Thirdly, even in applications whose solution generation costs is significant, it has recently been shown that using the entire search history may still give performance gains in spite of the memory overhead (see [13] for our study on an NP hard problem). Finally, EA are seldom used alone. Memetic algorithms, the idea of hybridizing EA with another algorithm, often results in the best algorithm. In such a scenario, the total number of evaluations that the EA is subject to is limited.

In keeping with traditional GA, NrGA has a finite resolution parameter *d* for each gene in the chromosome. This allows it to be applied to both discrete (combinatorial) (e.g. the combinatorial optimization problem [13]) and continuous (real) parameter problems. In [5], we show that the performance of NrGA is relatively insensitive to the settings of the resolution parameter *d*, which is desirable. However, there are many practical applications in which parameters take on continuous values. Hence there is a practical need to consider a new type of NrGA that work on real valued parameters.

In this paper, we extend the NrGA model to perform real valued optimization. The modified NrGA, named *Continuous Non-Revisiting Genetic Algorithm* (cNrGA), adopts the parameter-less adaptive mutation scheme of NrGA to handle continuous search space. It uses the distribution of the evaluated solutions in the search space to perform a real valued adaptive mutation.

cNrGA consists of two modules: the genetic algorithm module (GAM) and the adaptive mutation module (AMM). GAM executes the general process in a genetic algorithm with crossover and selection. AMM manages the search history and when revisits occur, redirects/mutates offspring generated by GAM before the fitness evaluation. The search history is stored by a memory archive called *Density Tree*, which is a structure modified from the BSP tree in NrGA.

Though cNrGA and NrGA share the same nature in that they both memorize all evaluated solutions to perform adaptive mutation, cNrGA and NrGA are quite different algorithms. As aforementioned, NrGA is applied on discrete search space where the precision of its optima depends on the axis resolution. On the other hand, cNrGA deals with continuous search space to which the precision of its optima is up to the precision of the floating point number in the computer.

Moreover, the mechanism of cNrGA is also different from that of NrGA. Firstly, since the search space of cNrGA is continuous, the number of possible solutions in a space partition is infinite. Thus, the Density Tree node has no *Closed* flag for indicating a fully evaluated sub-region (called "subspace" in [4, 5]. The term sub-region is used instead in this paper to avoid possible confusion with the mathematical term subspace in linear algebra); and node-pruning operation in NrGA [4, 5] is not necessary for Density Tree. This makes the cNrGA algorithm somewhat simpler and tidier compared with the NrGA algorithm.

The "standard" NrGA is a (μ+λ) GA with elitist selection, uniform crossover at the maximum rate of 0.5 and resolution parameter *d*, which is application dependent. The adaptive mutation is parameter-less. A background mutation (e.g. at fixed mutation rate of *1/D*) may (as in [4]) or may not (as in [5]) be included. There are only a set of three parameters (μ, λ, d) that needs to be set. cNrGA exempts the setting of *d*, which is nice aesthetically. In a companion paper, we study in more details NrGA's sensitivities to operator and parameter choices [14].

The rest of this paper is organized as follows: Section II reports the structure and the distinct features of Density Tree. Section III presents the mechanism of cNrGA. Section IV reports the experimental results. Section V gives the conclusion.

II. Density Tree

As in NrGA, cNrGA uses a BSP tree as an archive which stores the positions of the evaluated solutions $\{\mathbf{s_i}\}$. It partitions the whole search space *S* according to the distribution of $\{\mathbf{s_i}\}$. A tree node represents a partitioned sub-region of *S*. Suppose a parent node has two child nodes **l** and **r**. The sub-regions represented by **l** and **r** are disjoint and their union is the sub-region of the parent, (i.e., the child nodes binary partitions the parent sub-region). As the tree construction depends on the sequence of solutions found by the GA, the BSP tree is a random tree and its topology is different from trial to trial.

It should be pointed out that the memory archive of cNrGA and NrGA have two differences in terms of the represented information and the tree operation:

1. A sub-region at a leaf node will be called a partition in this paper, refer to *Definition 1* below. Since the search space of cNrGA is continuous, the number of possible solutions in a partition, either for a small partition or a large

partition, is infinite. No partition can be fully evaluated. Thus, no node-pruning is performed.

2. Hence the number of nodes of the tree is exactly the same as the number of evaluated solutions. Thus a node of the tree represents a sub-region that encodes the density of the visits.

Because of property 2, we call the tree a *density tree*.

Thus the tree used in cNrGA is the same as that used in NrGA except all node status management, pruning and backtracking operation are removed. Initially, the density tree *Ar* consists of only the root node. Each node of the tree records a distinct previously visited solution **x** of the GA. It also represents an unvisited "sub-region" **X** of the search space. Each sub-region is a hyper-rectangular box of the search space. The tree is organized such that the nodes binary partition the search space using metric *d(.)*, which in this paper is the Euclidean metric. Two child nodes **x** and **y** binary partition the space of its parent node into two disjoint halves **X** and **Y,** $\mathbf{X} \cap \mathbf{Y} = \phi$, by a hyper-plane at dimension *j*, chosen such that the difference of **x** and **y** along the dimension is the largest amongst all dimensions. In this way, each previous solution generated by the GA is recorded in a node of the tree, and the BSP tree serves as an efficient data structure to query whether a new solution **z** is a revisit. If it is not a revisit (revisit flag RF = 0), then a new node is generated to represent **z**, and implicitly its unvisited sub-region **Z**. If it is a revisit, then the revisit flag RF is set to 1 and the search goes down to the leaf node of the tree, finds its unvisited sub-region, mutates randomly to a new solution in this sub-region, and records this new solution in the tree by creating a new node under the leaf node. By definition, this new solution must be unvisited. The fitness of this new solution is then evaluated.

The pseudo code is as follows:

```
1. Initial condition: Ar consists of the root node
   revisit flag RF = 0
2. A new solution z (generated by GA) is presented to Ar
3. Curr_node := root
4. If (Curr_node has two child nodes)
  {
     Compare z with child node x and y;

     If (z = x) or (z = y)
        RF = 1

     Define the comparing dimension j:
        j = arg max_{k∈[1,D]} d(x, y|k)
     If    d(x,z | j) ≤ d(y,z | j)
           Curr_node := child node x
     Else
           Curr_node := child node y
     Repeat (Step 4)
  }
  Else
  {
     If    (RF = 0)
     {
           Insert a child node to Curr_node that records z
           Finish
     }
     Else
     {
           Create a child node by randomly mutating z
           within the sub-region of Curr_node
           Finish
     }
  }
```

Note that NrGA and cNrGA naturally avoid generating any out-of-bound solutions. This is an advantage over some other methods that may generate out-of-bound solutions and need to define an extra repair operator to change the solutions back to valid ones.

For more details on the tree construction as well as a working example, please refer to [5]. The part in **bold** is the only modification needed to Algorithm A.2 of [5]. Note that it is much simpler to the full NrGA algorithm – Algorithm A.3 of [5].

III. Continuous Non-Revisiting Genetic Algorithm

cNrGA is a real coded genetic algorithm. Instead of bit representation, each gene x of an individual in cNrGA is represented by a real value, i.e. $x \in \Re$. For a *D*-dimensional objective function, an individual **x** of cNrGA is a *D*-dimensional real valued vector, i.e. $\mathbf{x} \in \Re^D$, and the search space *S* of cNrGA is a *D*-dimensional continuous space, i.e. $S \subset \Re^D$. The precision of the solution obtained by cNrGA, rather than predefined by the user, is the precision of the floating point number represented in the computer.

The flow of CNrGA, composing of GAM and AMM, is shown in Figure 1. The algorithm starts by initializing the population pool. Then, the offspring pool *O* is generated by genetic operators in GAM. Note that every offspring **x** in *O* is then adaptively mutated by AMM. In this mutation, AMM searches for the partition h_x of **x** (see *Definition 1* below for details). Afterwards, **x** is mutated to **x'** where **x'** is randomly selected from h_x.

Definition 1: The partition of **x**

Suppose $\mathbf{x} \in S$ is a solution in the search space *S*, and *S* is partitioned as the sub-region set $H = \cup_i h_i$ by Density Tree, we define the partition (sub-region) $h_x \subseteq H$ as the '*partition of* **x**' if h_x is represented by a leaf node of Density Tree.

After evaluating the offspring, the distribution of the evaluated solutions is changed, and the Density Tree responds

to this change by inserting all individuals in *O* to the tree. The individuals in the offspring pool and the parent pool are then selected to form the new population pool. The reproduction and selection processes are repeated until the termination criterion is satisfied.

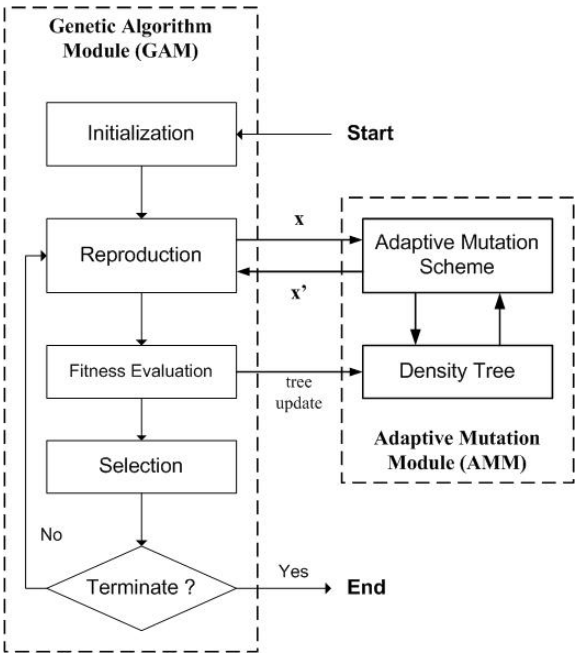

Figure 1. Block diagram of cNrGA

In summary, cNrGA mutates offspring according to the density of the evaluated solutions. For an offspring **x** where the solution-density at **x** is low, **x** is preferred to be mutated with a large step size. Conversely, cNrGA performs a small mutation on **x** if the solution-density at **x** is high. In addition, because of the space partitioning scheme, the partition size is small (large) if the corresponding evaluated solution is close to (far from) its neighbor. Thus, the density of the evaluated solution at **x** can be estimated by the partition size of **x**.

The behavior of the adaptive mutation scheme is elaborated as follows: Initially, the Density Tree consists of only the root node and offspring is randomly assigned in the search space. Also, in the beginning of the evolution, since the number of evaluated solutions is small, the sizes of the partitions and hence the possible mutation step sizes are relatively large. The crossover and adaptive mutation operators act as a far-search operator which explores the search space. As more generations goes by, it is expected that the fittest individuals converge to a certain region *X*. This convergence will increase the solution-density at *X*, to which the scheme keeps sub-dividing the partitions at *X*. It results in a small mutation step such that a near search is performed. In conclusion, the mutation step size adjusts adaptively, governed by the search dynamics rather than a predefined rule; and the operation is completely parameter-less, i.e. no additional control parameter is needed. It performs far search when the partition size is large and/or the number of revisits is small and vice versa. Thus, the parameter-less adaptive mutation scheme obtains the step size from both the spatial information (i.e., the density of the evaluated solutions) and the temporal information (i.e., the convergence in the sense of the number of revisits) of the evolution.

IV. Experimental Results

A. Objective functions

A real valued function set $\mathbf{F} = \{f_1(\mathbf{x}), f_2(\mathbf{x}),\ldots, f_{14}(\mathbf{x})\}$ consisting of 14 functions are employed to illustrate the performance of cNrGA. The 14 test functions are as follows:

1. Sphere function
2. Schwefel's problem 2.22
3. Schwefel's problem 1.2
4. Schwefel's problem 2.21
5. Generalized Rosenbrock function
6. Quartic function
7. Generalized Rastrigin function
8. Generalized Griewank function
9. Generalized Schwefel's problem 2.26
10. Ackley function
11. Shekel's Foxholes function
12. Six-Hump Camel-Back function
13. Branin function
14. Goldstein-Price function

They are well known benchmark test functions taken from [15]. The mathematical forms, the search space and the optima of these functions are given in Table 1.

The first six functions are uni-modal functions; the remaining eight are multi-modal functions designed with a considerable amount of local minima. Meanwhile, the dimensions of the first ten functions are adjustable while the dimensions of $f_{11} - f_{14}$ are fixed at two. All functions with the exceptions of f_9, f_{11}, f_{12} and f_{13}, have the global minimum at the origin or very close to the origin. Simulations are carried out to find the global minimum of each function.

B. Test algorithms

In this section, we compare the optimal fitness found by cNrGA with five benchmark evolutionary algorithms. The search spaces of all test algorithms are continuous. The design and settings of cNrGA and the algorithms for comparison are summarized below.

Test algorithm 1 – Canonical real coded GA (RC-GA): RC-GA is tailored for optimization in real-valued search spaces. The genes are real-valued object parameters; evolution operates on the *natural* representation, i.e. each gene is represented by a 64-bit floating point number. Two RC-GA genetic operators employed in this paper are summarized as follows:

Definition 2: Uniform crossover of RC-GA

Suppose $\mathbf{X_1} = [x_{1,1}, x_{1,2}, \ldots, x_{1,D}]$ and $\mathbf{X_2} = [x_{2,1}, x_{2,2}, \ldots, x_{2,D}]$ are chosen from the population uniformly at random. The corresponding crossover-generated offspring pair $\mathbf{O_1} = [o_{1,1}, o_{1,2}, \ldots, o_{1,D}]$ and $\mathbf{O_2} = [o_{2,1}, o_{2,2}, \ldots, o_{2,D}]$ are reproduced for which $(o_{1,I} = x_{1,I} \wedge o_{2,I} = x_{2,I})$ for $I = 1, \ldots, D$ with probability r_x and $(o_{1,I} = x_{2,I} \wedge o_{2,I} = x_{1,I})$ with probability $1 - r_x$, where r_x is the crossover rate.

Definition 3: Mutation of RC-GA

Given an individual $\mathbf{X}$, the corresponding mutation-generated offspring $\mathbf{O}$ is defined as $\mathbf{O} = \mathbf{X} + \mathbf{N}(\sigma_m)$ where $\mathbf{N}(\sigma_m)$ is a 1 by D vector consisting of D Gaussian random variables with zero mean and standard deviation σ_m. σ_m is denoted as mutation standard deviation.

Test algorithm 2 – Continuous non-revisiting genetic algorithm (cNrGA): cNrGA is a real-coded GA with an adaptive mutation scheme. Uniform crossover operator (Definition 2) is used to generate offspring. Also, to illustrate the adaptive mutation effect of the scheme, no mutation operator is used in the offspring reproduction.

Test algorithm 3 – CMA-ES: CMA-ES [16] is an evolution strategy that adapts the full covariance matrix of a normal search (mutation) distribution. It is designed with the emphasis that the same parameters are used in all applications in order to be "parameter-less". The source code of CMA-ES is taken from [16] (Aug. 2007 version).

Test algorithm 4 – Dissipative particle swarm optimization (DPSO): DPSO [17] is a modified particle swarm optimization (PSO) which introduces random mutation that helps particles to escape from local minima. Its formula is described as follows:

$$\text{If } \eta_3 < C_v \text{ then } v_i = \eta_4 \times V\max / C_m$$

where η_3 and η_4 are uniformly distributed random variables in the range [0,1]; C_v is the mutation rate to control the velocity; C_m is a constant to control the extent of mutation; and V_{max} is the maximum velocity.

Test algorithm 5 – PSO with mutation (PSOMS): PSOMS [18] prevents premature convergence according to the averaged similarity between each particle and the historical best particle explored by the swarm. The clustering degree of the swarm is computed to measure the swarm diversity and the position of a particle is re-initialized if:

$$\eta_4 < \alpha \times c(t) \times s(I,g)$$

where α is a predefined constant, $c(t)$ is the collectivity at the t^{th} generation and $s(I,g)$ denotes the similarity of the i^{th} particle to the current best particle.

Test algorithm 6 – Differential Evolution (DE): Differential evolution [19] is a stochastic parallel search evolution strategy optimization. It is mainly applied to continuous search space. The source code of DE is taken from [20].

For RC-GA and cNrGA, the crossover rate r_x is chosen as 0.5. This is the recommended setting in [21, pg. 48]. Suppose $X = \prod[L_i, U_i]$ for $i = 1,\ldots,D$ is the search space of a D-dimensional objective function, the mutation standard deviation σ_m in RC-GA is set to $0.05max\{U_i\text{-}L_i\}$.

For PSO-class test algorithms, the values of c_1, c_2 are set to 2. The inertia w is linearly decreasing from 1 to 0. The maximum velocity V_{max} is set to $0.1R$ where $R = \max(U_i - L_i)$. The parameters used in DPSO and PSOMS are assigned to be the same as suggested in the original works: the parameters C_v and C_m of DPSO are chosen to be 0.001 and 0.002 respectively. For PSOMS, the parameters d_{min}, d_{max}, β and α are set as $0.001RD^{0.5}$, $0.01RD^{0.5}$, 1 and 3 respectively.

C. Simulation settings

For cNrGA, RC-GA and DE, the population sizes are set to 100. (100+100) selection is used. For CMA-ES, the population size λ is chosen by the suggested setting in [16] (i.e. $\lambda = 4 + \lfloor 3\ln D \rfloor$). For DPSO and PSOMS, the swarm sizes are set to 100 and 100 offspring are reproduced at each generation. To provide a fair comparison amongst the test algorithms, the total number of function evaluations of all algorithms is kept a constant: For functions $f_1 - f_{10}$, cNrGA, RC-GA, DPSO, PSOMS and DE are terminated after 400 generations. CMA-ES is terminated after 40,000 function evaluations, i.e., the total number of fitness evaluations of all the algorithms is fixed at 40,000. Similarly, for functions $f_{11} - f_{14}$, the total number of fitness evaluations is fixed at 1,000. The swarm sizes of DPSO and PSOMS are set to 50. The population sizes of cNrGA, RC-GA and DE are set to 50 also. CMA-ES is terminated after 1,000 function evaluations.

All test functions with the exception of $f_{11} - f_{14}$, which are two-dimensional, are tested with dimensions 30 and 40. Since the test algorithms are stochastic, their performance on each test function is evaluated based on statistics obtained from 100 independent runs. All simulations are done on a PC with 3.2GHz CPU and 1GB memory. The test algorithms: cNrGA, RC-GA, DPSO and PSOMS are implemented in C language. CMA-ES uses source code in [16] and MATLAB version 6.1. DE uses source code in [20] and MATLAB version 6.1.

D. Simulation results

The detailed simulation results are reported in Table 2. Figure 2 presents a summary of the results. The shaded cells in the figure indicate that the corresponding test algorithm is the best algorithm on a particular test function at a particular function dimension. The values inside the table cells for cNrGA indicate the ranking of cNrGA on a particular test function at a particular function dimension when it is not the best algorithm. From the results, it can be observed that CMA-ES is the clear winner for uni-modal functions f_1 - f_6, while the performance of cNrGA is average. (The average

rank is 3.5 out of 6). However, for multi-modal functions f_7 - f_{14}, cNrGA outperforms every other algorithms including CMA-ES. It ranks 1st in 7 out of 8 functions and is only second to CMA-ES in one function.

The detailed simulation results (mean and standard deviation) are listed in Table 2. It lists the average and the standard deviation (inside brackets) of the optimal fitness for 100 trials. A value in bold indicates that the corresponding algorithm is the best amongst the algorithms on a particular test function at a particular function dimension.

E. Overhead

cNrGA has two overheads, namely, the memory storage cost $N = O(N)$ of storing all N visited solutions, and the $O(NlogN)$ computational effort for finding the partition.

As discussed above, memory is not a problem in many practical applications, for which fitness evaluation is expensive and/or time consuming (e.g. 1 sec. to 1 day); and fitness evaluation cost is substantially higher than the solution generation cost – the so called black box optimization scenario [12]. Similarly, N is not unmanageably large when cNrGA is hybridized with an application dependent local improvement heuristic in a memetic algorithm. Finally, the amount of memory available is rapidly increasing due to Moore's Law.

Concerning the second overhead, the $O(NlogN)$ computational effort is only moderate, especially against the backdrop that N is not extremely large in engineering problems and memetic algorithms. This additional effort offers an attractive compensation however: a parameter-less adaptive mutation operator. It is well known that parameter control and setting is tricky and extremely difficult [6, 7]. Thus the ability to adaptively mutate without parameter is effectively delivering an algorithm that discovers its control and settings *from* the search process. This is a worthy merit and arguably, is well worth the extra computational effort. In the larger context, the partitioning scheme offers a novel non-parametric probability distribution that encodes all past experiences, and is potentially extremely useful for making good future search decisions.

In this experiment, $N = 40{,}000$. Using a PC with 3.2 GHz CPU and 1 GB memory, the size of the BSP tree with 40,000 nodes is just 0.0894% of the memory, assuming that each node occupies 24 bytes. The worst case computational overhead of cNrGA compared with RC-GA amongst the 14 test functions (24 test cases) is merely 0.6227 sec.

V. Conclusion

Non-revisiting genetic algorithm (NrGA) [4, 5] can adaptively mutate offspring according to the search history. Though it is a powerful mutation scheme, it is limited to search in discrete space. To obtain a more precise optimum, a real valued optimization algorithm is necessary. In this paper, we modify the NrGA model to handle continuous search space. The new genetic algorithm, namely continuous non-revisiting genetic algorithm (cNrGA), integrates a real coded genetic algorithm with a memory unit that encodes and intelligently uses the search history. cNrGA has the following properties:

1. cNrGA adaptively suggests mutation vectors based on the distribution of the evaluated solutions.
2. Because of the partitioning scheme, the distribution of partition sizes follows the solution-density in the search space; a fast density approximation is obtained. This can be considered as a form of non-parametric probability distribution of fitness whose data comes from the whole search history. This view leads to interesting connections with estimation of distribution algorithms [22].
4. The adaptive mutation naturally avoids the out-of-bound solution problem in bounded real valued optimization algorithms.
5. The overhead of cNrGA is reasonably small or insignificant for most applications.

cNrGA is a bi-modulus evolutionary algorithm consisting of the genetic algorithm module (GAM) and the adaptive mutation module (AMM). GAM performs the general genetic operators such as crossover and selection, while AMM adaptively mutates every offspring generated by GAM according to the density of the evaluated solutions. AMM stores all evaluated solutions by a tree-structure archive. The archive divides the search space into non-overlapped hyper-rectangular partitions according to the distribution of the evaluated solutions. For a point in the search space with high solution-density, it infers a high probability that the point is close to the optimum and hence a near search (exploitation) is suggested. Alternatively, a far search (exploration) is recommended for a point with low solution-density.

In the experiment section, we examine cNrGA on fourteen benchmark problems, including both uni-modal and multi-modal functions. The dimensions of the test functions are from 2 to 40. We compare the performance of cNrGA with five bench mark real coded evolutionary algorithms. It is found that for multi-modal functions, cNrGA outperforms all the other algorithms, while it does not perform as well for uni-modal functions. This suggests that cNrGA should be used in the optimization of multi-modal functions, which represents the harder and more challenging application problems.

References

[1] Glover and M. Laguna, *Tabu Search*. Kluwer Academic Publishers, 1997.

[2] R.G. Reynolds, "An overview of cultural algorithms", in *Advances in Evolutionary Computation*, McGraw Hill Press, 1999.

[3] J.D. Farmer, N. Packard and A. Perelson, "The immune system, adaptation and machine learning", *Physica D*, vol. 2, pp. 187-204, 1986.

[4] S. Y. Yuen and C. K. Chow, "A Non-revisiting genetic algorithm", in *Proc. IEEE CEC*, pp. 4583 – 4590, 2007.

[5] S.Y. Yuen and C.K. Chow, "A Genetic algorithm that adaptively mutates and never revisits," *IEEE Transactions on Evolutionary Computation*, to be published.

[6] A.E. Eiben, R. Hinterding, and Z. Michalewicz, "Parameter control in evolutionary algorithms", *IEEE Transactions on Evolutionary Computation*, vol. 3, no. 2, pp. 124-141, 1999.

[7] F.G. Lobo, C.F. Lima and Z. Michalewicz (Eds.), *Parameter setting in evolutionary algorithms*. Springer, 2007.

[8] S.Y. Yuen and C.K. Chow, "A Non-revisiting simulated annealing algorithm," in *Proc. IEEE CEC*, pp. 1886-1892, 2008.

[9] C.K. Chow and S.Y. Yuen, "A Non-revisiting particle swarm optimization," in *Proc. IEEE CEC*, pp. 1879-1885, 2008.

[10] C.W. Sung and S.Y. Yuen, "On the analysis of the (1+1) evolutionary algorithm with short-term memory," in *Proc. IEEE CEC*, pp. 235-241, 2008.

[11] C.W. Sung and S.Y. Yuen, "On the analysis of (1+1) evolutionary algorithms with memory," *Evolutionary Computation*, submitted.

[12] I. Wegener, *Complexity theory*, Springer 2005.

[13] S.Y. Yuen and C.K. Chow, "Applying non-revisiting genetic algorithm to traveling salesman problem," in *Proc. IEEE CEC*, pp. 2217-2224, 2008.

[14] S.Y. Yuen and C.K. Chow, "A study of operator and parameter choices in non-revisiting genetic algorithm, " in *Proc. IEEE CEC*, 2009.

[15] X. Yao, Y. Liu, and G. M. Lin, "Evolutionary programming made faster," *IEEE Trans. Evolutionary Computation*, vol. 3, no. 2, pp. 82–102, 1999.

[16] N. Hansen, "The CMA evolutionary strategy: A tutorial", Technical Report, code version: 31 Aug. 2007. Link: www.bionik.tu-berlin.de/user/niko/cmatutorial.pdf

[17] X. F. Xie, W. J. Zhang, Z. L. Yang, "A dissipative particle swarm optimization," in *Proc. IEEE CEC,* pp. 1666 – 1670, 2002.

[18] J. Liu, X. Fan and Z. Qu, "An improved particle swarm optimization with mutation based on similarity," in *Proc. IEEE Int. Conf. on Natural Computation*, pp. 824 – 828, 2007.

[19] R. Storn and K. Price, "Differential evolution—A simple and efficient adaptive scheme for global optimization over continuous spaces," Berkeley, CA, Tech. Rep. TR-95-012, 1995.

[20] Differential evolution source code link: http://www.icsi.berkeley.edu/~storn/code.html

[21] A. E. Eiben and J. E. Smith, *Introduction to evolutionary computing*, Springer 2003.

[22] P. Larrañaga and J.A. Lozano, *Estimation of distribution algorithms: a new tool for evolutionary computation*, Norwell, MA: Kluwer, 2002.

TABLE 1 DETAILS OF THE FOURTEEN TEST FUCNTIONS.

test function	*mathematical form*	*range*	*optimum*
1. Sphere function	$f_1(\mathbf{x}) = \sum_{i=1}^{D} x_i^2$	$[-100, 100]^D$	$[0,0,\ldots,0]$
2. Schwefel's problem 2.22	$f_2(\mathbf{x}) = \sum_{i=1}^{D} \lvert x_i \rvert + \prod_{i=1}^{D} \lvert x_i \rvert$	$[-100, 100]^D$	$[0,0,\ldots,0]$
3. Schwefel's problem 1.2	$f_3(\mathbf{x}) = \sum_{i=1}^{D} \left(\sum_{j=1}^{i} x_j \right)^2$	$[-100, 100]^D$	$[0,0,\ldots,0]$
4. Schwefel's problem 2.21	$f_4(\mathbf{x}) = \max_{i \in [1,D]} \lvert x_i \rvert$	$[-100, 100]^D$	$[0,0,\ldots,0]$
5. Generalized Rosenbrock function	$f_5(\mathbf{x}) = \sum_{i=1}^{D-1} [100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2]$	$[-29, 31]^D$	$[1,1,\ldots,1]$
6. Quartic function Note: This is a noisy fitness function. There is a random measurement noise in each fitness evaluation.	$f_6(\mathbf{x}) = \sum_{i=1}^{D} i x^4 + random[0,1]$	$[-1.28, 1.25]^D$	$[0,0,\ldots,0]$
7. Generalized Rastrigin function	$f_7(\mathbf{x}) = \sum_{i=1}^{D} [x_i^2 - 10\cos(2\pi x_i) + 10]$	$[-5.12, 5.12]^D$	$[0,0,\ldots,0]$
8. Generalized Griewank function	$f_8(\mathbf{x}) = \frac{1}{4000} \sum_{i=1}^{D} x_i^2 - \prod_{i=1}^{D} \cos \frac{x_i}{\sqrt{i}} + 1$	$[-600, 600]^D$	$[0,0,\ldots,0]$
9. Generalized Schwefel's problem 2.26	$f_9(\mathbf{x}) = -\sum_{i=1}^{D} x_i \sin \sqrt{\lvert x_i \rvert}$	$[-500, 500]^D$	$[420.9687,\ldots, 420.9687]$
10. Ackley function	$f_{10}(\mathbf{x}) = -20\exp\left(-0.2\sqrt{\frac{1}{D}\sum_{i=1}^{D} x_i^2} \right) - \exp\left(\frac{1}{D}\sum_{i=1}^{D} \cos 2\pi x_i \right) + 20 + e$	$[-32, 32]^D$	$[0,0,\ldots,0]$

11. Shekel's Foxholes function

$$f_{11}(\mathbf{x}) = \left[\frac{1}{500} + \sum_{j=1}^{25} \frac{1}{j + \sum_{i=1}^{2} (x_i - a_{i,j})^6} \right]^{-1} \quad \text{where}$$

$$\{a_{i,j}\} = \begin{bmatrix} -32 & -16 & 0 & 16 & 32 & -32 & \cdots & 0 & 16 & 32 \\ -32 & -32 & -32 & -32 & -32 & -16 & \cdots & 32 & 32 & 32 \end{bmatrix}$$

$[-98, 34]^2$ | $[-32, 32]$

12. Six-Hump Camel-Back function

$$f_{12}(\mathbf{x}) = 4x_1^2 - 2.1x_1^4 + \frac{1}{3}x_1^6 + x_1x_2 - 4x_2^2 + 4x_2^4$$

$[-4.91017, 5.0893] \times [-5.7126, 4.2874]$ | $[0.08983, -0.7126]$ and $[-0.08983, 0.7126]$

13. Branin function

$$f_{13}(\mathbf{x}) = (x_2 - \frac{5}{4\pi^2}x_1^2 + \frac{5}{\pi}x_1 - 6)^2 + 10(1 - \frac{1}{8\pi})\cos x_1 + 10$$

$[-8.142, 6.858] \times [-12.275, 2.725]$ | $[-3.142, 12.275]$, $[3.142, 2.275]$, $[9.425, 2.425]$

14. Goldstein-Price function

$$f_{14}(\mathbf{x}) = g(\mathbf{x}) \times h(\mathbf{x})$$

$$g(\mathbf{x}) = 1 + (x_1 + x_2 + 1)^2 \times (19 - 14x_1 + 3x_1^2 + 6x_1x_2 + 3x_2^2)$$

$$h(\mathbf{x}) = 30 + (2x_1 - 3x_2)^2 \times (18 - 32x_1 + 12x_1^2 + 48x_2 - 36x_1x_2 + 27x_2^2)$$

$[-2, 2] \times [-3, 1]$ | $[0, -1]$

	Uni-modal										*Noisy Uni-modal*		*Multi-modal*											
Function	f_1		f_2		f_3		f_4		f_5		f_6		f_7		f_8		f_9		f_{10}		f_{11}	f_{12}	f_{13}	f_{14}
D	*30*	*40*	*30*	*40*	*30*	*40*	*30*	*40*	*30*	*40*	*30*	*40*	*30*	*40*	*30*	*40*	*30*	*40*	*30*	*40*	*2*	*2*	*2*	*2*
cNrGA	2	2	2	2	5	5	5	5	4	4	3	3			2	2								
DE																								
RC-GA																								
CMA-ES																								
DPSO																								
PSOMS																								

Figure 2. Indicators of the best test algorithm in the experiments: The cell with grey color represents that the corresponding test algorithm outperforms the others for a particular function and a particular function dimension.

TABLE 2 THE AVERAGE AND THE STANDARD DEVIATION OF THE BEST FITNESS VALUES FOUND BY CNRGA, DE, RC-GA, CMA-ES, DPSO AND PSOMS.

	D	cNrGA	DE	RC-GA	CMA-ES	DPSO	PSOMS
f_1	30	0.89 (0.93)	32.92 (13.141)	254.79 (32.15)	**0.00 (0.00)**	72.96 (4.50)	45.47 (3.96)
	40	2.14 (1.66)	60.95 (20.510)	493.76 (41.13)	**0.00 (0.00)**	130.12 (5.21)	122.11 (5.95)
f_2	30	1.90 (1.30)	26.47 (5.88)	17.05 (0.83)	**0.00 (0.00)**	7.78 (1.06)	11.95 (4.57)
	40	4.11 (1.73)	38.80 (7.16)	27.46 (1.31)	**0.00 (0.00)**	35.76 (7.80)	53.43 (8.77)
f_3	30	4644.12 (1882.7)	3898.78 (1840.1)	7954.00 (1466.7)	**1346.70 (3500.0)**	1638.09 (21.66)	2045.42 (24.49)
	40	17706.3 (5232.9)	9195.87 (4773.8)	23477.7 (4088.3)	**2615.4 (39090.9)**	3156.77 (28.80)	4077.51 (32.83)
f_4	30	41.40 (7.91)	18.69 (4.34)	17.52 (1.10)	100.00 (0.00)	**9.70 (1.26)**	13.84 (1.79)
	40	50.54 (6.23)	22.31 (3.84)	23.04 (1.12)	100.00 (0.00)	**11.41 (1.28)**	16.41 (1.66)
f_5	30	14479 (21291)	39655.6 (19994)	194518.3 (3840)	10117.10 (11872)	8407.94 (65.0)	**6674.39 (67.6)**
	40	34641 (42164)	83083.2 (30407)	591738.5 (9608)	20117.20 (90419)	**18691.25 (83.4)**	21132.67 (110.4)
f_6	30	8.75 (0.50)	0.544 (0.42)	9.71 (0.50)	**0.20 (0.09)**	9.85 (0.82)	9.99 (0.82)
	40	13.30 (0.66)	1.725 (1.33)	14.91 (0.69)	**0.24 (0.08)**	14.54 (0.87)	14.50 (0.95)
f_7	30	**28.92 (6.79)**	113.55 (23.54)	190.63 (11.44)	51.09 (13.70)	122.50 (4.12)	99.83 (4.33)
	40	**45.71 (9.32)**	183.83 (26.42)	287.74 (10.77)	74.22 (13.30)	191.58 (4.87)	162.29 (5.11)
f_8	30	1.79 (1.02)	32.46 (11.32)	15.62 (1.51)	**0.00 (0.00)**	3.53 (0.77)	2.74 (0.76)
	40	3.10 (1.74)	65.46 (20.97)	28.86 (2.18)	**0.00 (0.00)**	5.84 (1.04)	5.37 (1.12)
f_9	30	**-12982.0 (272.9)**	-9191.79 (837.59)	-9506.40 (283.3)	-5406.80 (92.6)	-4256.03 (21.3)	-6562.55 (25.8)
	40	**-16728.2 (415.6)**	-11165.3 (905.5)	-11369.60 (434.7)	-7187.40 (184.1)	-5079.23 (24.5)	-7718.63 (29.0)
f_{10}	30	**3.93 (1.19)**	20.383 (0.248)	9.15 (0.30)	18.77 (4.77)	5.66 (0.74)	6.04 (0.95)
	40	**5.42 (1.52)**	20.538 (0.289)	11.28 (0.28)	19.37 (3.42)	6.86 (0.81)	8.18 (0.96)
f_{11}	2	**1.331 (0.839)**	2.248 (1.725)	1.924 (1.128)	269.891 (244.52)	1.928 (1.03)	1.0080 (0.31)
f_{12}	2	**-1.031 (0.001)**	-1.030 (0.001)	-1.028 (0.002)	-0.999 (0.160)	-1.014 (0.131)	-1.0310 (0.01)
f_{13}	2	**0.398 (0.001)**	0.399 (0.001)	0.400 (0.001)	0.399 (0.001)	1.997 (1.631)	0.3982 (0.01)
f_{14}	2	**3.001 (0.050)**	3.011 (0.004)	3.031 (0.031)	13.774 (25.693)	3.286 (0.786)	3.0230 (0.01)