Introducing full memory in Genetic Algorithms

A. E. Charalampakis

Proc. 2nd International Conference on Soft Computing Technology in Civil, Structural and Environmental Engineering (CSC2011), Chania, Greece; 2011.

Abstract

The concept of using full memory in Genetic Algorithms (GAs) is examined. This is achieved by an encapsulated operator that is placed between the GA and the subroutine evaluating the objective function. The operator records all evaluated solutions and selectively replaces the solutions requested by the GA with similar ones taken from the memory. Significant increase in performance is observed, which is evident even at the early stages of evolution, in accordance with the "Birthday Problem". Implementation with Standard GA shows great promise, while the encapsulation of the code facilitates implementation with other Evolutionary Algorithms.

Keywords: genetic algorithms, external memory, evolutionary computation.

1 Introduction

Evolutionary Algorithms (EAs) are biologically inspired stochastic algorithms that have been successfully applied to a wide variety of problems [1]. The most widely known type of EAs are the Genetic Algorithms (GAs), which were initially conceived by John Holland [2] and later employed in virtually any problem imaginable.

Most EAs utilize some kind of short- or long-term memory to explore the Design Space (DS) and produce better solutions. In the case of Standard GA (SGA) the algorithm is practically memoryless, as it simply evolves a population of candidate solutions. Elitism is introduced as a short-term memory operator, linking the current to the previous generation only, which is shown to improve performance [3]. In the case of Particle Swarm Optimization (PSO) algorithms [4], each particle retains memory of its personal best-so-far position while the whole swarm also retains memory of its global best-so-far position. Optimization is achieved through Newtonian dynamics between the particles of the swarm. In this case, the memory is

long-term, since it extends to the beginning of evolution, but it is not full since only the best results (local and global) are retained and utilized.

EAs have been applied in numerous engineering problems, including optimization and identification, to great success. In real life, most engineering problems feature a medium to high number of design variables and a computationally expensive black-box objective function. In these cases, it is safe to assume that the computational cost of the EA itself is negligible when compared to the cost of the function evaluations required for evolution. Indeed, a single function evaluation may require some minutes or even hours in the most advanced computer systems [3], [5].

To cope with this problem, several methods have been developed. Herein, these methods are classified into three broad categories based on their main motivation, as follows:

1) Those that aim at reducing the complexity of the problem. The most direct method is *problem approximation* [6], which reduces the computational cost of the objective function. A typical example is the use of coarser meshes for finite element analyses. Other methods tackle the high dimensionality or take advantage of problem-specific characteristics through *problem partitioning* [7], *screening* [3], *space reduction* [3], etc.
2) Those that use approximation to reduce the number of necessary function evaluations. This can be achieved using *functional approximation* [8], *fitness inheritance* [9] or *clustering* [10].
3) Those that aim at improving the performance of the EA itself. Examples include elitism [3], variable population size and partial initialization of the population as in Sawtooth-GA [11], dynamic change of crossover and mutation probabilities [3], hybrid methods [12], to name a few.

Seen from different points of view, some methods may belong to different categories while combination of methods may produce even better results.

The initial motivation of this work lies in the methods of the third category. It was observed that GAs often re-evaluate a solution that has already been evaluated in the past. This is unacceptable, especially in case the computational cost of the objective function is significant. The removal of duplicate genotypes has been confirmed to increase performance by numerous researchers. Mauldin [12] was probably the first to observe its benefits. He devised a "uniqueness operator" which was employed in a binary-encoded GA and prevented duplicate and similar genotypes in the population. Later, Ronald [14] elaborated on this and presented a hash tagged duplicate removal algorithm, which reduced the complexity of comparing genotypes to $O(L_c)$ instead of $O(L_c P^2)$ of the traditional comparison algorithms (L_c=chromosome length, P=population size). To extend the search beyond the current generation, an external memory operator is required. Indeed, Povinelli and Feng [15] employed a small hash table to store the most recently evaluated chromosomes. Kratica [16] employed a fixed-size cache to store all evaluated individuals. Recently, Yuen and Chow [17] presented a non-revisiting GA (NrGA) which uses complete memory in the form of a binary space partitioning tree. In this, revisits are completely eliminated.

Apart from the removal of duplicate genotypes (or prevention of their re-evaluation), the *reuse* of stored information has also been investigated in the literature. One example is the "Hall of Fame", introduced by Rosin and Belew [18], which contains the best individuals from previous generations. Reintroducing genotypes which were successful in the past is important in the concept of competitive co-evolution. The reason is that these genotypes may also be successful in the future and, once extinct, they are difficult to rediscover. In static optimization problems, however, reintroducing extinct genetic material in the hope of rediscovering successful solutions is usually not very effective because old genotypes were strictly worse according to the fitness measure [18].

Nevertheless, the concept of chromosome reuse in static optimization problems has been investigated by Acan and Tekol [19]. They observed that a significant number of chromosomes of average fitness remain unused, in the sense that they do not actually participate in the recombination process. As an estimate, this number is about equal to 74% of the population in the case of the Ackley function with 20 variables and tournament selection. To cope with this fact, a number of chromosomes of above-average quality, which are not utilized for recombination in the current generation, are inserted into a chromosome library of fixed size and selectively utilized at later stages of evolution. The motivation in [19] is to trace some of the untested search directions in the recombination of potentially promising solutions. Improved performance with respect to conventional GAs is also reported.

A similar concept, under the term "evolutionary reincarnation", was examined by Prime and Hendtlass [20]. Two "islands", or separate populations, are utilized in tandem. The first holds the conventional population while the second contains information taken from earlier populations, i.e. extinct genetic material which may be "reincarnated" based on a suitable strategy. The motivation in [20] is to preserve diversity, by allowing some backtracking to occur along the path to the solution, thus providing an escape mechanism from evolutionary "dead-ends".

Common aim in the aforementioned methods is the increase of diversity among the population in order to improve performance. A different approach is presented herein. A simple complete-memory operator is proposed which records all evaluated individuals. Henceforth, this operator will be referred to as "the Registrar", while the list of all accurately evaluated solutions will be referred to as "the registry". Unlike other methods that reuse extinct genetic material in order to increase diversity (in fact, *opposite* to them), the Registrar does so to avoid function evaluations. Unlike other methods that prevent revisits, the Registrar not only allows revisits but actually *encourages* them, as follows: when a function evaluation is requested by any EA, the new solution is compared to the ones in the registry. If some similarity criteria are met, the candidate solution is replaced by the most similar (and already evaluated) solution. In the opposite case, the function evaluation is performed as usual and the result is stored into the registry for later use.

It will be shown that the Registrar is effective both at the early and later stages of evolution. In any case, a function evaluation that is avoided due to an exact match from the registry cannot be but beneficial for performance [17]. The Registrar is very effective even at the early stages of search space exploration, although few individuals have been stored into the registry. This beneficial effect is more

important because it is in accordance with the limited computational time budget associated with computationally expensive problems.

2 The "Birthday Problem" or "Birthday Paradox"

The effectiveness of the Registrar at the early stages of search space exploration is best explained by the "Birthday Problem", which is one of the most famous counter-intuitive problems in probability [21]. The problem is stated as follows: How many persons must a group contain in order to achieve a better than even (>50%) probability that two members share the same birthday? (We assume that the birthday of a person forms a DS with 365 distinct and equally probable values). Most people think that the answer is about half the number of days in a year (183), but this is the correct answer to another problem: How many people with *different* birthdays are needed in order to achieve a better than even chance that one of them shares *your* (or any other given) birthday? The correct answer to the Birthday Problem is a remarkably small number; it can be proved that only 23 persons suffice [21].

Now consider an EA that is sequentially evaluating potential solutions, which are stored into a memory pool. The analogy shows that, although the size of the DS may be comparatively huge, it is actually very probable that the pool contains two identical solutions, one of which could have been avoided in the process. (The possibility of three- or more-member match is small compared to the case of two-member match and, although beneficial, it is neglected in the reasoning). Note the subtle difference: although the probability of match between each new random solution and one of the *unique* members of the pool is indeed small (which is in accordance to the common belief), multiple such "coin tosses" have been performed in order to fill the pool with these unique members. In other words, it becomes increasingly difficult to fill the pool with unique members. Computer simulations of the workings of the Registrar have confirmed that the possibility of two-member match is in accordance with the results of the Birthday Problem.

Trying to take advantage of this ideal situation, the well-known "curse of dimensionality" plagues our effort. Omitting details, this is equivalent of requiring two persons to share not only their birthday, but also their year of birth. Thus, the possibility of match quickly deteriorates as the dimensionality of the DS is increased.

On the other hand, two things are supportive: first, for the purposes of evolutionary computation, an exact match may not be required. An already evaluated solution that is "close enough" to the one requested by the EA may be totally satisfying. Omitting details, this is equivalent of requiring two persons to be born on the same week of the year, rather than on exactly the same day. Thus, the effective size of the DS is reduced to 52 (from 365) and it can be derived that only 9 persons are required (instead of 23) to reach a 50% possibility of a "relaxed-criteria match".

The second thing in favour is that the solutions requested by the EA are hardly random. On the contrary, as the evolution progresses, the requested solutions tend to congest in the promising areas of the DS, thus increasing the possibility of match.

Omitting details, this is equivalent of selecting people to fill the group that come from a certain region where it is known that most births occur in August.

It will be demonstrated that, indeed, the DS is more congested than one might think, should we choose to remember all evaluated solutions and relax the match criteria. In fact, the problem is not finding matches but *restricting their number* before they obscure the EA's capability of differentiating between adjacent solutions when focusing in the promising areas of the DS.

It is worth noting that an EA featuring the Registrar with exact match criteria is actually a non-revisiting version of the simple EA, because the objective values of already visited points are not evaluated but rather provided by the registry. In accordance with the No Free Lunch theorem [22] and the reasoning in [17], this suggests that the performance of the EA with the Registrar is always superior (or at least equal) to the one of the simple EA. Indeed, since the *same* sequence of potential solutions is encountered, any given point of the common evolution is reached by the simple EA with more (or at least equal) function evaluations. On the other hand, as the match-criteria are relaxed, the number of avoided function evaluations is increased but the evolution is no longer common. Nevertheless it will be shown that, for a given problem, the overall performance of the EA with the Registrar may be tremendously superior to the one of the simple EA.

In illustration, the Registrar will be implemented with SGA. In this study, a limited set of four representative test functions is employed as well as two levels of problem dimensionality, namely 10 and 30 variables. Both genotypic and phenotypic similarity criteria are examined. Based on the results, proof of concept is established and the potential of the proposed approach is demonstrated.

3 The Registrar

For our purposes, we will examine single-objective optimization. Typically, the EA passes the individual that needs to be evaluated as a by-reference argument to a dedicated subroutine. Note that any modification of by-reference arguments is reflected to the caller routine. The subroutine calculates the objective value and returns it to the EA. Our Registrar operator is a fully-encapsulated simple piece of code that is inserted in between, as shown schematically in Figure 1.

When a function evaluation is requested, the operator scans the registry for possible matches based on current similarity criteria. If one or more matches are found, the Registrar replaces the individual by its best match. This replacement is reflected to the actual population of the EA. The Registrar also extracts the corresponding objective value and returns it to the EA, bypassing the subroutine. Thus, a function evaluation is avoided. If no satisfactory matches are found, the call is forwarded to the subroutine for evaluation. Then, the result is intercepted before it is returned to the EA and stored into the registry, together with the associated individual. The memory requirements for storing the individuals and the corresponding objective values are easily accommodated by modern personal computers. Note that the Registrar does not make use of the random number generator.

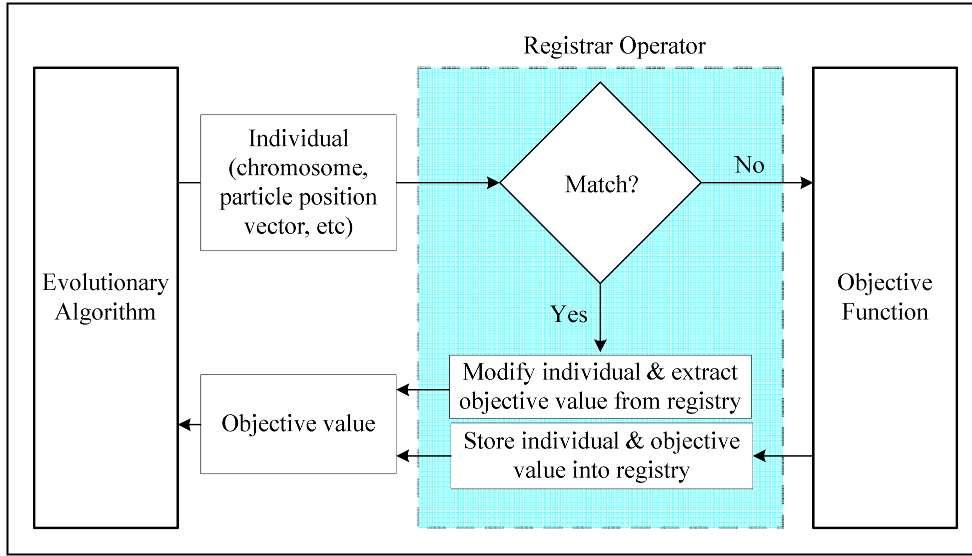

Figure 1: Registrar operator

4 Implementation

The Registrar will be implemented with SGA. The details of the test functions are summarized in the Appendix. It is stressed that our aim is neither to examine the performance of the SGA itself nor to assess whether the chosen parameters are the best for a given problem. In each case, we choose commonly used parameters and seek to examine solely the effect of introducing the Registrar into the SGA. All test cases refer to the best individual ever found and they are based on a series of 50 independent runs using the same sequence of 50 different seeds. The 10 best and 10 worse runs are ignored and the results produced herein are the mean values of the remaining 30 runs.

4.1 SGA

SGA is implemented herein, which can be described by the following pseudo-code:

1. Initialize the population of individuals (chromosomes).
2. Calculate the fitness of each individual in the population.
3. Select individuals to form a new population according to each one's fitness.
4. Perform crossover and mutation.
5. Repeat steps (2) to (4) until some condition is satisfied.

Unless stated otherwise, the parameters of SGA are taken as follows [23]: gene length L_g=10 bits; population size P=100; single crossover with probability 0.7; jump mutation probability $1/P$; creep mutation probability $L_c/N_p/P$ (N_p=number of variables); biased roulette wheel selection and elitism with one individual.

4.2 Match criteria defined by genotype

The first implementation of the Registrar with SGA employs match criteria that are formed in genotypic terms (*RegG*). If two chromosomes c_A and c_B differ in (at most) a certain number of corresponding bits, then there is a "match". The number of different corresponding bits D (i.e. the Hamming distance) is evaluated as:

$$D = \sum_{i=1}^{L_c} \left| c_{A,i} - c_{B,i} \right| \tag{1}$$

where, $c_{A,i}$, $c_{B,i}$ = i^{th} bit of chromosome c_A and c_B, respectively. Thus, the normalized relaxed-match criterion can be expressed as:

$$D \leq MDB \times L_c \tag{2}$$

where, *MDB*= maximum different bits fraction. To draw some initial conclusions, we examine four representative test functions with N_p=10 variables, 10 bits each. Three levels of *MDB* are considered, namely *MDB*=0%, 5% and 10% (RegG/0.00, RegG/0.05 and RegG/0.10, respectively). Since L_c=100 bits, these correspond to no different bits (exact match), 5 and 10 different corresponding bits, respectively.

Figure 2 shows the mean performance, where the Y axis (i.e. the objective value) is drawn in logarithmic scale. When exact matches are required (RegG/0.00), a small increase in performance is noticed with regard to the simple SGA without Registrar (NoReg). However, this includes the elite chromosome as well as the chromosomes that were not altered by crossover and mutation. It is worth noting that the evolution of each run of RegG/0.00 is *identical* to the one of NoReg, provided that the same seed is used. In this case, the gain is exactly equal to the virtual function evaluations that were provided by the registry. As the match criteria are relaxed, a consistent pattern in the performance is observed, i.e. a significantly increased initial performance followed by stagnation. The more relaxed are the match criteria, the more increased is the initial performance and the faster stagnation is observed.

Although the increased performance displayed in Figure 3 is significant, it comes at a cost. In particular, RegG/0.10 at some point *struggles* to find chromosomes that have no matches within the registry. Although the number of total evaluations (= actual evaluations + virtual evaluations from the registry) is increasing normally, practically all of them come from the registry. This results into excessive computational cost without end; in addition, stagnation is observed as the algorithm cannot focus into the promising areas of the DS.

The obvious remedy is the *adaptive reduction of the MDB*, i.e. the tightening of the match criteria. Ultimately, when *MDB*=0, exact matches are sought which cannot be but beneficial [17]. A simple rule is implemented herein: begin evolution with an initial maximum different bits fraction *IMDB*; whenever the rate of virtual over total evaluations exceeds a maximum rate *MR*, multiply the current *MDB* with a number *MLT* in the range (0,1) to reduce it. In order to maintain encapsulation of the code, we measure the rate of virtual over total evaluations fully within the

Registrar. This is accomplished using an integer array R of fixed size L_R. Each time a function evaluation is requested by the EA, the current position in the array is increased by one, or reset to the beginning of the array if it has reached its end. A unity is assigned at the current position if the function evaluation was virtual, a zero otherwise. The current rate CR of virtual over total evaluations is given by:

$$CR = \frac{\sum_{i=1}^{L_R} R_i}{L_R} \tag{3}$$

Herein, L_R=100 and the array R is reinitialized whenever $CR \geq MR$, in order to avoid multiple consecutive triggering of the *MDB* reduction.

The adaptive reduction scheme allows the use of very relaxed initial match criteria without fear of entrapment into endless virtual evaluations. In illustration, Figure 3 shows the mean performance using IMDB=25%, MR=99% and MLT=99% (RegG/0.25/0.99/0.99, respectively), as compared to using a constant *MDB*=10% or no Registrar at all.

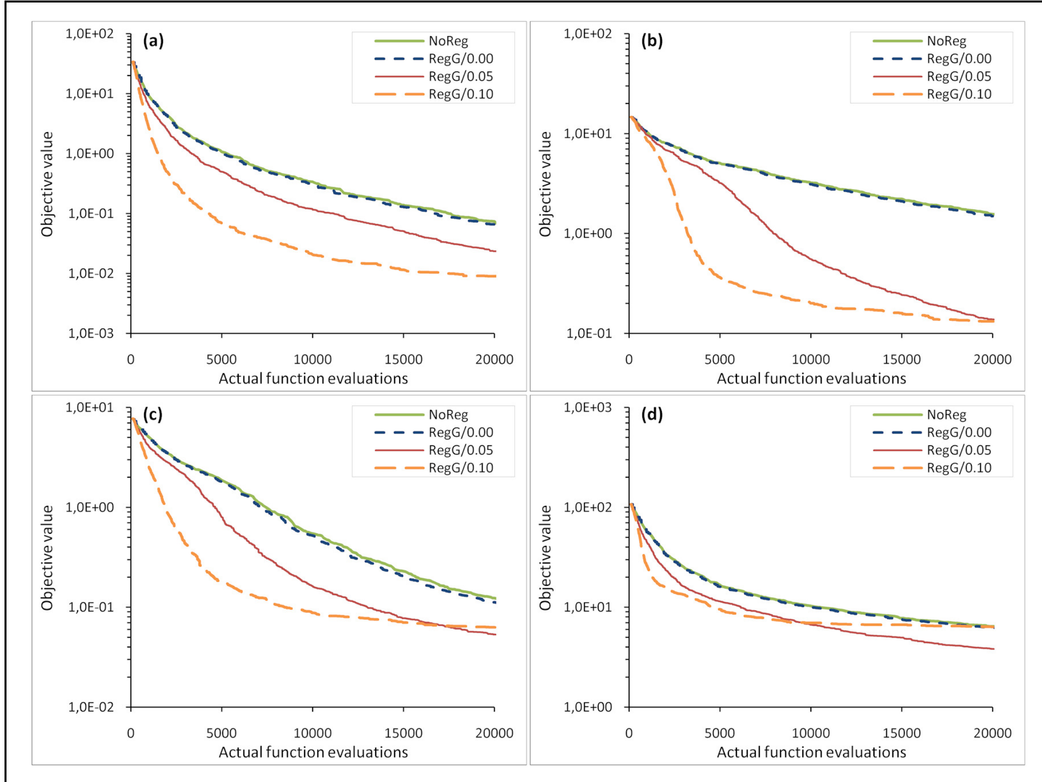

Figure 2: Mean best with genotypic match criteria in problems with 10 variables x 10 bits, (a) F1 Sphere (b) F2 Schwefel (c) F3 Ackley (d) F4 Rastrigin

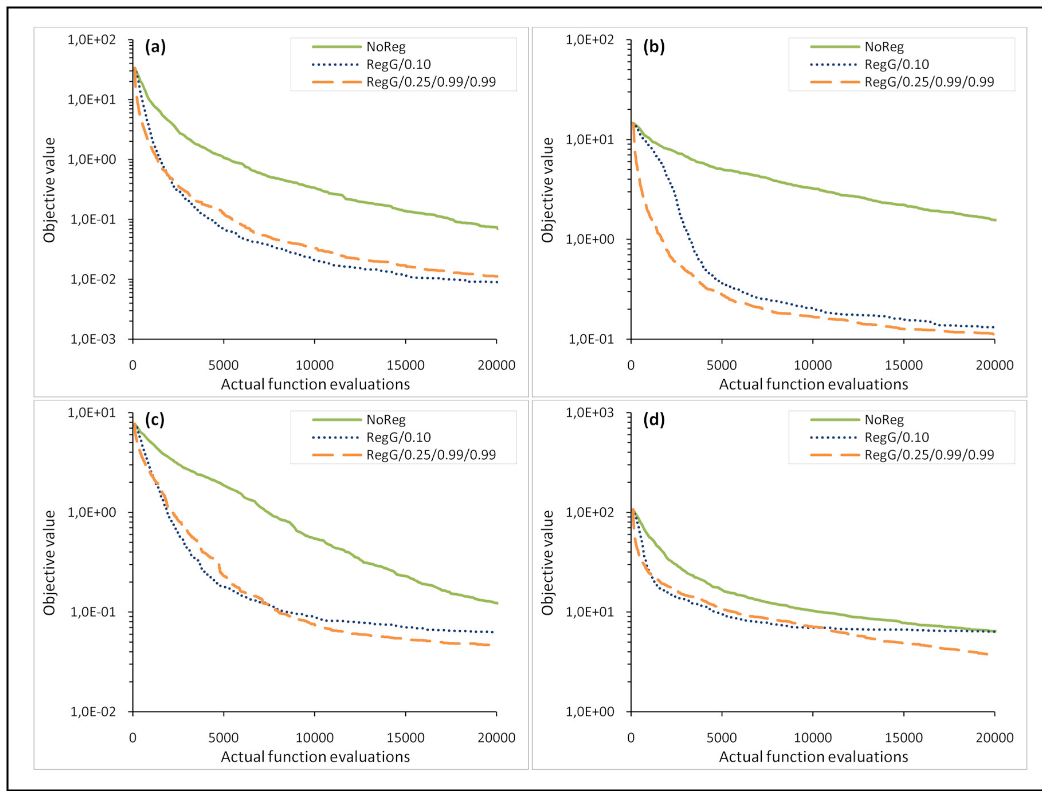

Figure 3: Mean best with genotypic match criteria and adaptive *MDB* reduction (10 variables x 10 bits), (a) F1 Sphere (b) F2 Schwefel (c) F3 Ackley (d) F4 Rastrigin

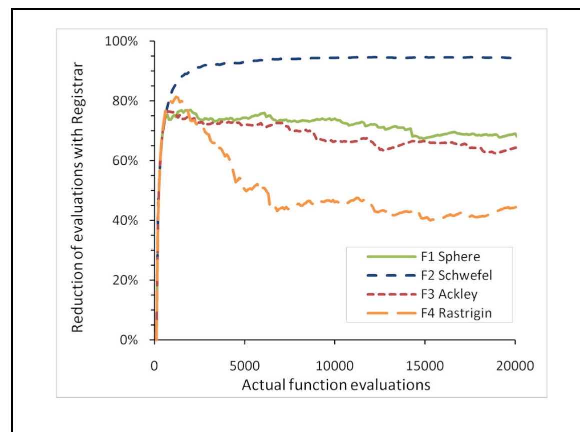

Figure 4: Mean reduction of function evaluations with RegG/0.25/0.99/0.99 for the same level of objective value reached by NoReg (10 variables x 10 bits)

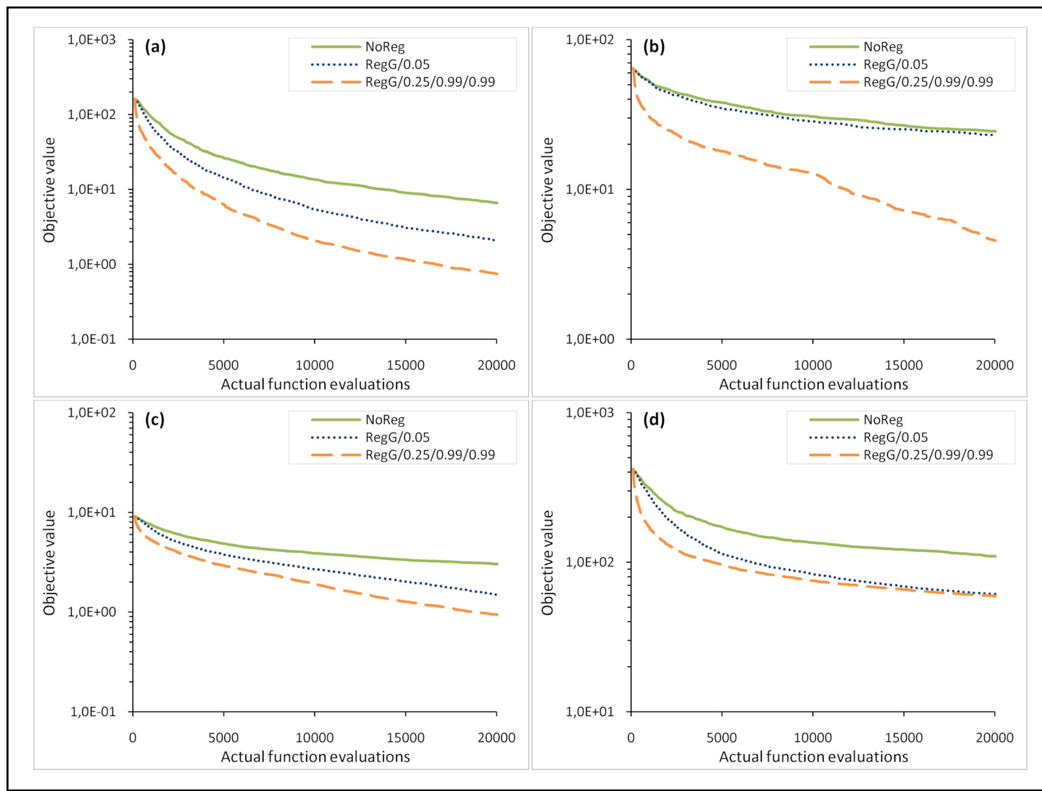

Figure 5: Mean best with genotypic match criteria and adaptive *MDB* reduction (30 variables x 10 bits), (a) F1 Sphere (b) F2 Schwefel (c) F3 Ackley (d) F4 Rastrigin

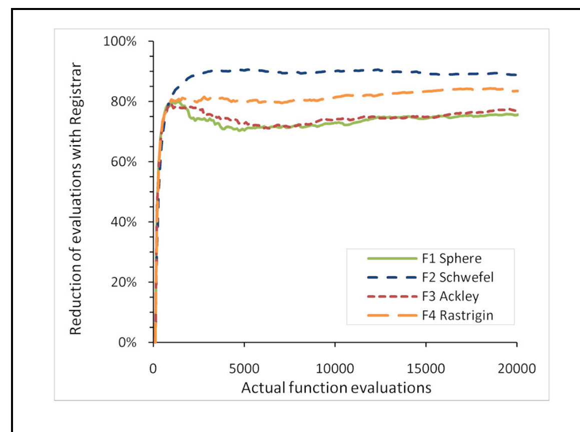

Figure 6: Mean reduction of function evaluations with RegG/0.25/0.99/0.99 for the same level of objective value reached by NoReg (30 variables x 10 bits)

The adaptive reduction scheme seems to impede progress in some parts, when measured strictly by function evaluations. However, it is highly recommended because it greatly reduces the computational cost of each run and ensures its completion. Figure 4 shows the mean reduction of function evaluations achieved when using RegG/0.25/0.99/0.99 and N_p=10 (see Figure 3). Regarding the case of function F2 (Schwefel), it is observed that the reduction reaches 95%, i.e., the same level of best objective value is achieved using only 5% of the function evaluations.

Applying the proposed methodology into problems of high dimensionality (N_p=30 variables, 10 bits each), one also observes significantly increased performance (Figure 5).

Figure 6 shows the mean reduction of function evaluations achieved when using RegG/0.25/0.99/0.99 with N_p=30 (shown in Figure 5). It is observed that a reduction of function evaluations in the range of 70% to 90% was achieved for the four 30-variable test functions (L_c=300).

4.3 Match criteria defined by phenotype

The second implementation of the Registrar refers to SGA with match criteria that are formed in phenotypic terms (RegP). Herein, the "distance" D between two chromosomes c_A and c_B is defined as the sum of the normalized "city block distances" (or normalized Manhattan distances) of their corresponding phenotypic values:

$$D = \sum_{i=1}^{N_p} \frac{\left| x_{A,i} - x_{B,i} \right|}{U_i - L_i} \tag{4}$$

where, $x_{A,i}$, $x_{B,i}$ = i^{th} variable of chromosome c_A and c_B, respectively and U_i, L_i = upper and lower bound of variable x_i, respectively. When D is smaller than a certain threshold, then there is a match. Thus, the criterion can be written as:

$$D \leq MDST \times N_p \tag{5}$$

where, $MDST$= mean distance. An adaptive $MDST$ reduction method is employed, in accordance with the case of genotypic match criteria. We begin evolution with an initial mean distance $IMDST$; whenever the rate of virtual over total evaluations exceeds a maximum rate MR , we multiply the current $MDST$ with a number MLT in the range (0,1) to reduce it.

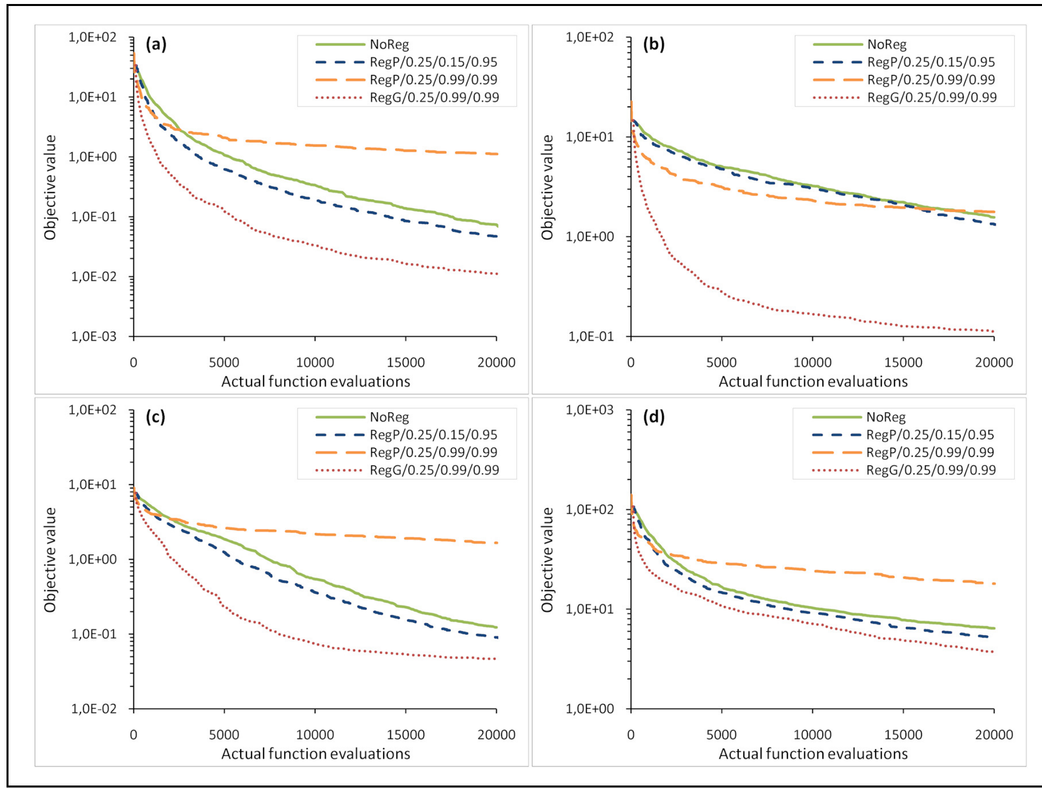

Figure 7: Mean best with phenotypic match criteria and adaptive *MDST* reduction (10 variables x 10 bits), (a) F1 Sphere (b) F2 Schwefel (c) F3 Ackley (d) F4 Rastrigin

It was observed that very strict match criteria lead to small improvement in performance, similar to the exact-match case of Figure 2. On the other hand, a very aggressive configuration (*IMDST*=25%, *MR*=99%, *MLT*=99% = RegP/0.25/0.99/0.99, respectively), which produced excellent results when used with RegG, is clearly inappropriate in this case (Figure 7). A less aggressive configuration was also employed (RegP/0.25/0.15/0.95) which resulted in increased performance with respect to NoReg. In any case, the increase is much smaller compared to RegG/0.25/0.99/0.99 (Figure 3). This is even more pronounced in the case of $N_p = 30$ (Figure 8).

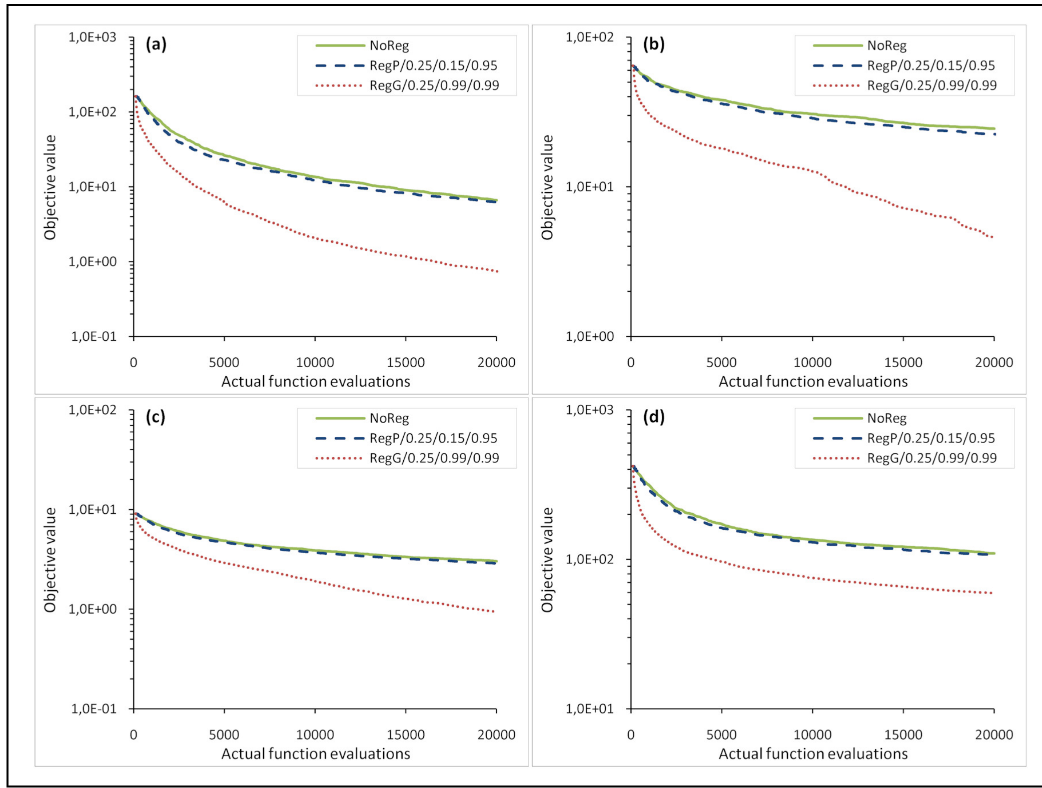

Figure 8: Mean best with phenotypic match criteria and adaptive *MDST* reduction (30 variables x 10 bits), (a) F1 Sphere (b) F2 Schwefel (c) F3 Ackley (d) F4 Rastrigin

5 Conclusions

This study is part of an ongoing research which examines the introduction of full memory into EAs. A fully encapsulated operator, dubbed “the Registrar”, is proposed which is placed between the EA and the objective function. The Registrar records the results of all function evaluations and selectively replaces the individuals requested by the EA with similar ones taken from the registry, based on adaptive match criteria. Thus, function evaluations are avoided in an aggressive manner.

Preliminary results from the application of the Registrar with SGA clearly show that:

(a) the recording and exploitation of the results of all function evaluations is feasible for most real-life problems using modern-day computer systems and

(b) great improvement in performance may be observed.

For the case of SGA, great improvement in performance was observed when defining the match criteria in the genotypic level, in contrast to defining them in the phenotypic level.

Further research will be focused in examining an extended test function test-bed for SGA, as well as real-life engineering problems. Focus will also be given to the implementation of the Registrar into other EAs.

6 Appendix: Test Functions

The test functions employed in this study are presented below. The corresponding 2-dimensional graphs are shown in Figure 9.

6.1 F1: Sphere

$$\begin{aligned} & f(\mathbf{x}) = \sum_{i=1}^{N_p} x_i^2 \\ & -5.12 \le x_i \le 5.11 \ \forall i \in \{1,...,N_p\} \\ & \min f(\mathbf{x}) = f(0,0,...,0) = 0 \end{aligned} \tag{6}$$

6.2 F2: Schwefel

$$\begin{aligned} & f(\mathbf{x}) = \sum_{i=1}^{N_p} |x_i| + \prod_{i=1}^{N_p} |x_i| \\ & -5.12 \le x_i \le 5.11 \ \forall i \in \{1,...,N_p\} \\ & \min f(\mathbf{x}) = f(0,0,...,0) = 0 \end{aligned} \tag{7}$$

6.3 F3: Ackley

$$\begin{aligned} & f(\mathbf{x}) = -20\exp\left(-0.2\sqrt{\frac{1}{N_p}\sum_{i=1}^{N_p} x_i^2}\right) - \exp\left(\frac{1}{N_p}\sum_{i=1}^{N_p}\cos(2\pi x_i)\right) + 20 + e \\ & -5.12 \le x_i \le 5.11 \ \forall i \in \{1,...,N_p\} \\ & \min f(\mathbf{x}) = f(0,0,...,0) = 0 \end{aligned} \tag{8}$$

6.4 F4: Rastrigin

$$\begin{aligned} & f(\mathbf{x}) = \sum_{i=1}^{N_p} \left[x_i^2 - 10\cos(2\pi x_i) + 10 \right] \\ & -5.12 \le x_i \le 5.11 \ \forall i \in \{1,...,N_p\} \\ & \min f(\mathbf{x}) = f(0,0,...,0) = 0 \end{aligned} \tag{9}$$

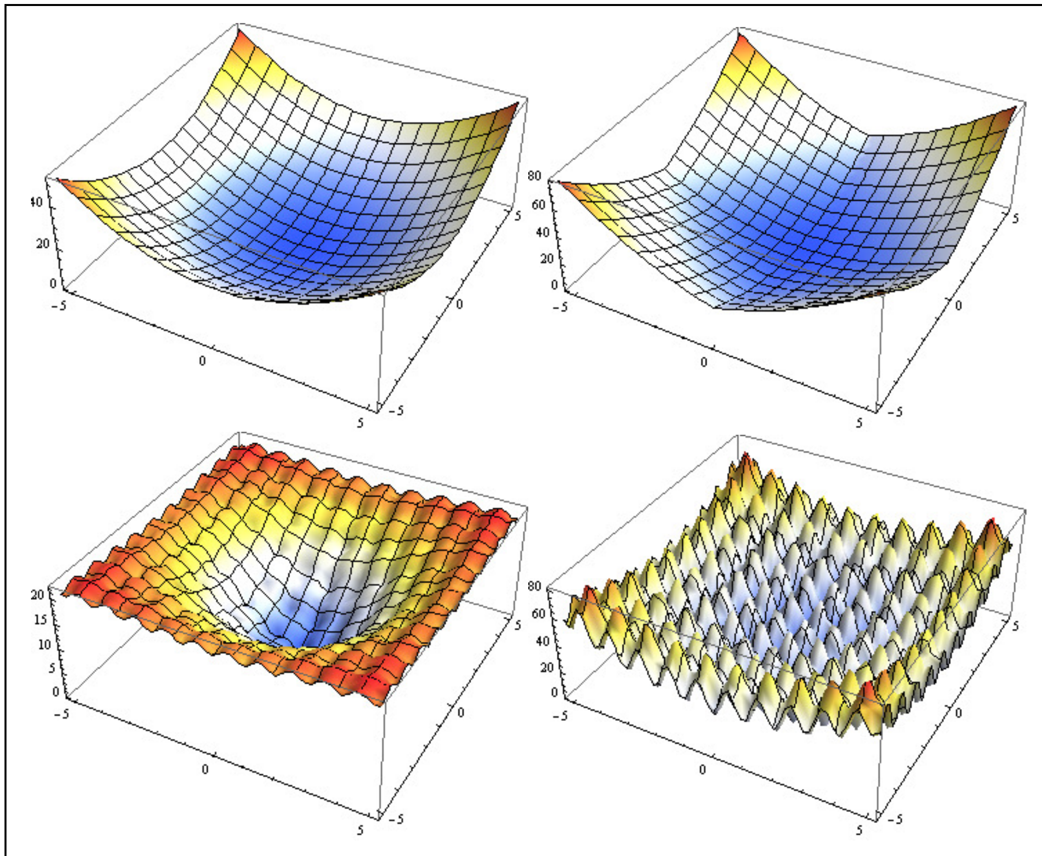

Figure 9: 2-dimensional graphs of test functions: (a) F1 Sphere (b) F2 Schwefel (c) F3 Ackley (d) F4 Rastrigin

References

[1] A.E. Eiben, J. E. Smith, “Introduction to evolutionary computing”, Springer, New York, 2003.

[2] J.H. Holland, “Adaptation in natural and artificial systems”, University of Michigan Press, Ann Arbor, MI, 1975.

[3] J.A. Vasconcelos, J.A. Ramírez, R.H.C. Takahashi, R.R. Saldanha, “Improvements in Genetic Algorithms”, IEEE Transactions on Magnetics, 37(5), 3414-3417, 2001.

[4] J. Kennedy, R.C. Eberhart, “Particle swarm optimization”, in “Proceedings IEEE International Conference on Neural Networks”, 1942-1948, 1995.

[5] Y. Jin, “A comprehensive survey of fitness approximation in evolutionary computation”, Soft Computing, 9, 3–12, 2005.

[6] J-F.M. Barthelemy, R.T. Haftka, “Approximation concepts for optimum structural design - a review”, Structural Optimization, 5, 129–144, 1993.

[7] P.Y. Papalambros, “Optimal design of mechanical engineering systems”, ASME Journal of Mechanical Design, 117, 55–62, 1995.
[8] T.W. Simpson, J.D. Peplinski, P.N. Koch, J.K. Allen, “Metamodels for computer-based engineering design: Survey and recommendations”, Engineering with Computers, 17(2), 129-150, 2001.
[9] R.E. Smith, B.A. Dike, S.A. Stegmann, “Fitness inheritance in genetic algorithms”, in “Proceedings ACM Symposium on Applied Computing”, 345-350, 1995.
[10] R. Xu, D.C. Wunsch II, “Survey of clustering algorithms”, IEEE Transactions on Neural Networks, 16(3), 645-678, 2005.
[11] V.K. Koumousis, C.P. Katsaras, “A saw-tooth Genetic Algorithm combining the effects of variable population size and reinitialization to enhance performance”, IEEE Transactions on Evolutionary Computation, 10(1), 19-28, 2006.
[12] A.E. Charalampakis, V.K. Koumousis, “Identification of Bouc-Wen hysteretic systems by a hybrid evolutionary algorithm”, Journal of Sound and Vibration, 314, 571-585, 2008.
[13] M.L. Mauldin, “Maintaining diversity in genetic search” in “Proceedings National Conference in Artificial Intelligence”, 247-250, 1984.
[14] S. Ronald, “Preventing diversity loss in a routing genetic algorithm with hash tagging”, in “Complex Systems: Mechanism of Adaption”, R. Stonier and Xing Huo Yu, (Editors), Amsterdam, IOS Press, 133-140, 1994.
[15] R.J. Povinelli, X. Feng, “Improving genetic algorithms performance by hashing fitness values”, in “Proceedings of Artificial Neural Networks in Engineering”, 399–404, 1999.
[16] J. Kratica, “Improving performances of the genetic algorithm by caching”, Computers and Artificial Intelligence, 18(3), 271–283, 1999.
[17] S.Y. Yuen, C.K. Chow, “A Genetic Algorithm that adaptively mutates and never revisits”, IEEE Transactions on Evolutionary Computation, 13(2), 454-472, 2009.
[18] C.D. Rosin, R.K. Belew, “New methods for competitive coevolution”, Evolutionary Computation, 5(1), 1-29, 1997.
[19] A. Acan, Y. Tekol, “Chromosome reuse in Genetic Algorithms”, in “Proceedings GECCO 2003”, 695–705, 2003.
[20] B. Prime, T. Hendtlass, “Mechanisms for evolutionary reincarnation” in “Proceedings ACAL 2007”, 245–256, 2007.
[21] E.H. McKinney, “Generalized birthday problem”, American Mathematical Monthly, 73(4), 385-387, 1966.
[22] D.H. Wolpert, W.G. Macready, “No free lunch theorems for optimization”, IEEE Transactions on Evolutionary Computation, 1(1), 67–82, 1997.
[23] D.L. Carroll, FORTRAN Genetic Algorithm (GA) Driver. [Online]. Available: http://www.cuaerospace.com/carroll/ga.html, 1999.