Variable-size Memory Evolutionary Algorithm:

Studies of the impact of different replacing strategies in the algorithm's performance and in the population's diversity when dealing with dynamic environments

Anabela Simões [1,2], Ernesto Costa [2]

[1]Dept. of Informatics and Systems Engineering
ISEC - Coimbra Polytechnic
Rua Pedro Nunes - Quinta da Nora
3030-199 Coimbra – Portugal

[2]Centre for Informatics and Systems of the University of Coimbra
Pólo II – Pinhal de Marrocos
3030 - 290 Coimbra – Portugal
abs@isec.pt, ernesto@dei.uc.pt

CISUC TECHNICAL REPORT TR 2007/001 - ISSN 0874-338X **FEBRUARY 2007**

Abstract. Diversity and memory are two major aspects when dealing with dynamic environments. The algorithms' adaptability to changes is usually dependent on these two issues. In this paper we investigate some improvements to a memory-based evolutionary algorithm already studied with success in dynamic optimization problems. This algorithm uses a memory and a population both with variable sizes and a biological inspired recombination operator to control the population's diversity. We propose two new replacing strategies to incorporate in the algorithm and we perform a comparative study with previous approaches. These replacing strategies allow the memory to grow in a more controlled manner, storing relevant information from the different environments. The results show that the adaptability of the algorithm improves through the time, proving that the stored information becomes useful in future situations. Combined with the conjugation operator the proposed schemes powerfully improve the effectiveness of the algorithm. We also evidenced that high diversity levels doesn't always mean better performance of Evolutionary Algorithms in dynamic environments.

Keywords: Evolutionary Algorithms, Dynamic Environments, Memory, Diversity, Replacing Strategies

1 Introduction

Evolutionary Algorithms (EAs) have been used with success in a wide area of applications. Traditionally, EAs are well suited to solve problems where the environment is static. The generational process of evolution often leads the EA to the best solution. However, most of real-world applications are dynamic and the algorithms used to solve them must be able to adapt to the new circumstances.

One intuitive and widely explored idea of helping the EAs to continuously adapt to new conditions in the environment is to use information obtained in the past. A number of authors have addressed the issue of keeping track of best individuals' information in several points of the evolutionary process and use it whenever a change occurs. This scheme is usually referred as memory and can be implicit, such as diploid representations ([8], [10]), or explicit, usually implemented with an extra population ([1], [12], [13], [16]). Other extensions introduced to classical EAs to deal with dynamic environments include maintaining population's diversity ([3], [7], [12]) or using multi-populations ([2], [15]).

In this report we are interested in studying proficient ways of using memory. We propose two different and efficient schemes of storing individuals into the memory (replacing strategies) and will test them in an EA with population and memory of variable size previously proposed and studied in [13]. Also, we will use a biologically inspired genetic operator called conjugation to control the population's diversity. This genetic operator is compared with uniform crossover and interesting conclusions about the population's diversity may be drawn.
The report is organized as follows: section 2 presents previous work using the Variable-size Memory Evolutionary Algorithm (VMEA). Section 3 describes several replacement strategies found in literature and introduces two new replacing schemes. Section 4 describes the implementation of the conjugation operator. The experimental setup is explained in section 5. In section 6 we show the obtained results concerning algorithms efficiency, growth of the memory and the population's diversity. Finally, in section 7 we state the main conclusions and ideas for future work.

2. Variable-Size Memory Evolutionary Algorithm

Simões and Costa [13] proposed an EA called VMEA – Variable-size Memory Evolutionary Algorithm, to deal with dynamic environments. This algorithm uses a memory population, responsible for storing good individuals of the evolved population in several points of the search process. The innovative aspect of the algorithm is that the two populations - search and memory – have variable sizes that can change between two boundaries: the population starts with a value of POP_MIN and the memory starts with MEM_MIN individuals. Both populations are created at random. The size of the populations can change according to the evolutionary process, but the sum of the two populations cannot go beyond a certain limit (TOTAL_MAX). The memory is updated from time to time and if the established limits are not reached, the best individual of the current population is stored in the memory. If there is no room to save this new solution, the memory is cleaned, removing individuals of equal genotype. If no individual was removed with this cleaning process, the best individual of the current population is introduced, replacing a memory individual chosen accordingly to the *replacing scheme*. In that paper, it was used an aging based replacing strategy that will be explained in the next section.
The memory is evaluated every generation and a change is detected if at least one individual in the memory changes its fitness. If a memory updating happens at generation t, then the next memory updating time is ***Tchg = t + rand(5,10)***; ***rand(5,10)*** is a random integer generated in the interval [5, 10].
If an environmental modification is detected, the best individual of the memory is introduced into the population. In the case of either the population's size or the sum of the two populations reach the allowed maximum, the best individual in memory replaces the worst one in the current population.
The pseudocode of VMEA is given in Fig. 1.

3. Replacing Strategies

The idea of adding memory to an EA has been widely used by a large number of authors. The main reason for the fact that so many algorithms explore this idea is that memory can store individuals that can be useful in the future, when previous environments reappear. Memory can be also used to maintain diversity in the population, allowing better readaptation of the EAs. Some questions can be drawn concerning memory: 1) when and which individuals to store in memory, 2) how many, 3) which should be replaced when memory is full, 4) which individuals should be selected from memory and introduced in the population when a change happens [1]. In this report we will try to answer to the third question: since the size of the memory in not unlimited, it is necessary to choose which individuals should be replaced when new ones have been selected to be stored. This process is usually called by ***replacing strategy***. Branke [1] compares a number of replacement strategies for inserting new individuals into the memory. The most popular is called by *similar* and selects the most analogous individual current in memory to be replaced, as long as the new one it is better. This replacing strategy was already used in several works ([1], [16], [17]). In future references in this text, it will be called by ***sim***.
Simões and Costa [13] proposed a replacing strategy based in the aging of memory individuals: all individuals of the memory start with age equal to zero, and every generation their age is increased by one. Besides, if they were selected to the population when a change occurs, its age is increased by a certain value and finally if a limit age is reached, their ages are reset to zero. When the memory is full and it is necessary to start replacing individuals, the youngest one is selected. The basic idea is to replace one individual with less contribution in the evolutionary

process: one that was never selected to the population or that was in the memory for a long time and so, its age became zero. This replacing strategy in future references will be called by ***age1***.
In this report we propose two different replacing strategies to use in VMEA. The first, called by ***age2***, is also based on the age of memory individuals, but their ages are computed in a different way. The age of an individual *i* is given by:

$$age_i(t) = age_i(t) + 1 + fitness_i * fit\_rate \qquad (1)$$

As suggested by [1], the individuals' age is computed every generation as a linear combination of their actual age and a contribution of its fitness (*fit_rate*).
Besides, memory individuals never die, i.e., their age is not reset to zero. This way, we do not penalize individuals that last long in memory.

```
pop_size = POP_MAX
mem_size = MEM_MIN
Initialize Memory Randomly
Initialize Population Randomly
T_M = rand(5,10)
t=0;
repeat
       Evaluate memory
       Evaluate population
       Select mating pool
       Recombination
       Mutation
       if  is time to update memory  then
            T_M = t + rand(5,10)
            Select best individual of the population
            if there is room to one more individual in memory then
              Store best individual in memory
              Increase mem_size
            else
              if cleanMemory is successful then
                    Update mem_size
                    Store best individual in memory
              else
                     Replace individual of memory according the replacing scheme
       if  change is detected  then
            Select best individual from memory
            if there is room to one more individual in population then
                    Store individual in population
                    Increase pop_size
            else
                    if cleanMemory is successful then
                        Update mem_size
                        Store individual in population
                   else
                     Replace worst individual of population
until stop_condition
```

Fig. 1 - Pseudo-code of the Variable Memory Evolutionary Algorithm (VMEA)

When its time to update memory, if there is room for the best individual of current population, we add it to memory, increasing its size. If memory is full, we will replace the youngest individual, considering that age was calculated using equation 1.
The second replacing strategy, denominated by ***generational***, selects the worst individual present in the memory **since the last change**. For instance, if last change occurred at generation *t1* and currently the algorithm is in generation *t2*, when it is time to insert an individual into the memory we will replace the worst individual that was stored between generation *t1+1* and *t2-1*.
When its time to update memory, if no individual has been stored since last change and the memory is not full, we add the best individual of the population to the memory. In the case of the maximum size of the memory has been reached, we use the 'most similar' strategy and substitute the closest individual in terms of Hamming distance, if it is worse than the current best. This way, the maximum size of the memory is reached slower, the redundant information is minimized and the best information of a certain period of the evolutionary process is stored. This strategy will be called by ***gen***.

4. Bacterial Conjugation

It is widely accepted in the Evolutionary Computation community that genetic operators are essential to the efficiency of EAs. They allow creating new individuals, using the genetic information of previous ones, allowing the algorithm to converge to the desired solution. Traditionally crossover is used as the main genetic operator. Nevertheless, other biologically inspired operators have been proposed and tested with some degree of success. These new genetic operators were applied either in static ([5], [11]), or dynamic environments ([12], [17]).
In biology, bacterial conjugation is the transfer of genetic material between bacteria through cell-to-cell contact. Sometimes bacterial conjugation is regarded as the bacterial equivalent of sexual reproduction or mating, but in fact, it is merely the transfer of genetic information from a donor to a recipient cell [9].
Computational conjugation was introduced independently by Harvey and Smith. Smith [14] proposed an implementation of this operator, called simple conjugation: the donor and the recipient were chosen randomly, transferring the genetic material between two random points. Harvey [5] introduced a tournament based conjugation: two parents are selected on a random basis, and then the winner of the tournament becomes the donor and the loser the recipient of the genetic material. That way, the conjugation operator can be applied repeatedly by different donors to a single recipient.
Simões and Costa [13] used conjugation in the context of dynamic environments. The authors applied conjugation after selecting the individuals to the mating pool, using the idea of donor-recipient genetic transfer. As it happens in biology, the donor individuals give genetic material to the recipient ones. After selecting the individuals to mate, using the established selection method, they are divided into two groups: the n/2 best individuals become the 'donor', the remaining become the 'recipient' (n is the current size of the population). Then, the i^{th} donor transfers part of its genetic material to the i^{th} recipient (i=1, …n/2). This injection is controlled by two points randomly chosen. The donor remains unchanged. Following that, all offspring created by this process are joined with the donor individuals and they become the next population of size n. Fig. 2 shows how conjugation is applied to one pair of individuals of the mating pool.

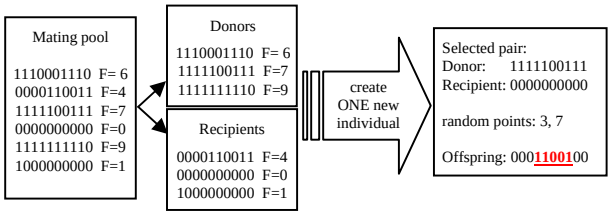


Fig. 2 - Creating a new individual using conjugation

The results obtained using this operator in an EA to solve dynamic optimization problems were quite promising [13].

5. Experimental Design

5.1. Dynamic Test Environments

The dynamic environments used to test our approaches were constructed using Yang's Dynamic Optimization Problems (DOP) generator. With this generator it is possible to construct different dynamic environments from any binary-encoded stationary function using the bitwise exclusive-or (XOR) operator. The basic idea of the generator can be described as follows: when evaluating an individual x in the population, first we perform the operation $x \oplus M$ where $\oplus$ is the bitwise XOR operator and M a binary mask previously generated. Then, the resulting individual is evaluated to obtain its fitness value. If a change happens at generation t, then we have $f(x, t+1) = f(x \oplus M, t)$. Using the DOP generator the characteristics of the change are controlled by two parameters: the speed of the change, r, which is the number of generations between two changes, and the magnitude of the change, ρ, that consists in the ratio of ones in the mask $\boldsymbol{M}$. The more ones in the mask the more severe is the change. The DOP generator also allows constructing problems where the changes can be cyclic, cyclic with noise or non-cyclic. In the first case, several masks are generated according to the ρ parameter and are consecutively applied when a change occurs. It is thus possible that previous environments reappear later. In the second case noise is added by mutating some bits in the mask with a small probability. In the third case, the mask applied to the individuals is always randomly generated every time we change the environment. More details about Yang's DOP generator can be found in [18].

In this paper we constructed 20 cyclic DOP, setting the parameter r to 10, 50, 100 and 200. The ratio ρ was set to different values in order to test different levels of change: 0.1 (a light shifting) 0.2, 0.5, 1.0 (severe change). In order to study the behavior of the algorithm in randomly changing environments we also set ρ to a uniformly randomly generated value in the interval [0.01 and 0.99] (called by ***rnd***).

To test and compare the several replacing schemes combined with two different genetic operators, we selected two benchmark problems: the knapsack problem (100 items) and OneMax problem (300 bits).

Knapsack Problem

The knapsack problem is a NP-complete combinatorial optimization problem often used as benchmark. It consists in selecting a number of items to a knapsack with limited capacity. Each item has a value (v_i) and a weight (w_i) and the objective is to choose the items that maximize the total value, without exceeding the capacity of the bag. We used a knapsack problem with 100 items using strongly correlated sets of randomly generated data [6]. The fitness of an individual is equal to the sum of the values of the selected items, if the weight limit is not reached. If too many items are selected, then the fitness is penalized in order to ensure that invalid individuals are distinguished from the valid ones.

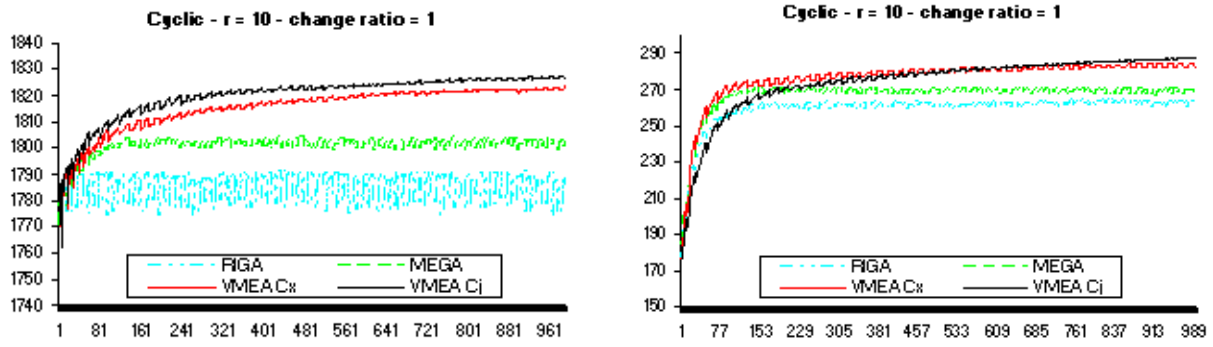
OneMax problem

The OneMax problem aims to maximize the number of ones in a binary string. So, the fitness of an individual consists in the number of ones present in the binary string. This problem has a unique solution. In our experiments we used individuals of length 300.

5.2. Experimental Setup

Algorithms' parameters

Previous work compared VMEA using the *age1* replacing strategy with other algorithms: the random immigrants' algorithm [4] and the memory-enhanced GA (MEGA) studied in [16]. The results proved that VMEA, using either crossover (VMEA Cx) or conjugation (VMEA Cj), generally outperformed the other approaches. A typical comparative graphic of the behavior of VMEA and the other studied algorithms is shown in Fig. 3. To see more results consult [13].



(a) (b)

Fig. 3 - VMEA versus MEGA and RIGA in a) Knapsack and b) OneMax

In this study we will be focused in the performance of VMEA and we will compare its efficiency using several replacing strategies: similar (*sim*), proposed by Branke in [1], *age1*, already used in previous work [13] and the two new ones proposed and explained before in this paper (*age2* and *gen*). For each case, VMEA was tested using conjugation (VMEA-Cj) and uniform crossover (VMEA-Cx). This way we could make some conclusions about the efficiency of this genetic operator in changing problems and study its impact in the population's diversity. The algorithms parameters were set as follows: generational replacement with elitism of size one, tournament selection with tournament of size two, uniform crossover with probability p_c=0.7 (the same probability was used with conjugation) and mutation applied with probability p_m=0.01. The search population starts with 100 individuals, the memory starts with 10 individuals, both created randomly. Their sizes are variable, changing through the time, but the sum of the two populations can not surpass 120. The *age1* strategy was implemented the same way as described in [13]. For the strategy *age2*, computed according equation (1), we set *fit_rate* with 0.1 (value chosen after some preliminary experimentation).

For each experiment of an algorithm, 30 runs were executed and the number of environmental changes was 200 with *r* =10 (2000 generations), 80 with *r* =50 (4000 generations) and 40 with *r*=100 and 200 (4000 and 8000 generations, respectively). The overall performance used to compare the algorithms was the best-of-generation fitness averaged over 30 independent runs, executed with the same random seeds:

$$Foverall = \frac{1}{G}\sum_{i=1}^{G}\left[\frac{1}{N}\sum_{j=1}^{N} F_{best\,ij}\right] \qquad (2)$$

G=number of generations, N=number of runs.

6. Experimental Results

The statistical validation of the results was made using paired one-tailed t-test at a 0.01 level of significance. Each line of the tables 1 and 2 compares a pair of replacement schemes and the used notation is '+", "-", "++" or "--", when the first strategy is better than, worse than, significantly better than, or significantly worse than the second one. In the next sections we show the results concerning the efficiency of the algorithm using the different replacing strategies, the diversity levels measured in population and memory and finally, the observed results in the growth of the population and memory sizes using the different schemes.

6.1. Comparing Replacing Strategies

The global results are plotted in Fig. 4 and 5. Analyzing the results on dynamic knapsack, through the values of Table 2 and Fig. 4, we can see that the proposed *gen* replacing scheme obtained the best results always, except when r=10. This happens because the change period is small and, when a change occurs, most of the times, no individual of that period was already stored. So, in this case, when memory is full, we replace the most similar. Therefore, the observations made when r=50, 100 and 200 didn't happen for r=10. In this case, the best replacing strategy was the

proposed *age2*. The *age2* scheme compared with the remaining was not so effective. In fact, the results in Table 1 show that, *age2* performs better in small change periods and combined with conjugation. These results confirm the difficulty of finding a trade-off between the fitness and age contributions, as suggested in [1].
On dynamic OneMax, the *gen* replacing strategy outperformed the remaining schemes on most DOPs. Once again the weak performance was observed when r=10. Table 2 and Fig. 5 show the global results on dynamic OneMax.

age2 vs others		Knaspsack					OneMax				
T-test results	*r*↓	0.1	0.	0.	1	rnd	0.	0.	0.	1	rnd
age2 Cx -- age1		++	++	++	++	++	++	+	+	+	++
age2 Cx -- age1		--	--	-	-	--	++	++	--	--	++
age2 Cx -- sim		++	++	++	++	++	++	+	+	+	++
age2 Cx -- sim		--	++	--	-	--	++	++	+	+	++
age2 Cx -- gen		+	++	+	++	-	-	-	+	+	-
age2 Cx -- gen		--	-	--	-	--	++	++	--	--	++
age2 Cj -- age1	10	++	++	++	++	++	--	-	++	++	--
age2 Cj -- age1		++	++	++	++	++	++	+	-	-	++
age2 Cj -- sim		++	++	++	++	++	--	-	++	++	--
age2 Cj -- sim Cj		++	++	++	++	++	++	++	++	++	+
age2 Cj -- gen		++	++	++	++	++	--	--	++	++	--
age2 Cj -- gen Cj		++	++	++	++	++	+	+	-	-	-
age2 Cj - age2		++	++	++	++	++	--	--	++	++	--
age2 Cx -- age1		--	+	++	+	--	--	++	+	-	--
age2 Cx -- age1		--	--	--	--	--	--	--	--	--	--
age2 Cx -- sim		--	--	-	+	--	--	--	+	-	--
age2 Cx -- sim		--	--	--	--	--	--	++	--	--	--
age2 Cx -- gen		--	--	--	--	--	--	--	--	--	--
age2 Cx -- gen		--	--	--	--	--	--	--	--	--	--
age2 Cj -- age1	50	++	++	++	++	++	++	++	++	++	++
age2 Cj -- age1		--	++	++	-	--	++	+	-	--	++
age2 Cj -- sim		++	++	++	++	+	--	++	++	++	--
age2 Cj -- sim Cj		--	--	-	+	--	--	++	+	--	--
age2 Cj -- gen		--	--	--	--	--	--	--	--	--	--
age2 Cj -- gen Cj		--	--	--	--	--	--	++	+	-	--
age2 Cj - age2		++	++	++	++	++	++	++	++	++	++
age2 Cx -- age1		--	--	++	+	-	--	--	-	+	--
age2 Cx -- age1		--	--	--	--	--	--	--	--	--	--
age2 Cx -- sim		--	--	-	+	--	--	--	--	-	--
age2 Cx -- sim		--	--	--	--	--	--	--	--	--	--
age2 Cx -- gen		--	--	--	--	--	--	--	--	--	--
age2 Cx -- gen		--	--	--	--	--	--	--	--	--	--
age2 Cj -- age1	100	++	++	++	++	++	++	++	++	++	++
age2 Cj -- age1		--	+	+	-	--	--	++	--	--	--
age2 Cj -- sim		++	++	++	++	++	+	++	++	++	+
age2 Cj -- sim Cj		--	--	--	-	--	--	--	--	--	--
age2 Cj -- gen		--	--	--	--	--	--	--	--	--	--
age2 Cj -- gen Cj		--	--	--	--	--	--	--	--	-	--
age2 Cj - age2		++	++	++	++	++	++	++	++	++	++
age2 Cx -- age1		-	-	+	-	+	-	+	++	-	-
age2 Cx -- age1		--	--	--	--	--	--	--	--	--	--
age2 Cx -- sim		--	--	--	--	--	--	--	--	--	--
age2 Cx -- sim		--	--	--	--	--	--	--	--	--	--
age2 Cx -- gen		--	--	--	--	--	--	--	--	--	--
age2 Cx -- gen		--	--	--	--	--	--	--	--	--	--
age2 Cj -- age1	200	++	++	++	++	++	++	++	++	++	++
age2 Cj -- age1		-	-	+	-	-	+	+	+	-	-
age2 Cj -- sim		++	++	++	++	++	++	+	++	++	++
age2 Cj -- sim Cj		++	++	++	++	++	--	--	++	++	--
age2 Cj -- gen		--	--	--	--	--	--	--	--	--	--
age2 Cj -- gen Cj		--	--	--	--	--	--	--	--	--	--
age2 Cj - age2		++	++	++	++	++	++	++	++	++	++

Table 1. The t-test results of comparing the *age2* scheme with all the other

gen vs others		KP					OneMax				
T-test results	*r↓ ρ→*	0.1	0.	0.	1	rnd	0.	0.	0.	1	rnd
gen Cx - ag1		++	-	+	+	++	++	++	-	+	++
gen Cx - age1		--	--	--	--	--	++	++	--	--	++
gen Cx - age2		-	--	-	--	+	+	+	-	-	+
gen Cx - age2		--	--	--	--	--	++	++	--	--	++
gen Cx - sim		++	++	+	++	++	++	++	+	+	++
gen Cx - sim Cj		--	-	--	--	--	++	++	+	+	++
gen Cj - ag1 Cx	10	++	++	++	++	++	--	--	++	++	--
gen Cj - age1		+	-	+	-	+	++	+	-	+	++
gen Cj - age2		++	+	++	+	++	--	--	++	++	--
gen Cj - age2		--	--	--	--	--	-	-	+	+	+
gen Cj - sim Cx		++	++	++	++	++	--	--	++	++	--
gen Cj - sim Cj		++	++	-	-	+	++	++	++	++	++
gen Cj - gen Cx		++	++	++	++	++	--	--	++	++	--
gen Cx - ag1		++	++	++	++	++	++	++	++	++	++
gen Cx - age1		++	++	++	++	++	++	++	++	++	++
gen Cx - age2		++	++	++	++	++	++	++	++	++	++
gen Cx - age2		++	++	++	++	++	++	++	++	++	++
gen Cx - sim		++	++	++	++	++	++	++	++	++	++
gen Cx - sim Cj		++	++	++	++	++	++	++	++	++	++
gen Cj - ag1 Cx	50	++	++	++	++	++	++	++	++	++	++
gen Cj - age1		++	++	++	++	++	++	-	-	-	++
gen Cj - age2		++	++	++	++	++	++	++	++	++	++
gen Cj - age2		++	++	++	++	++	++	--	-	+	++
gen Cj - sim Cx		++	++	++	++	++	++	++	++	++	++
gen Cj - sim Cj		++	++	++	++	++	++	++	+	-	++
gen Cj - gen Cx		++	++	++	++	++	--	--	--	--	--
gen Cx - ag1		++	++	++	++	++	++	++	++	++	++
gen Cx - age1		++	++	++	++	++	++	++	++	++	++
gen Cx - age2		++	++	++	++	++	++	++	++	++	++
gen Cx - age2		++	++	++	++	++	++	++	++	++	++
gen Cx - sim		++	++	++	++	++	++	++	++	++	++
gen Cx - sim Cj		++	++	++	++	++	++	++	++	++	++
gen Cj - ag1 Cx	100	++	++	++	++	++	++	++	++	++	++
gen Cj - age1		++	++	++	++	++	++	++	++	--	++
gen Cj - age2		++	++	++	++	++	++	++	++	++	++
gen Cj - age2		++	++	++	++	++	++	++	++	+	++
gen Cj - sim Cx		++	++	++	++	++	++	++	++	++	++
gen Cj - sim Cj		++	++	++	++	++	++	++	++	--	++
gen Cj - gen Cx		++	++	++	++	++	-	++	++	--	+
gen Cx - ag1		++	++	++	++	++	++	++	++	++	++
gen Cx - age1		++	++	++	++	++	++	++	++	++	++
gen Cx - age2		++	++	++	++	++	++	++	++	++	++
gen Cx - age2		++	++	++	++	++	++	++	++	++	++
gen Cx - sim		++	++	++	++	++	++	++	++	++	++
gen Cx - sim Cj		++	++	++	++	++	++	++	++	++	++
gen Cj - ag1 Cx	200	++	++	++	++	++	++	++	++	++	++
gen Cj - age1		++	++	++	++	++	++	++	++	++	++
gen Cj - age2		++	++	++	++	++	++	++	++	++	++
gen Cj - age2		++	++	++	++	++	++	++	++	++	++
gen Cj - sim Cx		++	++	++	++	++	++	++	++	++	++
gen Cj - sim Cj		++	++	++	++	++	++	++	++	++	++
gen Cj - gen Cx		++	++	++	++	++	++	++	+	-	++

Table 2. The t-test results of comparing the *gen* scheme with all the other

Observing Fig. 6 to 15 we confirm that using the proposed *gen* scheme the VMEA obtained excellent performances. The information stored in memory allows a continuously adaptation of the algorithm, contrary to the remaining replacing strategies. Another observed fact is that conjugation, combined with any replacing strategy, is in general superior to uniform crossover, just as observed in previous results published in [13].

Even with random change ratio, the algorithm implemented with the *gen* scheme was able to readapt quickly after a change. This behaviour indicates that the information stored in memory is useful in future situations and in general the important information about the past is not destroyed when it's necessary start replacing individuals.

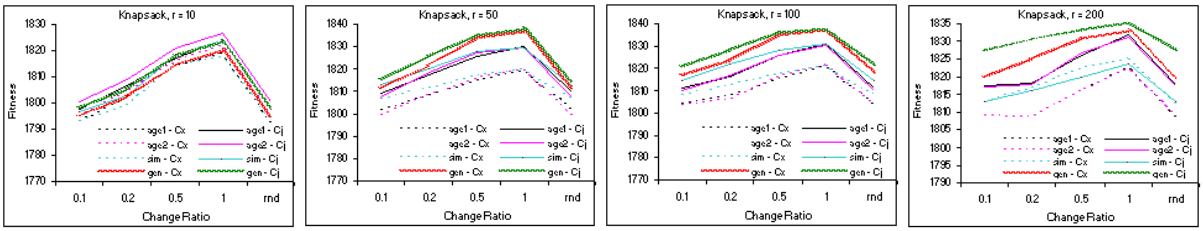


Fig. 4 - Results on dynamic Knapsack problem, with different change ratios (ρ) and r=10, 50, 100 and 200

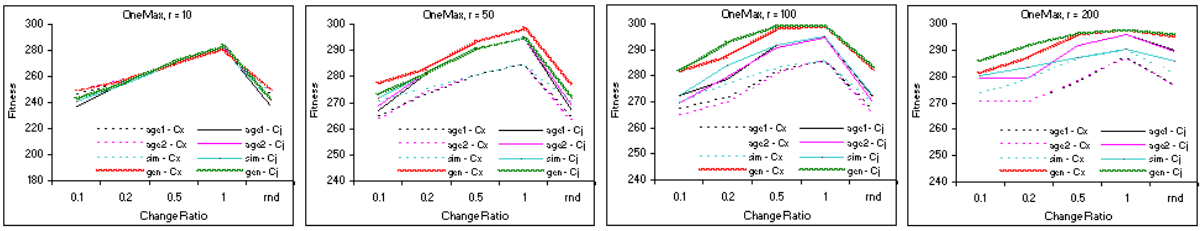


Fig. 5 - Results on dynamic OneMax problem, with different change ratios (ρ) and r=10, 50, 100 and 200

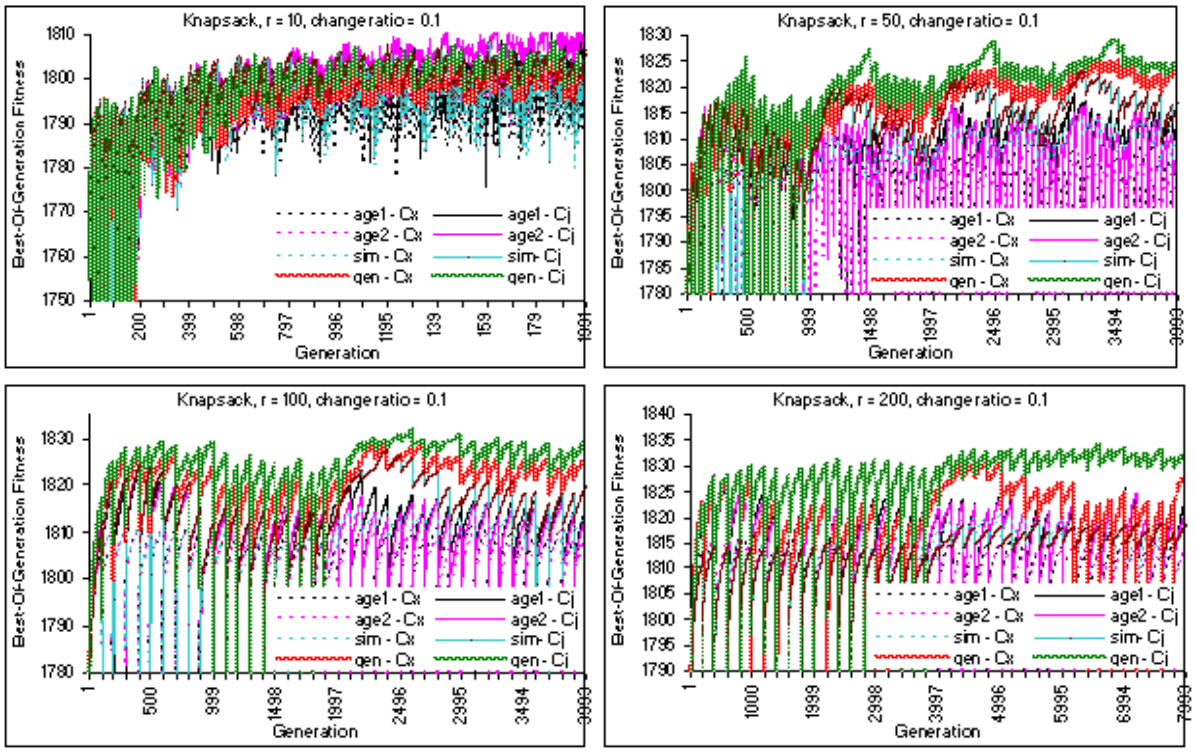


Fig. 6 – Dynamic behavior of VMEAs on dynamic Knapsack, with r=10, 50, 100 and 200, when ρ = 0.1

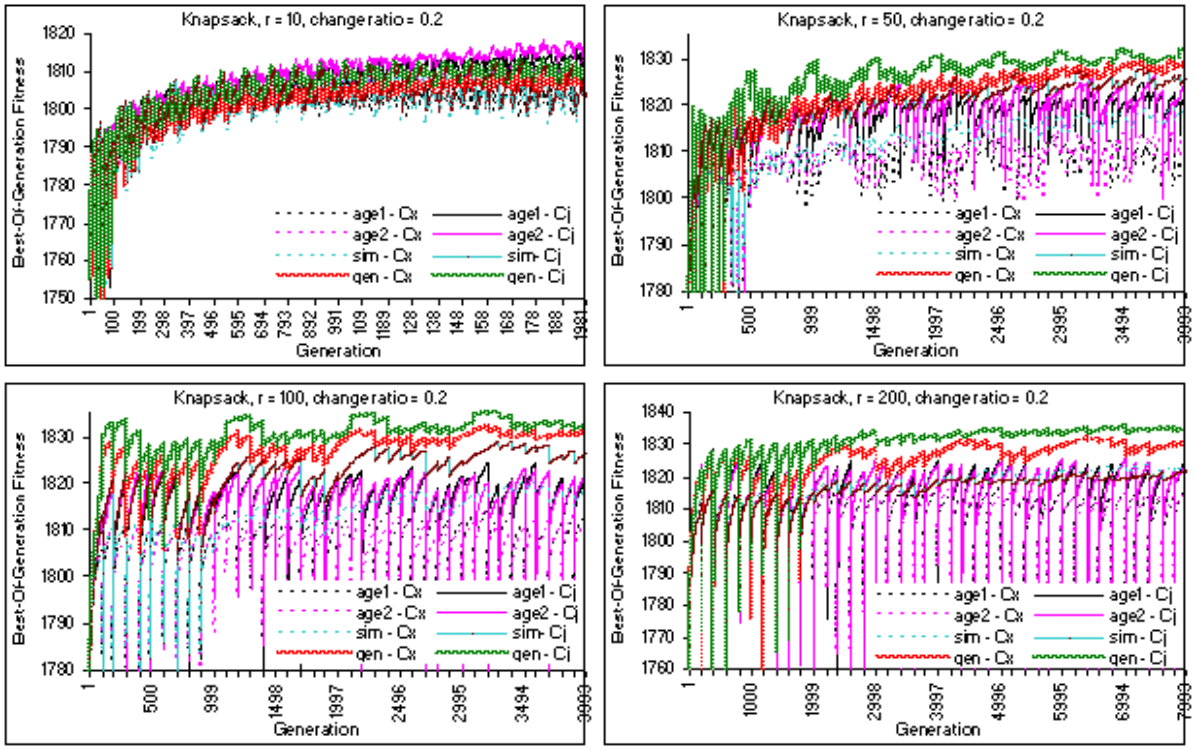


Fig. 7 - Dynamic behavior of VMEAs on dynamic Knapsack, with r=10, 50, 100 and 200, when ρ = 0.2

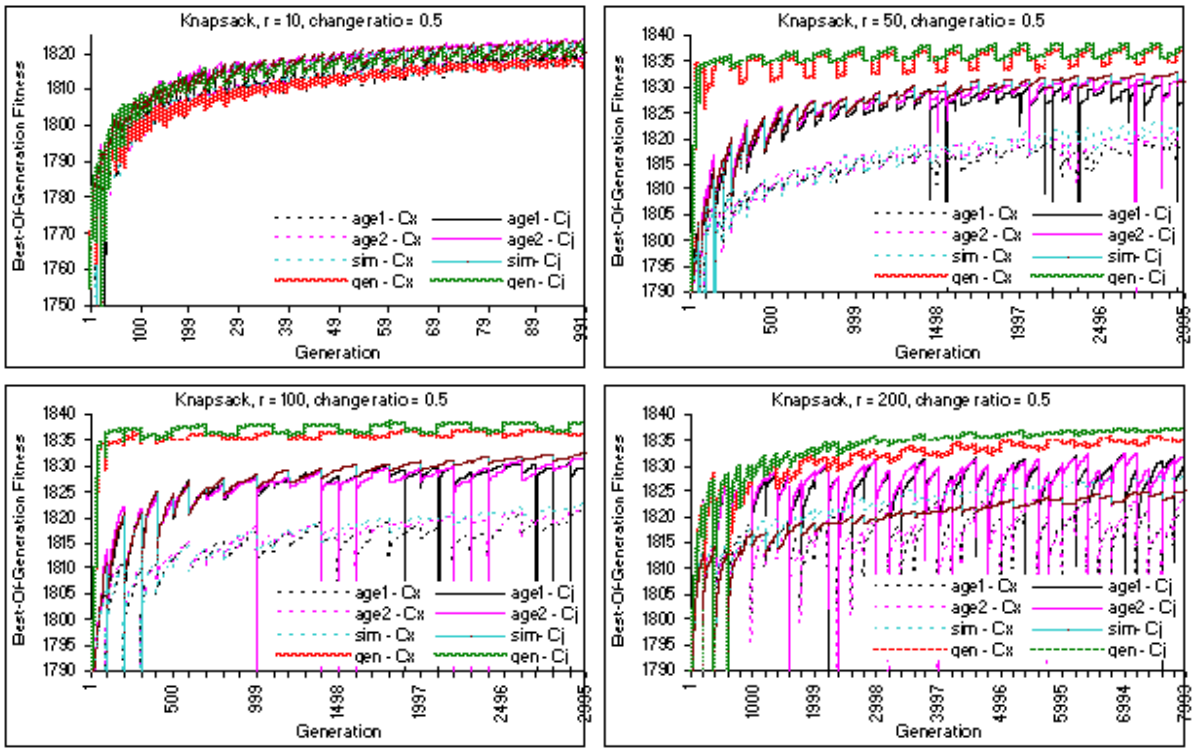


Fig. 8 - Dynamic behavior of VMEAs on dynamic Knapsack, with r=10, 50, 100 and 200, when ρ = 0.5

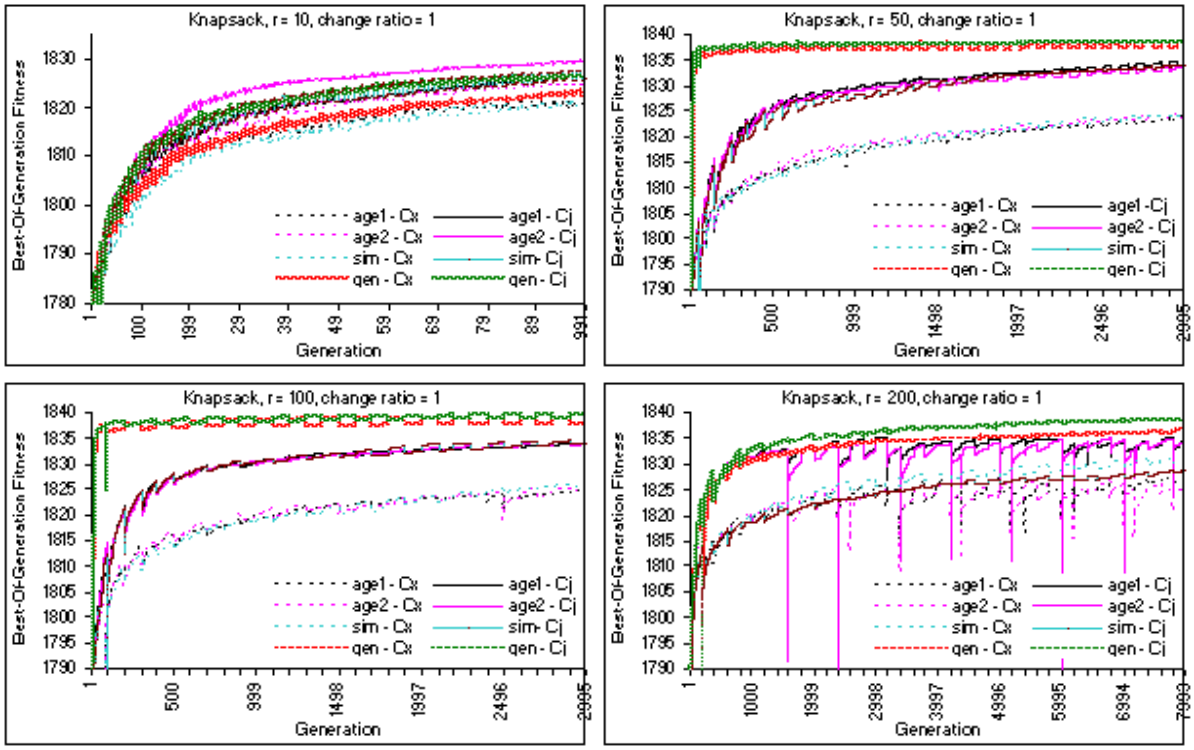


Fig. 9 - Dynamic behavior of VMEAs on dynamic Knapsack, with r=10, 50, 100 and 200, when ρ = 1.0

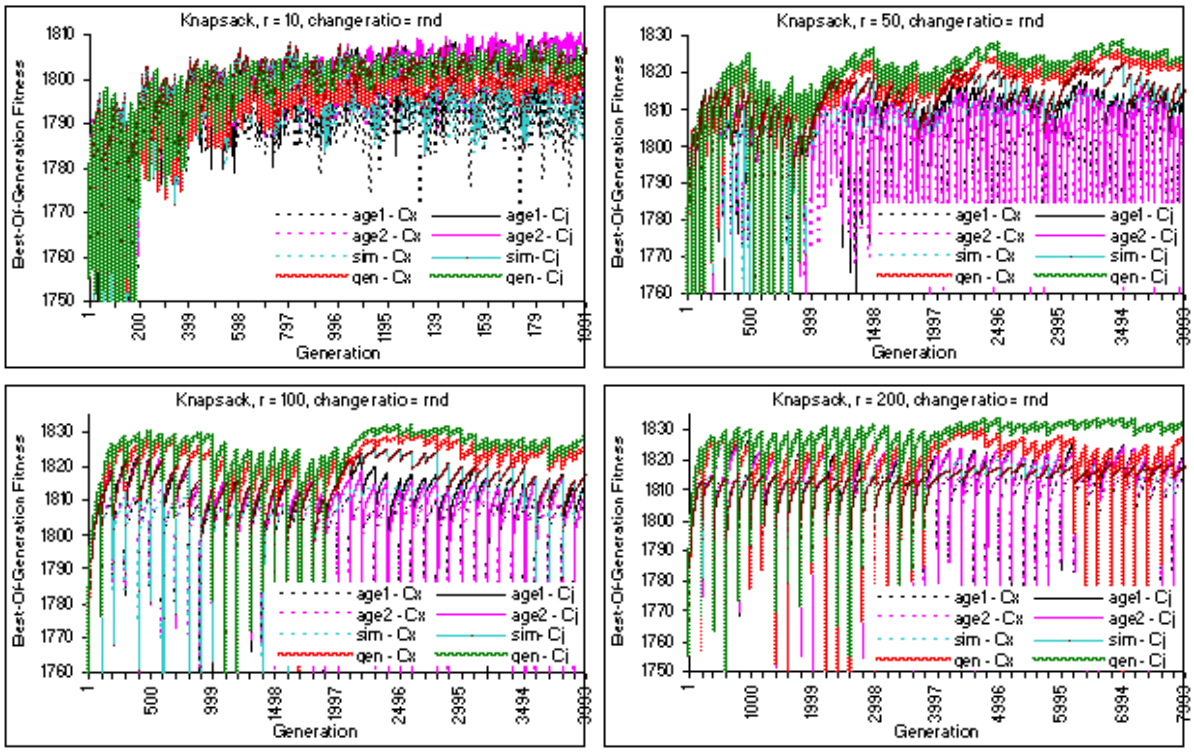


Fig. 10 - Dynamic behavior of VMEAs on dynamic Knapsack, with r=10, 50, 100 and 200, when ρ = rnd

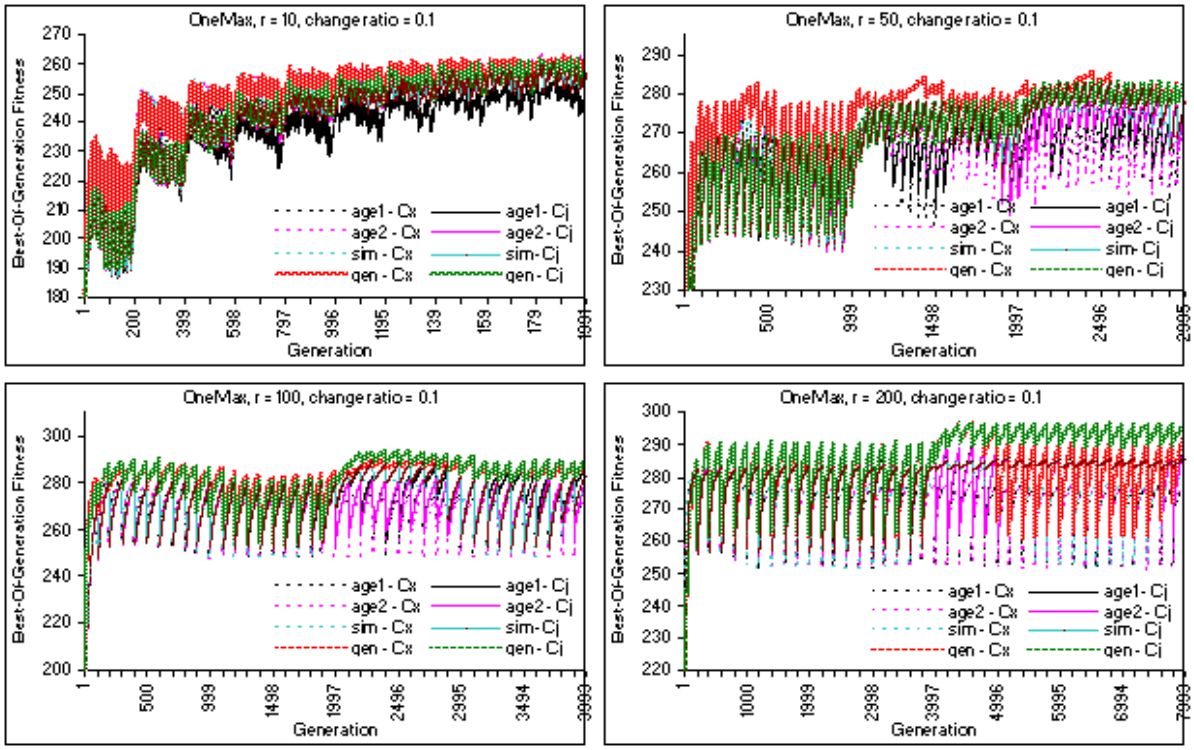


Fig. 11 - Dynamic behavior of VMEAs on dynamic OneMax, with r=10, 50, 100 and 200, when ρ = 0.1

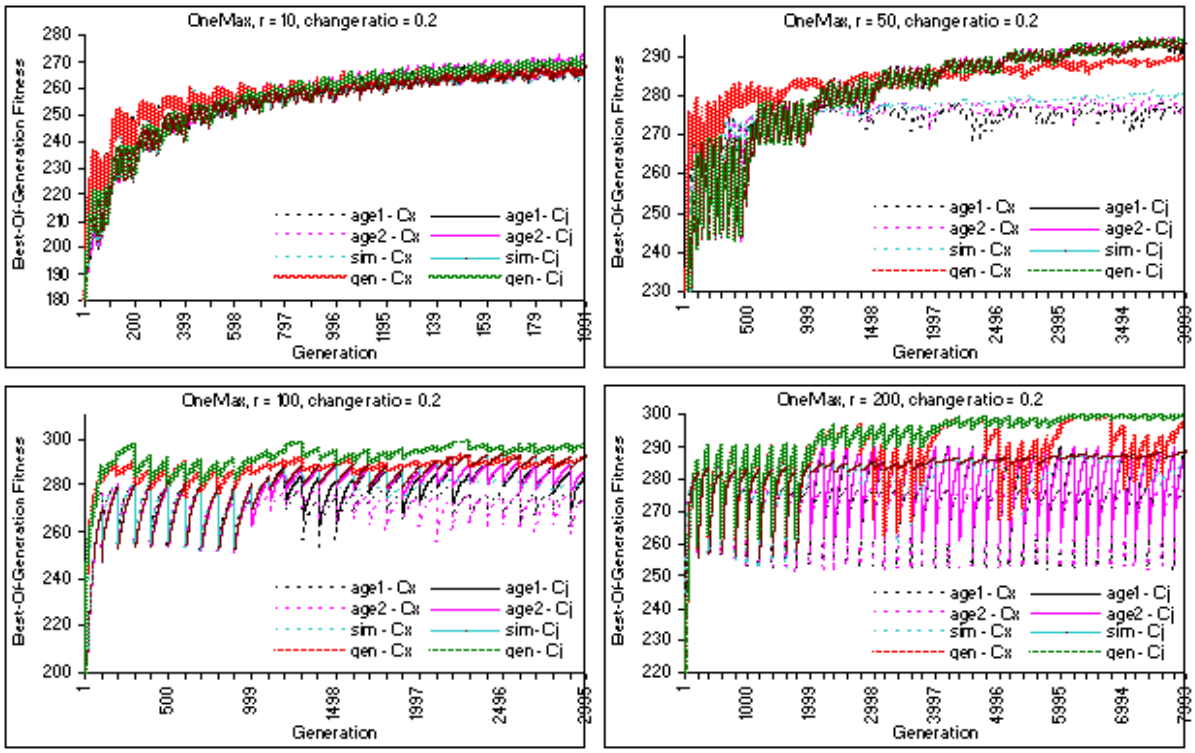


Fig. 12 - Dynamic behavior of VMEAs on dynamic OneMax, with r=10, 50, 100 and 200, when ρ = 0.2

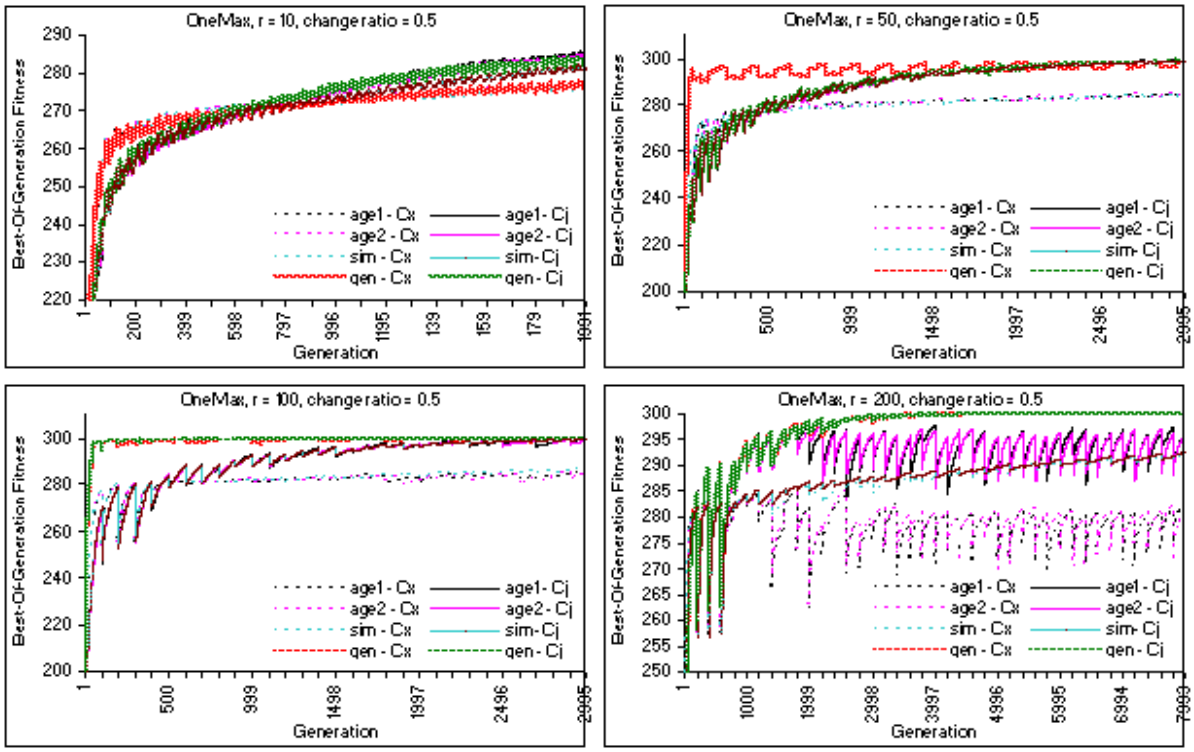


Fig. 13 - Dynamic behavior of VMEAs on dynamic OneMax, with r=10, 50, 100 and 200, when ρ = 0.5

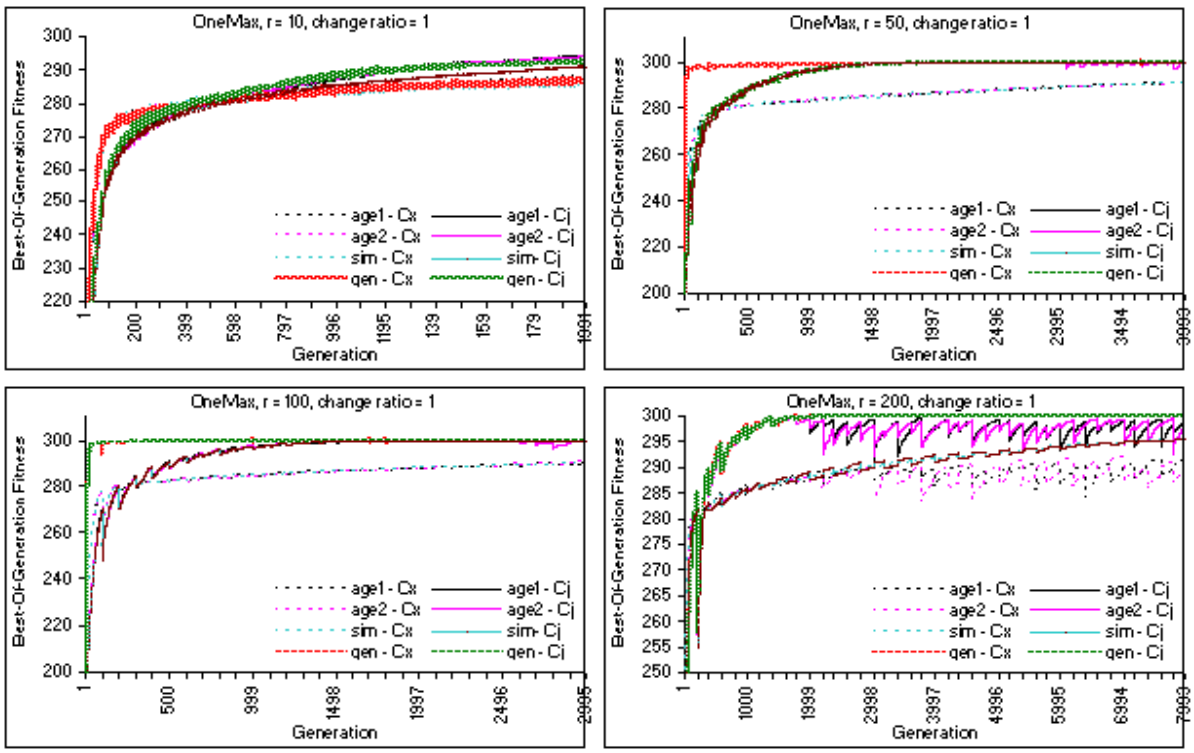


Fig. 14 - Dynamic behavior of VMEAs on dynamic OneMax, with r=10, 50, 100 and 200, when ρ = 1.0

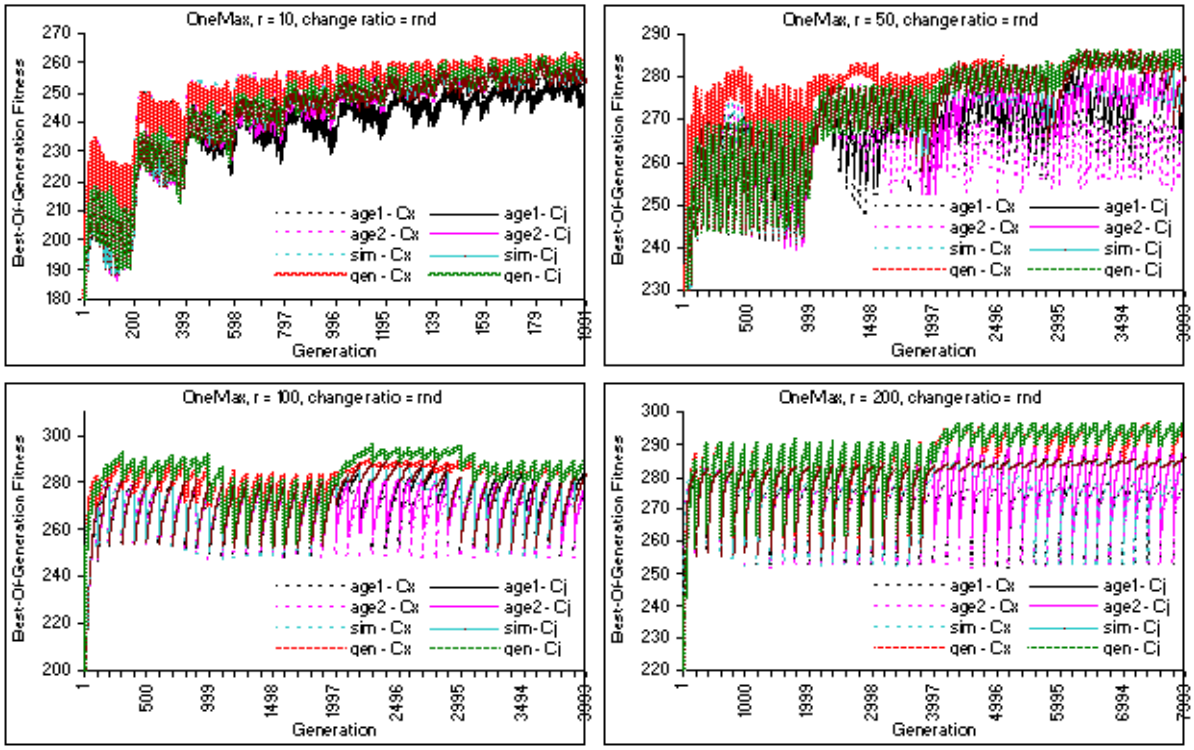


Fig. 15 - Dynamic behavior of VMEAs on dynamic OneMax, with r=10, 50, 100 and 200, when ρ = rnd

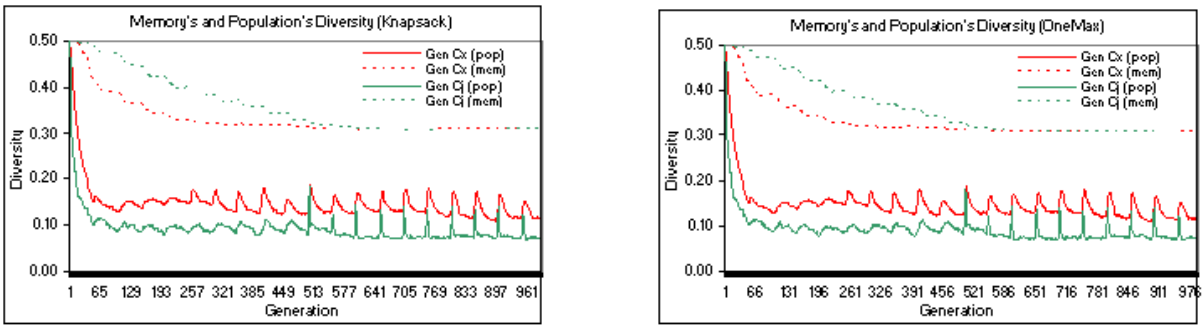
6.2. Population's and Memory's Diversity

In order to study the impact of the genetic operators in the diversity dynamics, we stored the population's and memory's diversity at every generation using the standard measure based on the Hamming distance. At generation t, the diversity is given by:

$$Div(t) = \frac{1}{N}\sum_{k=1}^{N}\left[\frac{1}{l.n(l-1)}\sum_{i=1}^{n}\sum_{j\neq i}^{n} HDij(k,t)\right] \qquad (3)$$

where N is the number of runs, l is the length of the chromosomes, n is the population (or memory) size at generation t and $HD_{ij}(k,t)$ is the Hamming distance between individuals i and j at generation t, in the k^{th} run.

Analyzing the recorded values for population's and memory's diversity, we conclude that the conjugation operator always kept lower diversity levels in the population. Nevertheless, commonly, the best results are achieved using conjugation operator! As an example, we show the diversity of memory and population stored on the studied benchmarks, with r=50 and ρ=0.2, using the *gen* replacing scheme. As we saw in Fig. 6 to 15, the algorithm using the *gen* scheme and conjugation obtained the best results. From Fig. 16, it follows that the diversity level of the population using conjugation in this situation was inferior. In the memory, the opposite happens: the proposed replacing method keeps more diversity in the memory during the first environmental changes. After that, the memory's diversity remains constant because different individuals keep being introduced. This is also true for the remaining strategies and for different values of r and ρ.



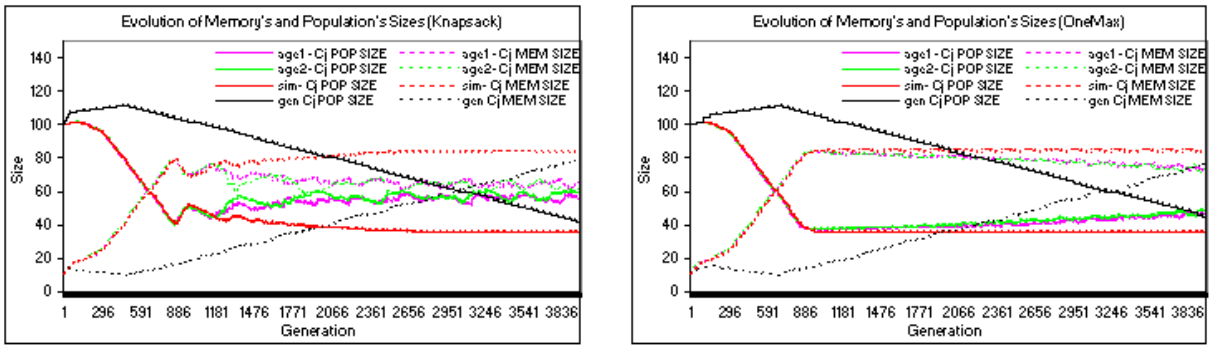
(a) (b)

Fig. 16 - Population's and Memory's diversity in the first 20 environmental changes in a) dynamic Knapsack and b) dynamic Onemax

These results show that the usual idea that in dynamic environments it is crucial to maintain high diversity in the population, in order to improve the adaptability of algorithm, is not always true. One reason to this fact maybe because conjugation is less disruptive, since the information of the best individuals is always preserved, and that appear to be positive in the studied problems. We are making more experimentation in order to provide more consistent conclusions.

6.3. Memory and Populations Sizes

The results concerning the growth of memory's and population's sizes validate our assumption that using the *gen* replacing scheme the memory should grow slower. In fact, this was observed in all studied DOPs. Fig. 17 shows a typical example of how population and memory grow using the different replacing schemes. We can see that, with *gen* replacing strategy (black lines), the memory's size increased very slowly and its maximum is only reached at the end of the evolutionary process.



(a) (b)

Fig. 17 - Population's and Memory's sizes on a) dynamic Knapsack and b) dynamic OneMax

By storing individuals in memory, replacing the worst one that was memorized in the same change period, we save space to the good individuals that will appear in future generations, without having the need of replacing useful information.

7. Conclusions

This report exhaustively studied two new replacing strategies and tested its efficiency in the Variable-size Memory Evolutionary Algorithm. A genetic operator called conjugation was used combined with the proposed schemes. An

experimental studied was carried out using different dynamic optimization problems based in two benchmark problems. From the obtained results several conclusions can be drawn: First, the *age2* replacing scheme obtained better results in dynamics with small period changes if combined with conjugation. For the remaining cases, the other replacing schemes were usually better. Second, analyzing the results inside the same method (*age1-Cx vs age1-Cj, age2-Cx vs age2-Cj, sim-Cx, vs sim-cj and gen-Cx vs gen-Cj*), conjugation typically outperformed uniform crossover (few exceptions were found). Third, the proposed gen replacing scheme provided excellent improvement in the algorithm's performance. Few exceptions were observed for some DOPs with small change periods. These results prove that, when dealing with dynamic environments, the information stored in memory is crucial to the effectiveness of the algorithms. Since the memory's capacity usually has limited size, this approach of replacing the individuals when memory is full proved to be an excellent choice.

8. References

[1] J. Branke. Memory Enhanced Evolutionary Algorithms for Changing Optimization Problems. Proc. of Congress on Evol. Computation 1999, pp. 1875-1882, IEEE, 1999.

[2] J. Branke, T. Kaußler, C. Schmidt and H. Schmeck. A Multi-Population Approach to Dynamic Optimization Problems. Adaptive Computing in Design and Manufacture (ACDM 2000), pp. 299-308, Springer, 2000.

[3] H. Cobb. An Investigation into the Use of Hypermutation as an Adaptive Operator in Genetic Algorithms Having Continuous, Time-Dependent Non-Stationary Environments. Technical Report AIC-90-001, 1990.

[4] J. J. Grefenstette. Genetic Algorithms for Changing Environments. Proc. of the 2nd Int. Conf. Parallel Problem Solving from Nature 2, pp. 137-144, North-Holland, 1992.

[5] I. Harvey. The Microbial Genetic Algorithm. Unpublished, 1996.

[6] Z. Michalewicz. Genetic Algorithms + Data Structures = Evolution Programs. 3rd Edition Springer-Verlag, 1999.

[7] R. W. Morrison and K. De Jong. Triggered Hypermutation Revisited. Proc. of Congress on Evol. Computation 1999, pp. 1025-1032, IEEE, 1999.

[8] K. P. Ng and K. C. Wong. A New Diploid Scheme and Dominance Change Mechanism for Non-stationary Function Optimization. Proc. of the 6th Int. Conf. on Genetic Algorithms, pp. 159-166, Morgan Kaufmann, 1995.

[9] P. J. Russell. Genetics. 5th edition, Addison-Wesley, 1998.

[10] A. Sima Uyar and A. Emre Harmanci. A New Population Based Adaptive Domination Change Mechanism for Diploid Genetic Algorithms in Dynamic Environments. Soft Computing, vol. 9, pp. 803-814, 2005.

[11] A. Simões and E. Costa. Transposition: A Biologically Inspired Mechanism to Use with Genetic Algorithms. Proc. of the 4th Int. Conf. on Artificial Neural Networks, pp. 612- 19, Springer-Verlag 1999.

[12] A. Simões and E. Costa. An Immune System-Based Genetic Algorithm to Deal with Dynamic Environments: Diversity and Memory. Proc. of the 6th Int. Conf. on Artificial Neural Networks, pp. 168-174, Roanne, France, 23-25 April, Springer, 2003.

[13] A. Simões and E. Costa. Variable-size Memory Evolutionary Algorithm to Deal with Dynamic Environments. Accepted as full paper in EVOSTOC 2007.

[14] P. Smith. Conjugation: A Bacterially Inspired Form of Genetic. Late Breaking Papers at the Genetic Programming 1996 Conf., Stanford Univ., July 28-31, 1996.

[15] M. Wineberg and F. Oppacher. Enhancing GA's Ability to Cope with Dynamic Environments. Proc. of the 2000 Genetic and Evol. Comp. Conf., pp. 3-10, Las Vegas, USA, 8-12 July, San Francisco, CA, 2000.

[16] S. Yang. Memory-Based Immigrants for Genetic Algorithms in Dynamic Environments. Proc. of the 2005 Genetic and Evol. Computation Conf, Vol. 2, pp. 1115-1122, ACM Press, 2005.

[17] S. Yang. A Comparative Study of Immune System Based Genetic Algorithms in Dynamic Environments. Proc. of the 2006 Genetic and Evol. Comp. Conf, pp. 1377-1384, ACM Press, 2006.

[18] S. Yang. Associative Memory Scheme for Genetic Algorithms in Dynamic Environments. Proc. of the EvoWorkshops 2006, LNCS 3097, pp. 788-799, Springer-Verlag, 2006.