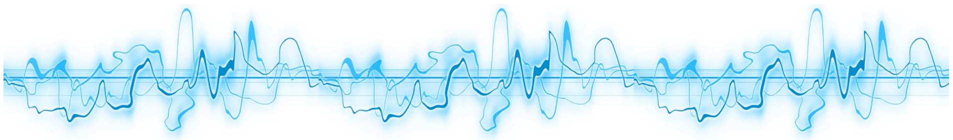
Universidade de Coimbra
Faculdade de Ciências e Tecnologia
Departamento de Engenharia Informática

Improving Memory-based Evolutionary Algorithms for Dynamic Environments



Anabela Borges Simões

Coimbra
March 2010

Improving Memory-based Evolutionary Algorithms for Dynamic Environments

A dissertation submitted to the
University of Coimbra
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
in Informatics Engineering

by

Anabela Borges Simões

University of Coimbra
Faculty of Sciences and Technology
Department of Informatics Engineering

March 2010

Financial support by Fundação para a Ciência e a Tecnologia
through the PhD grant SFRH/BD/39293/2006

ISBN 978-989-20-1900-0

This dissertation was prepared
under the supervision of

Ernesto Jorge Fernandes Costa
Full Professor
of the Department of Informatics Engineering
of the Faculty of Sciences and Technology
of the University of Coimbra

To my beautiful and sweet Daniela.

Acknowledgments

I would like to thank the people who made this research possible:

A special acknowledgment to my teacher, adviser, friend and companion Ernesto Costa. Without his permanent encouragement, strength and persistence this work would have never reached the end.

A special thank you to the ECOS (Evolutionary and Complex Systems) group. Many thanks to Sara Silva for lending me the LaTeX template of her PhD thesis and to Tiago Baptista for his help and support using the cluster.

An enormous thank you to Carolina, Mário and Kim, who worked so hard and willingly to improve my bad English in a short period of time.

To my family and friends, a special thanks to you all.

My last words are addressed to a special person. This thesis was made under difficult conditions, where there were never enough hours in the day to share among the lectures, my family, the thesis and everything else. Thank you, my beautiful little daughter Daniela, for your consistent requests for my company to play, to read or to talk. Thank you, sweetie, for the hours of jokes and laugher that kept me away from this thesis. Thank you, sweetie, for making me realize what is really important and for making me see the world with different eyes.

Anabela Simões

Coimbra, March 2010

Abstract

Evolutionary Algorithms (EAs) are powerful tools for optimization problems. The success of applying EAs to solve hard problems involving static environments is clear and well recognized. Nevertheless, many real-world problems have characteristics and conditions that can change over time. The EAs dealing with this type of problem can face difficulties due to the convergence of the population toward a specific region of the search space. When the environment changes it is hard for this converged population to quickly readapt to the new conditions. Different improvements have been made to the standard EA to make it more robust in dynamic problems: the increase of diversity, the incorporation of memory, the use of multi-populations or the inclusion of anticipation methods.

The use of memory is advantageous when the underlying dynamics of the environment follows a certain pattern. Typically, memory-based approaches react to the change after it has happened and use the memory to help the EA readapt to the new conditions. Also, the memory size is established off-line and kept constant, and is usually a small fraction of the global number of individuals. When the capacity of the memory is attained, a replacing strategy must be used to choose which individual should be deleted to insert a new one.

In this thesis we introduce important and novel contributions, to address some of the drawbacks of current approaches, thus enhancing memory-based EAs for coping with dynamic environments. First, we propose different approaches to make memory more useful and effective: different replacing strategies are proposed, which maximize the capacity and the diversity of the memorized solutions. We also study the influence of the choice of the memory size and propose an innovative algorithm that allows the memory size to evolve to a suitable capacity, according to the moment and characteristics of the dynamic problem. Second, we propose two different biologically inspired genetic operators, which promote different degrees of diversity of the population. We study the effect that different levels of diversity have in the performance of the algorithms. We are interested in analyzing if in memory-based EAs the promotion of high diversity is always necessary and advantageous. Third, we introduce different prediction techniques that allow the EA to forecast both the time of the next change and the direction of this change. Using this information we can anticipate the change and effectively prepare the EA before that change occurs, highly increasing the EA's performance and adaptability.

All the mentioned approaches are tested using different benchmark problems, working under different types of dynamics. The results obtained from an exhaustive experimentation are statistically analyzed, and they prove the effectiveness of the proposed contributions.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Evolutionary Algorithms (EAs) have been successfully used 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 an optimal (or quasi-optimal) solution. However, most of real-world applications are non-stationary and the algorithms used to solve them must be able to adapt to the new circumstances. For this type of optimization, an effective EA must be able to detect changes and rapidly deal with the changes when they occur. Classical EAs are not suited for these kinds of problems, since they have the tendency to prematurely converge to a solution and, when the conditions of the environment change, the population has all individuals usually concentrated in a specific point of the search space. So, it takes some time for the population to readapt and move towards the new solution. To deal with these limitations, some improvements have been proposed as extensions of the classical EA. These improvements include:

- maintaining diversity, using several strategies such as triggered hypermutation, random immigrants, new genetic operators or niching techniques;
- using memory schemes;
- using multi-populations;
- anticipating the changes in the environment.

If the environment changes cyclically and past situations reappear later, the use of memory is beneficial. Memory can be either implicit or explicit. When using **explicit memory** in the EA different design issues must be carefully thought, namely:

- Which is the best choice for the memory size?
- When should memory be updated?
- Which information should be stored?

- When the memory is full, which individuals may be replaced?

Although these subjects have already been studied by several authors (see next chapter), some questions remain unexplored and will be further investigated in this work.
In this thesis we are focused on memory-based EAs and our goal is to make this type of algorithms more robust, efficient and adaptable to situations involving dynamic environments.

1.2 Contributions

This thesis proposes several contributions to the field of evolutionary optimization in dynamic environments, which we believe are novel. These contributions include methods to improve the adaptability, robustness and performance of memory-based EAs facing dynamic environments. The main contributions of this work are centered around three main issues: memory, diversity and prediction.

1.2.1 Improving memory

The **first** contribution, refers to the improvement of **memory** itself. The storage of information is critical when using memory-based EAs, since memory has a limited capacity. It is thus important to decide which individuals must be replaced by new ones (*replacing strategy*). The replacing schemes proposed in this thesis aim to increase the efficacy of the available memory. We propose different methods of replacing the memory individuals when memory is full. Our main concern is to avoid redundancy in the memory and to ensure that the memorized individuals are good solutions for different environments. It is not very effective to have in the memory two or more individuals related with the same environment or to have individuals that were good solutions in the past but are presumably useless in future situations.
We propose three different replacing schemes : two of them are based on the idea of aging of memory individuals. In the first one (called **age1**), all individuals of the memory start with an age equal to zero, and at every generation their age is increased by one. If they are selected to the population when a change occurs, their age is increased by a certain value until a limit age is reached. When the limit age is attained, the individual's age is reset to zero. When it is necessary to update memory, the youngest one is selected to be replaced. In a different age-based replacing scheme (called **age2**) the age of each individual is calculated as a linear combination of its actual age and its fitness. Moreover, in this scheme, memory individuals never die, i.e., their age is not reset to zero. This way, individuals that last long in memory and contributed to the evolutionary process are not penalized. The third replacing strategy (called **generational**), selects for replacement the worst individual present in the memory since the last environmental change. For instance, if the last change occurred at generation t_1 and currently the algorithm is in generation t_2, when it is time to insert an individual into the memory, the worst individual that was stored between generation $t_1 + 1$ and $t_2 - 1$ will be replaced. If no individual has been stored since

the last change, the Branke's *similar* strategy is used and the closest individual in terms of Hamming distance is replaced, if it is worse than the current best.

The second issue analyzed under this topic is how the size of the memory (and the main population) affects the EA's performance and if the choice of these sizes can be optimized on-line. We studied several memory-based EAs using different values for population and memory sizes. Memory-based EAs for changing environments usually use a memory of small size, compared with the global number of individuals. In most cases, the dimension of memory is selected between 5% and 20% of the total number of individuals; 10% is the most frequent choice. Typically, in memory-based EAs the memory size remains unchanged and is always seen as playing a *secondary* role in the process, since it is always used with a smaller value when compared with the population size. This general assumption is questionable. In this context, our work provides relevant information for two important questions: first, what is the best choice for the memory and population sizes? Is the choice of smaller memory size always the right option? Second, can we develop an **EA** that can change the memory and the population sizes according to the stage of the evolutionary process? It might be advantageous to have a smaller memory size at the beginning of the run, and larger sizes in other phases. The proposed algorithm called **Variable-size Memory Evolutionary Algorithm** (VMEA), uses dynamic sizes for both the population and the memory.

1.2.2 The importance of diversity

The **second** contribution, is the proposal of two new genetic operators responsible for promoting the population's **diversity** and the study of how the level of diversity in the population affects the EA's performance. If the environment is dynamic, the success of the EA seems to depend on the population's diversity that the algorithm is able to preserve. If all the individuals of the population are concentrated in a region of the search space, when the environment changes, the EA will have more difficulty to find the new optimum. We looked to the natural systems and found that certain species (viruses and bacteria, for instance) use specific mechanisms for exchanging the genetic material. 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. The second biologically inspired genetic operator is ***transformation***. Some bacteria readily take up outside DNA. If they have this ability they are said to be *competent.* Competent bacteria can absorb fragments of DNA from dead bacteria and present in their environment. Usually, transformation consists of the transfer of small pieces of extra cellular DNA between organisms. These strains of DNA, or gene segments, are extracted from the environment and added to recipient cells. In this thesis, we introduce, implement and test computational counterparts for those two genetic operators that are used as alternatives to the standard crossover operators. An empirical study is performed comparing the diversity promoted by different genetic operators and the corresponding EA's performance under different types of dynamic environments. Results show that, in some situations, high diversity can be detrimental to the performance of memory-based EAs.

1.2.3 Prediction

The **third** contribution, explores the use of **prediction** to anticipate what will happen to the environment and when modifications will occur. If we know the moment when the next change will be observed and the characteristics of the new environment, we can use the information stored in memory in a more effective way and introduce it in the population *before* the change actually occurs. If the environmental changes are not random or chaotic, but instead follow a certain repeated behavior, prediction methods can be used to foresee what is coming. The anticipation consists of preparing the EA to the modifications that the environment will suffer and to avoid the decrease of its performance when the change actually happens. The use of prediction mechanisms in this context is practically nonexistent - this thesis intends to fill this gap and to introduce significant advances in this field. The explored prediction methods work in two levels: the algorithm must be able to anticipate **when** the next change will occur; also, based on previous observations, the algorithm must predict **to where** the optimum will move in the next change. The proposed methods are able to predict both the **time** and the **trend** of the modification. The prediction of the **time** when the next change will occur is explained in two phases: first a linear regression predictor is used. Although effective in many cases, the proposed predictor has some limitations when the change period follows a nonlinear function. To overcome these limitations, a nonlinear regression predictor is implemented and successfully tested. The second predictor is responsible for analyzing how the environment has changed in the past and for predicting the modifications that will occur in the environment. This predictor is supported by a Markov chain consisting of a set of states and transitions. Each state keeps information about the known environments and the states are linked according to the transitions observed in the past. Each transition has a probability that indicates the frequency of the transition between two states. By analyzing all the known states (environments) and transitions, the predictor estimates which environment(s) may appear the next time the environment changes. The proposed modifications are tested using standard benchmark problems, and the results are statistically analyzed to support the claim that they are indeed beneficial.

1.3 Structure

The next chapter provides a brief introduction to evolutionary computation. Chapter 3 presents a general overview about dynamic optimization problems. Chapter 4 describes the state of the art in the field of EAs and dynamic environments. The issues concerning the usage of memory are explained in Chapter 5, in which we detail the proposed replacing strategies and the description of the Variable-size Memory Evolutionary Algorithm (VMEA). Chapter 6 explains the proposed mechanisms for promoting diversity. Chapter 7 shows the implementation of the proposed predictors. Chapter 8 describes the benchmark problems and specifies the parameters used along the experiments. The results of the exhaustive experimental plan are presented in three separated chapters: Chapter 9 reports and analyzes the obtained results concerning memory; Chapter 10 reports the results of the importance of diversity; Chapter 11 provides the results obtained with the prediction modules. Chapter 12 concludes and points towards possible future developments of this work.

As a consequence of this work, several papers were published in top international conferences of the area. In particular, papers where the main topics were *memory* ([91], [89], [90], [93]), *diversity* ([81], [84], [85], [82], [86], [88]) and *prediction* ([92], [80], [83]).

Chapter 2

Evolutionary Algorithms

One hundred and fifty years ago, in 1859, Charles Darwin published *The Origin of Species*, where he explained the process of evolution by natural selection. Together with Greg Mendel's ideas about the mechanisms of heredity, it constitutes today's dominant paradigm about how species originate and evolve, called the *Modern Synthesis*. Population genetics and the modern molecular biology principles, developed in the last century, by and large confirmed these principles and mechanisms. In simple terms, the theory states that the best fit individuals live longer and have a higher chance of reproducing and passing their traits to the next generation. Moreover, the offspring are not exact clones of their parents, but suffer from a process of small random variation of their traits. Those individuals, modified or not, with an increased fitness to the current environment will, again, have a higher chance to survive and reproduce with variation. Throughout time, this process leads to species ideally adapted to their ecosystem. Evolutionary Algorithms (EAs) are stochastic search procedures inspired by the biological principles of evolution by natural selection. In this chapter, we will explain the main, broad, concepts behind these algorithms, illustrating them at the end with a simple example. For an in-depth review about EAs the reader may look into [25].

2.1 Components of Evolutionary Algorithms

Evolutionary Algorithms are based on a simplified model of the theory of evolution by natural selection. The field of Evolutionary Computation comprises several types, or families, of evolutionary algorithms. These include Genetic Algorithms (GAs) [39], Genetic Programming (GP) [47], [46] Evolution Strategies [69] and Evolutionary Programming [29]. Their differences are more historical than based on deep dissimilarities. To solve a particular problem a set of candidate solutions to that problem is randomly created. This set of solutions is called population. Then, the quality of each of these potential solutions is measured [1] and the best ones are selected as parents. The chosen individuals

[1]In nature there is no direct way to measure the quality of an individual, i.e., there is no explicit fitness function, and we talk of *open-ended evolution*. As a matter of fact, in nature

reproduce and undergo a variation process by means of genetic operators, e.g., recombination and mutation, defining a new, offspring, population. Finally, the next generation of the population is formed by combining parent and offspring populations. This process is repeated until a certain stop condition, e.g., number of generations, is attained. Figure 2.1 shows the pseudo code of a standard EA.

```
Function EvolutionaryAlgorithm

Initialize population
Evaluate population
repeat
    Select parents
    Recombine pairs of parents
    Mutate the offspring
    Evaluate the offspring
    Create new population from parents and offspring
until stop_condition is true
```

Figure 2.1: Pseudo code of the standard Evolutionary Algorithm

Selection and variation are the driving forces of the evolutionary algorithm. They determine the regions of the solutions' search space to be explored, and how well they will be explored when looking for good solutions. We say good and not optimal, because due to the stochastic nature of the process there is no guarantee that we will find an optimal solution. EAs, as stochastic search-based procedures, have been successfully used to solve hard, complex problems, i.e., problems for which there is no analytical solution or whose dimension of the search space is so vast, making the problem intractable by conventional computational techniques. Next, the description of each main aspect or concept of a simple EA is provided [2].

2.1.1 Population

A population is formed by a set of individuals, also called chromosomes, typically of a fixed size. Each individual represents a possible solution to the problem and consists of a sequence of smaller components, called genes. Each gene may assume different values, or alleles. At the beginning of the evolutionary process an initial population is generated randomly or using an established seed. This population evolves and, at the end of the process, it is intended that the population has converged to a region of the search space that contains an optimal or quasi-optimal solution. The size of a population is an important design parameter. Combined with the number of generations, it determines the

there is no goal, no plan, no end [28].

[2]As we already said, there are some differences among the representatives of the several families of EAs. Our description is based mainly on the simple genetic algorithm (sGA).

number of fitness evaluations that are needed, and, implicitly, the computational cost of the algorithm. Typically, the size of the population is constant, but there are approaches where this is not the case.

2.1.2 Representation

The choice for the representation of the individuals is made according to the type of problem to solve. The representation defines how the individuals of the population will be encoded. Different types of representations can be used, depending on the problem and the type of EA, e.g., binary, float, integer permutations, trees, graphs. Each encoded solution is called the individual's genotype. The genetic operators act on these representations and, therefore, the selection of a good representation is a critical design choice.

2.1.3 Fitness Function

The fitness function is used to measure the quality of the individuals of the population. To measure it, a decoding process is needed to obtain the individual's phenotype. The fitness is a real value obtained by applying the fitness function to the phenotype. Depending on the type of problem, the best fitness can be the highest (maximization problems) or the lowest (minimization problems) values.

2.1.4 Selection

The selection method is used to choose a pool of parents based on their fitness. Solutions with higher fitness values have more probability to be chosen for mating. This way, a more fitter population is created and the EA evolves towards the optimal solution. There are different selection methods that can be used: roulette wheel, tournament-based or ranked-based selection. The roulette wheel method selects individuals based on their relative fitness: a higher fitness yields to a higher probability of being selected. In the tournament selection, a certain number (tournament size) of individuals is randomly selected, and the best individual is included in the mating pool. Rank-based selection orders the individuals according to their absolute fitness. Each individual has a certain probability of being chosen, depending on its position in the rank.

2.1.5 Genetic Operators

The role of the genetic operators is to create variation among the individuals of the population. Genetic operators can be divided in two main categories: recombination (or crossover) and mutation. Recombination is applied using two (or more) selected parents and mixing the genetic content of them. The simplest example of that type of operator is called one-point crossover. It is applied on two individuals selected from he mating pool. One random point is chosen, i.e., a point on the frontier of two genes, and the genetic material of the two selected parents is swapped. We can also use two, or **n**, points to control the exchange of genes among the two parents. Another well-known crossover operator is called uniform crossover where the genes of two parents are swaped according to a random generated mask. Mutation is applied to the individuals' genes by making a small change in their corresponding alleles. There have been many studies made about the relative merits of both operators. In general, crossover promotes the exploration of the search space, and thus is very important at the beginning of the search, when the algorithm is searching for a promising region of the search space. Mutation, on the other hand, permits the exploitation of the search space and is important at the end of the search when the algorithm already converged the population into a good region of the search space. Of course there is an interplay between the genetic operators and the representation. This means that for different representations, different operators must be used. The genetic operators are applied according to a chosen probability [41]. Usually, a small probability (1% to 5%) is used in mutation, whereas crossover is applied with a probability of 60% to 70%.

2.1.6 New Population

Once we have generated a population of offspring we still have a design decision to make. In fact, there are two main ways of producing the population of the next generation. One is to kill all the parents and keep all the offspring. This is called the generational approach. Another possibility is to merge the two populations and choose the best ones to pass on to the next population. One possible variation is to keep a small fraction of the best parents and fill the rest of the population with the best offspring. This approach is called elitism.

2.2 An Example

To show how the components of an EA work together we will use a simple EA to evolve a string that matches a given template. Suppose that the goal is to evolve the string "ANABELA SIMOES". Each candidate solution in our population, i.e., each individual, will be a 14 characters long, fixed-length, string. A gene will be a position in that string and for each of them the possible alleles are the 27 upper case letters from 'A' to 'Z' plus the space character. In this simple case, there is no difference between the genotype and the phenotype. The fitness function will measure how many characters are already in the correct position. So, the best fitness, corresponding to the string "ANABELA SIMOES" will be 14. This is a maximization problem.
The first step of the EA is to generate a random initial population. In this example we will use a population of size 10. Suppose that we obtain the following initial population (in brackets is the fitness of each individual):

```
1. AXDBIEA XEESAS  (5)
2. SS OSSAXCIESSD  (2)
3. BBEER SSSNE SZ  (1)
4. ZAQ DDDWQ SSQW  (0)
5. CVFFREAWWEV XX  (1)
6. ANAAAAA BBMOXS  (8)
```

```
 7. ALABERSRRBSIES  (5)
 8. XMOBELAC IMOER  (8)
 9.  BSSEAJ LIASEE  (4)
10. CCBOPSS SSW WW  (0)
```

Using the roulette wheel selection method, the best individuals have a higher possibility of being selected for the mating pool [3]. Using this method, the mating pool of parents could be the following:

```
 1.  XMOBELAC IMOER
 2.  ANAAAAA BBMOXS
 3.  XMOBELAC IMOER
 4.  ALABERSRRBSIES
 5.  ANAAAAA BBMOXS
 6.   BSSEAJ LIASEE
 7.  ANAAAAA BBMOXS
 8.  SS OSSAXCIESSD
 9.  XMOBELAC IMOER
10.  XMOBELAC IMOER
```

The next step is the application of the genetic operators. We will use one point crossover, with a probability of 60%. With that value for crossover's probability and with a population of 10 individuals, we expect that (on average) 6 individuals will undergo crossover. For those pairs of parents selected we randomly choose a crossover point and bond the left part of the first parent with the right part of the second parent. Similarly, the right part of the first parent is joined with the left part of the second parent. Suppose that parents 1 and 2 are selected to recombine their genetic material.

```
Parents:
   1.  XMOBELAC IMOER
   2.  ANAAAAA BBMOXS
```

If the randomly chosen point of crossover is position 3, the new individuals will be:

```
Offspring:
   1.  XMOAAAA BBMOXS
   2.  ANABELAC IMOER
```

Crossover is used between the number of parents necessary to generate the desired number of individuals. After crossover, mutation is applied, say with a probability of 5%. Those positions (genes) chosen to undergo mutation will have their character (allele) changed to a new one. Suppose that the first individual mutates the gene 1 and 4 and the second the gene 8, changing to different and valid characters:

[3]With this method individuals of fitness 0 will never be chosen.

```
Mutated Offspring:
   BMOBAAA BBMOXS
   ANABELA  IMOER
```

After using crossover and mutation, this could be the generated offspring. The fitness values are already updated and it is clear that this population has better individuals than the previous one.

```
    1. BMOBAAA BBMOXS  (4)
    2. ANABELA  IMOER  (12)
    3. AMOBELARRBSIES  (7)
    4. ANABERSC IMOER  (9)
    5. ANABELA  IMOER  (12)
    6. ANAAAAA BBMOXS  (8)
    7. ANABAAAC IMOER  (9)
    8. AMOBELA BBMOES  (10)
    8. XMOBELAC IMOER  (8)
   10. AMOBELAC IMOES  (10)
```

Using a generational approach to select the next population, we repeat the process of selection, crossover and mutation for several generations until the algorithm eventually converge to a population containing the desired solution.

Chapter 3

Dynamic Environments

EAs have been used to solve complex, real world, optimization problems. In most situations these problems are embedded in contexts that make them hard to solve. For example, if the quality of a candidate solution depends on measurements made by sensors, the fitness function is typically noisy. Another situation appears when we have to simplify the fitness function, using an approximation, to reduce the computational costs of our algorithm. Several solutions have been proposed in the literature to deal with these kind of uncertainties. A third situation where uncertainties can be found, occurs when the problem has an associated time-varying fitness function. This last category of problems is said to operate in **dynamic environments**.
This thesis is focused on the application of EAs to situations dealing with dynamic environments. In this chapter, we describe the main concepts, namely, how dynamic environments can be classified according to the type of modifications, the benchmark problems used to test and evaluate the algorithms, and different performance measures.

3.1 Categories of Dynamic Environments

Dynamic environments can be classified in different ways. For example, Branke [14] categorized the environments using certain parameters of the problem: the frequency of change, the severity of change, the predictability of change and the periodicity of the change (i.e. the cycle length). A different categorization [42], used a direct description of the problems, classifying the problems in cycle, with changing morphology, drifting landscapes or abrupt and discontinuous. Weicker [110] proposed a classification of the dynamic environments consisting of a combination of these two.
In this thesis we propose an alternate way of classifying the dynamic environments by dividing the changes into two main groups depending on **when** the environment changes and **how** the environment changes.

3.1.1 When does the environment change?

The time **when** the changes occur is defined by the change period, which consists of the number of generations between two consecutive changes. Knowing the characteristics of the change period, different decisions can be made concerning the design of the EA. The change period has three main aspects to be considered: (1) the recurrence of the observed change periods, (2) the length and (3) the predictability.

(1) Types of change period

The change period can be classified as:

- **Periodic or linear**: the changes are observed at fixed intervals. If the change period is called r, in this type of classification, changes are observed every r generations.
- **Patterned**: the interval between the changes is not constant, but instead follows a repeated pattern. For instance, if we observe the first change at generation 5, the second at generation 15, the third at generation 20, the fourth at generation 25, and so on, we can say that the change period follows the pattern 5-10-5.
- **Nonlinear**: the generations where the changes are observed follow a nonlinear function.
- **Random**: the changes happen at random points without any pattern or periodicity.

Figure 3.1 shows the described types for change periods.

(2) Frequency of the change

Another important characteristic to be analyzed is how often the environment changes. Changes can occur every generation or at larger intervals. For example, in the case of periodic or linear changes, this means a different value of r. Smaller values of this parameter mean faster changes, which are typically harder to deal with. This is an important issue, since in most approaches using EAs, the algorithm only reacts after the change is detected and the frequency of change can determine if the EA is able to quickly readapt to the new environmental conditions.

(3) Predictability of the change

If the change period follows a linear or a repeated trend, we can say that the moment of the next change can be predicted. However, if the change period is completely chaotic, no prediction is possible. This aspect is very important if we are interested in designing EAs that can react **before** the change happens. If prediction is possible, the population can be prepared to the next change before it actually happens.

3.1.2 How does the environment change?

Knowing how the environment changes is important in deciding if the incorporation of a memory component will be useful to the EA or if the application of other methods can be more effective.

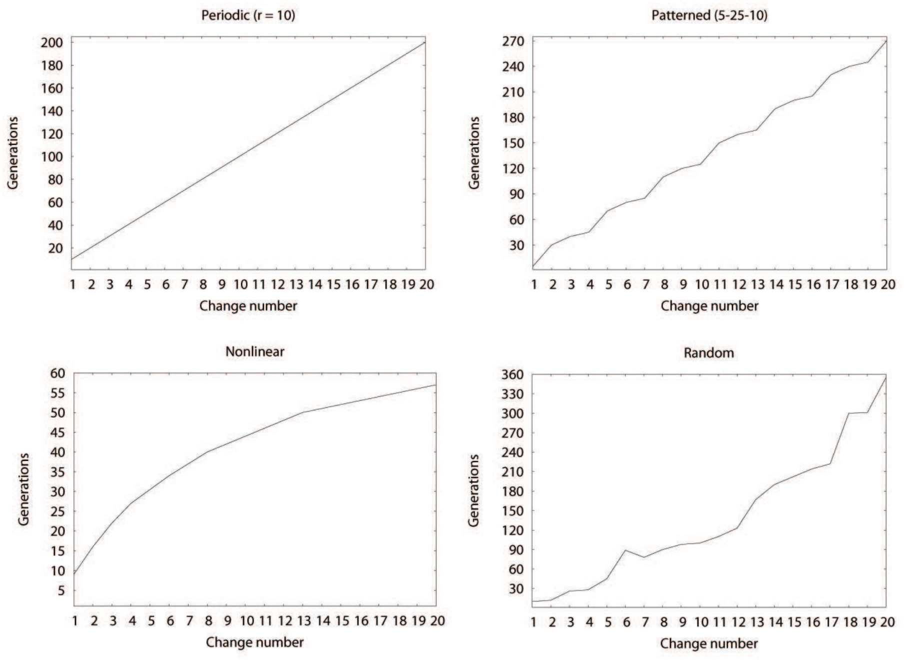


Figure 3.1: Types of change period

(1) Types of environmental changes

- **Cyclic**: in a cyclic environment, situations from the past reappear in the future in a cyclic manner. In this type of environment the number of different environments can determine the difficulty of the problem. We say that an environment is cyclic if the environments reappear always in the same order (A-B-C-A-B-C . . .).
- **Cyclic with noise**: environments from past reappear but with small differences introduced by a noise factor (A-B-C-A'-B'-C'- . . .).
- **Probabilistic**: when the transition between a fixed number of environments is governed by some probability.
- **Random**: the environments change from a state to another completely different state without any correlation with the past.

Figure 3.2 shows the described types for environmental changes [1].

In this work we will use cyclic and probabilistic types of environmental changes.

(2) Severity of change

The severity of change measures the strength of the modifications in the environment. The environment can change to a completely different state or to a similar one.

[1]In the probabilistic type, no value attached to transitions means probability equal to 1

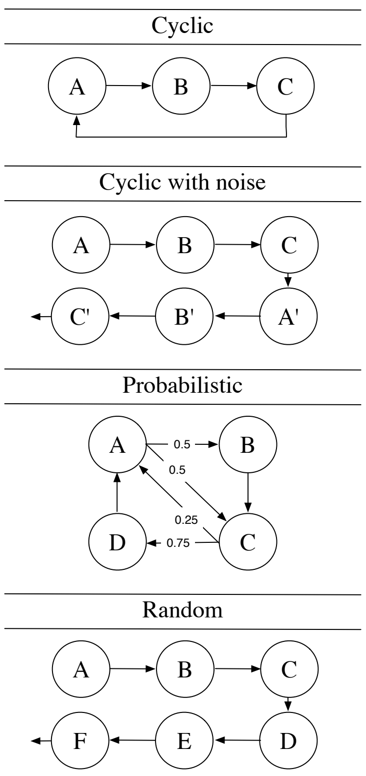


Figure 3.2: Types of environmental changes

(3) Predictability of the new environment
If the transitions between the environments are cyclic or follow a trend that can be captured by the algorithm, it is possible to predict which modifications will be observed at the next change. All of the previously mentioned types of environments, except the random ones, present some predicability.

3.2 Benchmark Problems

Benchmark problems are used to test the performance of the EAs in solving different types of dynamic environments. Over the years, different benchmark problems have been used. Those benchmarks have different characteristics and span from simple mathematical functions to more complicated real-word applications. Other problems, used for static environments, can be easily changed and configured to test the EAs in different types of dynamic environments.

Benchmark problems can be divided into two types. The first type includes problems where the environment is switched between different instances of a specific stationary problem. In this category the most popular benchmarks are:

- **Dynamic Bit-matching**: the bit-matching problem is an unimodal problem which goal is to find a solution that matches a given template. Changing the template from time to time makes this problem dynamic. This problem is encoded with binary representation and the number of bits that change in the template can define the severity of the change. The difficulty of the problem can be increased by using templates with larger dimensions.
  The onemax problem, where the EA has to find a solution that maximizes the number of ones, is a particular case of the bit-matching problem.

- **Dynamic Knapsack Problem**: consists of the dynamic version of the popular knapsack problem. There is a set of n items that have weights and values. The goal is to choose the items that maximize the values, but whose sum of weights doesn't surpass the capacity of the knapsack. Usually, this problem is encoded with binary representation and each gene indicates if an item is selected to be included in the knapsack (gene is 1) or not (gene is zero).
  In the dynamic version of this problem the capacity of the knapsack changes over time. This means, for instance, that, if the capacity is reduced, a previous good solution may become invalid.

The second type of benchmark problems uses a basic function which is changed to construct the dynamic environments. In this category we can find the following test problems:

- **Moving Peaks**: the moving peaks benchmark was proposed by Branke [14] and Morrison et al. [61]. It consists of m peaks, whose height (h_i), width (w_i) and location ($\vec{p_i}$) can change. The m peaks belong to a n-dimensional real space and the fitness landscape is defined as the maximum over all peaks. The environmental changes are made by changing those three parameters after an established number of evaluations that defines the frequency of change.

- **Moving Parabola**: this problem uses a real-numbers encoding and is analogous to the dynamic bit-matching problem. A basic parabolic function is used and the dynamics of the environment is created by moving the parabola in the space. The difficulty of the problem depends on the number of dimensions. The severity of the change is determined by the size of the shift and different types of dynamics can be applied to the parabola: circular, linear or random [1].

- **Dynamic Optimization Problems Generators**: the most used generator is the Dynamic Optimization Problems (DOP) generator introduced by Yang [126]. It is used in problems using binary encoding and is applied to construct different types of dynamic problems from any stationary function $f(x)$. The dynamics in the environment is created through the application of the exclusive-or (XOR) operator to the binary string that encodes a possible solution. The basic idea of the DOP generator can be described as follows: when evaluating an individual x in the population, first it is performed the operation $x \otimes M$ where $\otimes$ is the bitwise XOR operator and M a binary mask previously generated. Then, the resulting individual is evaluated to obtain its fitness value. The characteristics of the problem are controlled by two parameters: r is the change period and ρ is the severity of the change, consisting of the ratio of ones in the mask M. Three different types of environment can be constructed: cyclic, cyclic with noise and random. Later, Tinós et al. [98] proposed a different version of this generator to be used with real-valued encoding. A generalized framework of those generators was proposed and tested by Li et al [49].

- **Problem generator based on deceptive functions**: this problem generator was proposed by Yang [113]. It uses a decomposable trap function to construct a problem generator based on the problem difficulty. The generator starts from a base stationary function consisting of m basic trap functions which are justaposed and summed together. A scaling factor is used to define the weight of each trap function.

Other problems can be used as benchmark for testing EAs solving dynamic environments. For instance, the job shop scheduling problem, involving a set of n jobs and m machines. Each job is a sequence of n activities so there are $n * m$ activities in total. Each activity has a duration and requires a single machine for its entire duration. The activities within a single job all require different machines. An activity must be scheduled prior to every activity following it in its job. Two activities cannot be scheduled at the same time if they both require the same machine. The objective is to find a schedule that minimizes the overall completion time of all the activities. If a new job arrives when the scheduling is already defined, it is considered that the environment has changed and it is necessary that the EA finds a new solution to the problem. Evolutionary approaches that use this problem can be found in [9], [10], [16].
Other optimization problems with multi-objective characteristics were used to test EAs for dynamic problems [26], [27], [40]. Following Branke's comment about the inconvenience of using real-world applications, we will be focused on artificial benchmark problems, like Dynamic Knapsack, Dynamic Bit-matching and the DOP generator. These problems enable us to better control the possible dynamics, making an in-depth analysis and comparison of different results possible, thereby allowing us to assess the quality of our proposals.

3.3 Performance Measures

Performance measures allow to measure the performance, efficacy and adaptability of an EA. These aspects can be evaluated by numeric values whose evolution in time can be plotted for visual inspection. Those numeric values are:

- **On-line performance**: consists of the average of all evaluations over the entire run. More formally:

$$\textit{on-line}(t) = \frac{1}{N} \sum_{t=1}^{N} e_t$$

where N is the number of evaluations and e_t is the evaluation at time t.

- **Off-line performance**: is calculated as the average of the best individuals' fitness observed so far at each time step:

$$\textit{off-line}(t) = \frac{1}{N}\sum_{t=1}^{N} {e_t}^*$$

where N is the number of evaluations and ${e_t}^*$ is the evaluation of the best individual since the last change.

- **Fitness Overall**: consists of the average of the best individual's fitness observed at each generation

$$overall = \frac{1}{G}\sum_{t=1}^{G} best_t$$

where G is the number of generations and $best_i$ is the fitness of the best individual at generation t.

- **Average Error**: this measure can be used if the desired optimum is known. It is calculated as the difference between the optimum and the best individual found by the algorithm. The average error can be used in an on-line or off-line version.

- **Accuracy**: is based on a measure proposed by De Jong [41], the off-line performance. This can be used if the optimum is a known value. It evaluates the difference between the value of the current best individual and the optimum value, instead of evaluating only the value of the best individual. Accuracy consists of the difference between the value of the current best individual in the population of the "just before change" generation and the optimum value, averaged over the entire cycle. Accuracy measures the capacity to recover to the new optimum before a new modification occurs. If the accuracy reaches a zero value it means that the algorithm found the optimum every time before a change occurred [99].

$$acc = \frac{1}{K}\sum_{i=1}^{K} Err_{i,r-1}$$

where K is the number of changes during the run, r the number of generations between two consecutive changes and $Err_{i,r-1}$ is the difference between the fitness of the best individual at change i, just before change, and the desired optimum.

- **Adaptability**: also inspired on De Jong's measure, it is used combined with accuracy. It consists of the difference between the value of the current best individual of each generation and the optimum value averaged over the entire cycle. Adaptability measures the speed of the recovery. If adaptability is equal to zero it means that the best individual in the population was at the optimum for all generations, i.e., the optimum was never lost by the algorithm [99].

$$ada = \frac{1}{K}\sum_{i=1}^{K}\left[\frac{1}{r}\sum_{j=0}^{r-1} Err_{i,j}\right]$$

where K is the number of changes during the run, r the number of generations between two consecutive changes and $Err_{i,j}$ is the difference between the fitness of the best individual at generation j, after the last change, and the desired optimum after the i^{th} change.

In this thesis we will use the *off-line* performance and the *fitness overall*, which are sufficient to a complete and sound analysis of the performance. More information about measures used to evaluate the performance of EAs for dynamic environments can be found in [109].

Chapter 4

State of the art

Through the years various methods and techniques have been developed to cope with the difficulties raised by dynamics environments. In the following sections we review the most significant approaches for empowering EAs to deal with dynamic environments that were reported in the literature. The different methods are divided in four main classes: promotion of diversity, using memory, multi-populations approaches and anticipation of the changes.

4.1 Diversity

The main disadvantage of applying a conventional EA in a dynamic environment is that, once the algorithm starts to converge around some optimal or near-optimal solution, it will very likely lose its ability to continue the search for a new optima, when the environment changes. Hence, one key point in optima-tracking approaches is the need to increase or maintain high diversity among the individuals in the population, so that the algorithm retains its ability to explore the new search space when the problem changes, even after when the population has partially converged to an optimum or near optimum solution. To overcome this limitation different methods have been used to either increase the population's diversity after a change or to maintain the population's diversity throughout the entire run. The next section describes the relevant work about this topic.

4.1.1 Generating diversity

The methods described in this section aim to increase the population's diversity after a change is detected.
The first category consists of restarting the EA every time a change is detected. This approach is simple to implement and is a valid option, if the changes are too severe or if the representation changes after a modification in the environment. Since no information from the past is transferred to the actual configuration of the EA, this method is not suitable for all types of dynamic optimization problems. We can find this method described in [36], [45] and [44].

A different approach to promote diversity after a change consists of re-initializing the EA with individuals from the previous populations.
Ramsey and Grefenstette [68] introduced a case-based method for initializing the genetic algorithm when a change is detected. Louis and Xu [53] applied the same idea to the open shop scheduling problem. They used an EA combined with case-based reasoning. Good solutions from previous populations were memorized and when the problem changed the old solutions were injected in the EA's new population that was trying to solve the new problem. These methods will be detailed next, in the section about explicit memory approaches.

Another approach to promote diversity after a change is to adapt the mutation rate. Triggered hypermutation was proposed by Cobb [19] and Grefenstette [20] for reintroducing diversity into an EA population operating in changing environments. In these works the authors explored the use of mutation as a control parameter for enhancing optimization in an incrementally changing environment. This method was used with a standard generational EA, with a fixed population size, proportional selection, N-point crossover, and a small mutation rate (0.001) that was applied uniformly to the population. Where the algorithm differed from standard EA was that the small mutation rate (called the 'base' mutation rate) was not always the mutation rate that was applied to the population. The algorithm was adaptive, since the mutation rate did not remain constant over time. When a change in the fitness landscape was detected, the mutation rate was multiplied by a *hypermutation factor* before it was applied. Morrison and De Jong [62] revisited this mechanism and examined the effects on EA performance based on the relationship between the hypermutation factor and the environmental change period.
Angeline [1] compared the use of self adaptive mutation rate against an evolutionary program without self-adaptation for tracking the optimum in dynamic environments. The comparative study was applied with different dynamics, and tested with a simple static function. The authors concluded that, for some dynamic functions, self-adaptation was effective while for others it was detrimental.
Vavak and his colleagues [106] proposed a mutation operator, denominated Variable Local Search (VLS). The GA using this adaptive operator was applied on two industrial applications where the GA was responsible for the on-line control systems. The VLS slightly increased the mutation rate when a change was detected. The authors compared their approach with the hypermutation operator. The experimental results showed that the VLS operator outperformed the triggered hypermutation operator for small shifts in the environment.

4.1.2 Maintaining diversity

Instead of generating diversity only when the change happens, other approaches use methods that maintain the diversity during the entire run. This section discusses relevant work concerning this topic.
Grefenstette introduced the idea of the random immigrants [35]. Random immigrants consisted of the replacement of a percentage of the population by

randomly generated individuals. The percentage replaced was called the *replacement rate*. This replacement was made in every generation and aimed to maintain a continuous level of exploration of the search space, while minimizing the disruption of the ongoing search.
Exploring the same idea, Yang et al. [124] used a hybrid immigrants scheme and compared their approach to Grefenstette's random immigrants. This method combined the concepts of elitism, dualism and random immigrants: the best individual from the previous generation and its dual individual were retrieved in order to create immigrants via mutation. These elitism-based and dualism-based immigrants together with some random immigrants were substituted into the current population, replacing the worst individuals in the population. The strategy for using these three kinds of immigrants was to address environmental changes of slight, medium and significant severity, respectively. Related work using similar technics can be found in [122], [123] and [18].
Another random immigrants scheme was proposed by Tinós [97]. In this work, the worst individuals of the population were replaced by new ones. The replacement of an individual could affect other individuals in a chain reaction. A subpopulation was maintained with the new individuals created in the current chain reaction. One single replacement could affect a large number of individuals in the population. The system could exhibit a self-organizing behavior, useful to control the diversity of the population allowing the algorithm to escape from local optima when the environment changed.
In the *Thermodynamical GA* proposed in [57] the population's diversity was explicitly controlled using a measure called "free energy" (F). This algorithm used the concepts of temperature (T) and entropy (E) as in the simulated annealing, and maintained the diversity of the population explicitly and systematically. The free energy F was computed using the fitness function, the entropy and the current temperature. The individuals with minimum F were preserved for the next generation and the temperature T was used to control the diversity of the population. The algorithm was tested on the time-varying knapsack problem. A modified version of this algorithm, called *Feedback Thermodynamical GA* (FTDGA) was proposed in [59] with some enhancements, namely the adaptive control of the temperature F to regulate the level of diversity of the population.

Different approaches modify the selection methods for promoting population's diversity during the entire run. One of these methods consists of building *niches* that force the individuals of the population to explore different areas of the search space. The most popular methods for creating niches are *sharing* and *crowding*. Sharing was first introduced by Holland [39] and further explored by Goldberg et al. [31]. When using a sharing mechanism, the individuals in the same region of the search space shared their fitness. The fitness of the individuals that were highly similar was reduced. This method penalized redundant individuals while rewarding the individuals that were isolated, thus enabling the exploration of different areas of the search space. Crowding was introduced by Holland [39] and selected a proportion of the population (called generation gap (G)) to reproduce each generation. For each offspring, a certain number of individuals, called *crowding factor* (CF), were randomly selected and the most

similar individual was replaced by the offspring. By choosing the most similar member for replacement, the crowding mechanism slowed down the tendency of the EA to converge to a single point of the search space. Sharing and crowding mechanisms were used in EAs for dynamic environments by Cedeno and Vemuri [17]. This work used a replacement scheme called *worst among similar* (WAS) to promote competition among similar individuals while allowing competition among members of different niches as well. The authors showed that the proposed method was capable of tracking different regions and adapting to new peaks that appeared in the landscape.
Other works using these techniques in dynamic environments can be found in [75], [77], [50], [54].

Morrison [60] proposed a different approach to promote diversity based on the placement of *sentinels* in different regions of the search space. Sentinels were a sub-set of the population and were uniformly distributed through the search space. They consisted of normal individuals of the population, that could be selected and recombined, but were fixed, since they could not be replaced or mutated. Through the use of sentinels, different regions of the landscape were tracked. When a change happened, if the population had converged to a peak that was no longer the optimum, sentinels from other regions were used to create new individuals that increased the diversity of the population allowing the readaptation of the EA.

The use of the immune system (IS) ideas in the EA is another method for promoting the diversity through the run. The IS is a complex, distributed and multi-layered system, which includes cells, molecules and organs that constitute an identification mechanism capable of recognizing and eliminating foreign molecules called antigens. The human body maintains a large number of immune cells. Some belong to the innate IS, e.g. the macrophages, while others are part of the acquired, or adaptive IS and are called lymphocytes. There are mainly two types of lymphocytes, the B-cells and the T-cells, which cooperate but play different roles in the immune response. The main functions of the B-cells are the production and secretion of antibodies as a response to exogenous organisms. Each B-cell produces a specific antibody, which can recognize and attach to a specific pathogen. In order to do their job correctly the B-cells replicate by a process called clonal selection. This process is similar to the evolution of a population using a genetic algorithm with mutation and without recombination. The B-cells that have antibodies, which bind to the pathogen, are selected and cloned. Nevertheless, during cloning some variations may occur due to a process of somatic hypermutation. This may increase the affinity between the antibody and the antigen, making the B-cell more adapted to bind to the antigen. The foregoing discussion of the immune system was applied to promote diversity in EAs for dynamic environments by Simões and Costa [87], Yang [118] and Liu et al. [51].

Uyar et al. [103] used a diploid representation of individuals with a dynamic dominance map mechanism and meiotic cell division that to help preserve diversity.

4.2 Memory

In dynamic problems, memory is used to store successful past solutions with the assumption that the optimum may return to its former value. When certain aspects of the problem exhibit some kind of periodic behavior, old solutions might be used to bias the search in their vicinity and reduce computational time. The use of memory is beneficial on those types of environments. Memory-based approaches can be divided in two categories: implicit memory and explicit memory.

4.2.1 Implicit Memory

The use of redundant representations is the main characteristic of the implicit memory methods. Those approaches use diploid or multiploid chromosomes to implicitly memorize the best individual of the population. A dominance scheme controls which genes are expressed in the phenotype. Diploid representations were first used by Goldberg et al. [32]. They introduced a triallelic dominance scheme, where the value of each allele could be 0, dominant-1 or recessive-1. Their method was tested in the dynamic knapsack problem and the results were significantly better than with the standard GA. A different dominance method was suggested by Ng and Wong [64]. They used four possible alleles (0-recessive and dominant and 1-recessive and dominant). The proposed method analyzed the individuals' fitness: whenever the individuals' fitness decreased more than 20%, the dominance mechanism changed from dominant to recessive and vice-versa, resulting in the inversion of all alleles pairs. Other works using implicit memory based on diploid representations and dominance mechanisms can be found in [74], [48], [102], [104], [119].
Implicit memory can also be implemented using dualism mechanisms as suggested in [125]. This mechanism was inspired by the complementarity and dominance mechanisms present in natural systems. A pair of chromosomes was called primal-dual to each other, if their Hamming distance was the maximum in the search space. In every generation, after selection, recombination and mutation being performed on the population's individuals, a set of those individuals were selected and used to evaluate their duals. If the dual chromosome was fitter, the primal was replaced, otherwise the primal survived and was preserved in the next generation. Other works using dualism mechanisms were described in [107], [67] and [108].

4.2.2 Explicit Memory

The memory-based approaches using explicit memory need an extra space to explicitly store information about the individuals or the environments. In these methods, one population performs the search for the optimum and the other population, called memory, stores useful information that is reused later. Concerning which information is stored in memory, Yang [121] divided explicit memory approaches into two main groups:

- **Direct memory schemes**: store the best individuals and reuse them when a change in the environment is detected.
- **Associative memory schemes**: store the best individuals and the environmental information associated with these individuals. When a change is detected, both information is used to create new individuals which are introduced into the population.

A third group can be added to this classification - **immigrant memory schemes** - which store the best individuals of the population and use them to create immigrants that are introduced into the population.

Direct memory schemes

Ramsey and Grefenstette [68] introduced an EA model that stored good candidate solutions for a robot controller in a permanent memory together with information about the environment. The idea was that if the robot environment became similar to a stored environment instance, the corresponding stored solution was reactivated. The authors claimed that their technique prevented premature convergence by a higher level of diversity and yielded significant improvements.

Louis and Xu [53] applied the same idea to the open shop scheduling problem. Whenever a change occurred, the EA was restarted and the population was initialized with a seed from the old run and the rest randomly. The authors concluded from the experiments that a seed of 5-10 % from the old run produced better and faster results than running the EA with a totally randomly initialized population after a change occurred.

Mori et al. introduced a memory-based EA called *memory-based thermodynamical GA (MBTDGA)* proposed in [58]. The algorithm was an extension of the thermodynamical GA explained before. The MBTDGA used a search and a memory population, both controlled by a measure called "free energy" (F). The best individual of the population was memorized at every generation replacing an existing individual. The individual to be replaced was selected based on its age and its energy F. The new population was created by selecting from the old parents and the offspring the individuals that minimized the factor F. The algorithm was tested on the dynamic knapsack problem.

Branke proposed a memory/search EA [13] which used two populations, named memory and search populations. The memory stored good solutions, while the search population searched for the optimum. The seracg population was reinitialized with the memory individuals whenever a change was detected. A variant of this algorithm was also analyzed by Branke. This variant used two independent search populations, plus the memory population. Later [14], a multi-population method was combined with the memory.

Bendtsen et al. [8] used a dynamic memory EA. The memory was used to keep track of the dynamic changes, instead of storing old solutions. A fixed number of solutions were kept in the memory but were not replaced by others. The memory solutions were adjusted to the changes in the search environment. When a change was detected, the memory individuals were adjusted as follows: the best individual in the population was selected and the closest (most similar) stored candidate solution was found. The closest stored individual was gradually moved towards the currently best individual in the population. This direction of the movement was decided by calculating the direction from the genotypes of the individual to the best individual. The distance of the movement corresponded to the distance between the two locations multiplied with the absolute value of a Gaussian random number with a mean of zero and a variance of 0.5. Afterwards, the memory individual was introduced into the search population by replacing the worst individual.

A memory scheme inspired on the B-cells of the natural immune system was introduced by [87]. In order to mimic the natural immune system, the algorithm used gene libraries, clonal selection with somatic hypermutation and memory B-cells. The environment was saw as the antigen and the changes in the environment as the appearance of different antigens. This algorithm used two populations of individuals. The first one consisted of plasma B-cells individuals which stored the best individuals of the population. When a change occurred, the individual with best match to the optimum (antigen) was selected and cloned. During the cloning phase every individual was modified by a mechanism similar to the somatic hypermutation of B-cells. Other approaches of this algorithm to solve dynamic problems were investigated by [118] and [51].

Yang proposed a direct memory scheme, called memory-enhanced genetic algorithm (MEGA) [114]. This algorithm used a search and a memory population with fixed sizes. The search population evolved by using selection, crossover and mutation. From time to time, the memory was updated storing the current best of the population. Whenever a change was detected, the two populations were merged and the best p individuals (p was the population size) were selected to the next population, while the memory remained unchanged.

Associative memory schemes

Associative memory schemes use the memory to store the best individuals from the population and additional information about the environment. This information is used to create new individuals every time a change occurs.
Trojanowski et al. [100] introduced an evolutionary algorithm extended by two different mechanisms: a memory and a diversity maintenance technique. The memory was used to store individuals, and additional information associated to the individual's ancestors. The memory started empty, and each time after

a new individual was created, if it was included in the next generation of the population, the active chromosome of its parent (or a better parent - in case there are two parents) was added to the memory. In addition, the memorized solution inherited the chromosomes in the memory of its parent or better parent. When the memory buffer was full the oldest chromosome was deleted to make room for a new one. When a changed was detected, the best individual in the memory and the associated information, was used to update the population. Additionally, this algorithm used random immigrants that were introduced into the population, every time a change was detected.

Yang [126] proposed the population based incremental learning (PBIL) algorithms used for dynamic optimization problems. This algorithm used a probability vector to store the allele distribution of the individuals of the current population. This vector represented the environmental information associated with that population. When the memory was updated, the distribution vector was stored and associated with the best memorized individual. When an environmental change was detected, the best individual in the memory and the associated probability vector were retrieved and used for creating new individuals to replace the worst of the population. Different applications of this method was used in [126], [117] and [127].

An analogous memory indexing algorithm was proposed in [43]. The authors proposed a memory indexing algorithm (MIA) which also used a vector with the alleles distribution. The memory stored a distribution vector for each different environment (or group of similar environments) encountered during the run. When change occurred, the distribution alleles of the population was calculated and was stored in memory, associated to a group of similar environments. If a similar environment was found before, a percentage of the population was re-initialized using the corresponding distribution alleles, otherwise a standard hyper-mutation method was applied.

A different memory scheme based on abstraction was proposed in [70]. This abstraction based memory mechanism stored the good solutions as an *abstraction*. An abstraction of a good solution was defined by its approximate location in the search space. Memory corresponded to a n-dimensional matrix whose elements represented search sub-spaces. During the memorizing process, the selected individuals were sorted according to their partition in the search space which they represent. When a change was detected, the memorized individuals were selected according to their corresponding partition, mutated and inserted into the population. Further investigations of this algorithm were performed in [71].

Barlow [7] investigated a memory-enhanced evolutionary algorithm to the dynamic job shop scheduling problem. This memory-based EA used a classifier-based memory for abstracting and storing information about schedules, that was used to build similar schedules at future times. The authors compared their approach with a standard EA and several common EA diversity techniques both with and without memory.

Immigrant memory schemes

Other memory-based EAs use immigrants in order to increase the population's diversity.
The approach proposed by Trojanowski et al. [100], previously described, used random immigrants combined with memory. The random immigrants were created every time a change occurred, replacing a percentage of the individuals of the population.

A similar algorithm, called Memory Random Immigrants Genetic Algorithm (MRIGA) was studied in [123]. MRIGA combined memory and random immigrants. The memory was updated with the current best individual of the population and was used whenever a change occurred. At this time, the population and the memory were merged and the the best p individuals (p was the population size) were selected to form the next population. Before applying selection, crossover and mutation, a number of random immigrants was generated and inserted into the population, replacing the worst solutions.

Yang introduced a different immigrant memory scheme called Memory Immigrant Genetic Algorithm (MIGA) [114]. In this algorithm the memory, besides storing the best individuals of the population, was used to create immigrants. In every generation, the best individual from memory was selected to create immigrants by mutation. Those immigrants replaced the worst individuals of the population. This algorithm was compared with other approaches in [93] and [123].

4.2.3 Replacing strategies

For explicit memory schemes, since the memory space is limited, it is important to decide how the memory must be updated when its capacity is full. A general strategy is to select one individual from the memory and to replace it with the best individual from the population. Determining which individual must be replaced is made according to a chosen *replacing strategy.*
The previously described memory approaches used different replacing methods. For instance, Mori [58] decided which individual to replace based on its free energy and age. Trojanowski [100] replaced the oldest individual in memory. In general, the remaining approaches used the method called *similar* proposed by Branke [14]. Branke investigated and compared the following replacing schemes:

- **mindist**: this strategy analyzed the two individuals in memory with the minimum distance between them and replaced the worst with the best individual in the population. For example, suppose that individuals **i** and **j** were chosen:

  - if $fit_i < fit_j$, replaced the individual j by the current best

  - if $fit_i > fit_j$, replaced the individual i by the current best

- **midinst2**: this method was analogous to the *midist* scheme, but the individual from the memory selected to be replaced was compared with the new individual to store. Again, suppose that individuals **i** and **j** were chosen:

  - if $fit_j * \frac{d_{ij}}{d_{max}} \leq fit_{newind}$, replaced the individual j by the current best
  - otherwise, replaced the individual i by the current best

    where d_{ij} was the distance between the individuals i and j and d_{max} was the maximal possible distance between the individuals i and j.

- **variance**: replaced the individual i that, when deleted, preserved the maximum variance $V(i)$ in memory. The variance was calculated using the equation:

$$V(i) = \sum_{j=1}^{m} \sum_{k \in M \setminus \{i\}} (x_{jk} - \bar{x}_j)^2$$

  where m was the chromosome length; M was the memory; x_{jk} was the allele j of individual k and $\bar{x}_j$ was the average allele value at position j of all memory individuals.

- **similar**: in this method, the current best individual of the population replaced the most similar individual stored in memory as long as it was a better solution. The similarity measure depended on the used representation. For binary encodings, the similarity between to individuals was measured using the Hamming distance.

Branke studied these replacing strategies and concluded that the ***similar*** method was the most efficient.

No further investigations concerning this issue have been proposed and the most *similar* strategy introduced by Branke became the most popular method used in memory-based EAs.

4.3 Multi-populations

In this type of EAs, the main population is divided into several sub-populations that track different peaks of the landscape. The different sub-populations are placed in different regions of the search space, preserving diversity and allowing the EA to readapt when a change occurs.

The Shifting Balance Genetic Algorithm, suggested in [65, 111], used a core population and a number of smaller colony populations. The core population was dedicated to track the current optimum and the smaller populations were responsible for searching in different areas of the search space. If a colony population became too close to the core population, the parents used to build the next population were selected based on their distance to the core population. The colony was then forced to move towards a distant area. From time to time, individuals from the colonies were sent to the core population maintaining its diversity.

The Multinational Genetic Algorithm (MGA), introduced by Ursem [101], used multiple sub-populations called *nations*. Nations were used to group individuals using a method called *hill-valley detection procedure*. This method, given two points in the search space, computed the fitness of a number of random sample points on the line between these two points. A valley was found if the fitness in a sample point was lower than the fitness of both end points. The creation of a nation and the migration of individuals between nations followed a set of rules: in every generation the algorithm compared each individual with the policy of its nation. A policy corresponded to a single point representing the peak of the nation is approaching. If a valley was detected, the individual migrated to a different nation, since it was moving away from the peak where the other individuals of these nations were located. The destination nation was selected comparing the individual to the policy of each of the other nations. If no nation fitted the individual's characteristics, then that individual found a new peak and a new nation was created. Two nations were merged if they were close to the same peak.

The Self-Organizing Scouts (SOS), proposed by Branke [14,15], used the concept of Forking Genetic Algorithms (FGAs) within the context of dynamic optimization problems. The FGA was based on the idea of dividing the search space into several parts, each exclusively explored by one of several sub-populations. A parent population continuously searched for new peaks, while a number of child populations exploited previously detected promising areas. Whenever the parent population had converged to one region, the FGA separated this region from the parent population's search space and assigned a child population to it for further exploitation. If the maximum allowed number of forking populations was reached, the oldest forking population was deleted. The use of the FGA in the context of dynamic problems was called *Self-Organizing Scouts*, because the individuals operated as scouts that explored different regions of the search space. The algorithm started a single population searching through the entire search space. At regular intervals, this population (parent population) was analyzed and, depending on the distance of the individuals, a child population may be formed by splitting off from the parent population. The parent population continued to search in the remaining search space and other sub-populations could be created by forking.

The Adaptive Island Model (AIM), proposed by Younes et al. [128], used an EA with a certain number of sub-populations (islands) of the same size. The number of islands was decided by two measures: the island diversity and the population diversity. Both measures of diversity were based on the genotypic distance between individuals. Each island was a small population that evolved under the control of its own diversity independent from other islands. In this way, an island played the role of a niche, as it consisted of individuals that were close to each other. After a pre-determined number of generations, the best individual from one island migrated to another, transferring new genetic material to the destination island.

4.4 Anticipation and Prediction

Recently, several studies concerning anticipation in changing environments using EAs have been proposed. The main goal of these approaches was to estimate the likelihood of particular future situations and to decide what the algorithm should do in the present situation. Since information about the future typically was not available, it was attained through learning from past situations.

Branke et al. [16] tried to understand how the decisions made at one stage influenced the problems encountered in the future. Future changes were anticipated by searching not only for good solutions but also for solutions that influenced the state of the problem in a positive way. These so-called flexible solutions were easily adjustable to changes in the environment. Studies on the tardiness job-shop problem, with jobs arriving non-deterministically over time, showed that avoiding early idle times increases flexibility, and thus the inclusion of an early idle time penalty as secondary objective into the scheduling algorithm, could significantly enhance the system's performance.

Stroud [96] used a Kalman-Extended Genetic Algorithm (KGA) in which a Kalman filter was applied to the fitness values associated with the population individuals. The goal of this Kalman filter was used to determine when to generate a new individual, when to re-evaluate an existing individual, and which one to re-evaluate. This KGA was applied to the problem of maintaining a network configuration with minimized message loss in which the nodes were mobile and the transmission over a link was stochastic. As the nodes moved, the optimal network changed, but the information contained within the population of solutions allowed efficient discovery of better-adapted solutions.

Van Hemert et al. [105] introduced an EA with a meta-learner to estimate, at time t, how the environment would be at time $t+\delta$. This approach used two populations, one that searched the current optimum and another that used the best individuals in the past to predict the future best value. The prediction was made based on observations from the past, using two types of predictors: a perfect predictor and a noisy predictor. In fact, they should not be called predictors. Concerning the former, the correct optimal value at the future time step **was given to the solver**, and the latter just provided the system noisy values as the optimal solution for the next step. The idea was tested with two benchmark problems: the knapsack problem and the Ǒsmera's function.

An integrated system combining prediction, optimization and adaptation techniques was proposed in [76] and applied to a real world application used to find the best distribution of cars of a particular model across the nation. The problem was complex and the implemented model addressed the issues of transportation, volume sensitivity effect, price depreciation, recent history, current inventory, risk factors and dynamic market changes. The system had three main modules: optimization, prediction and adaptation. The prediction module used information from the past to give prediction about the sale prices. In this paper, no information was given about which techniques were used to provide the predictions. The other two modules used the information provided by the prediction module and provided an answer to the actual problem. The optimization module used an EA to make the optimization task. Finally, the adaptation module received new information about the problem and adapted the parameters of the prediction module in order to decrease the prediction error. Later, in [120], this model was used in three different case studies.

Bosman [11], [12] proposed several approaches focused on the importance of using learning and anticipation in online dynamic optimization. In his works, he analyzed the influence of time-linkage present in problems such as scheduling and vehicle routing. The presence of time-linkage in this kind of problem can influence the overall performance of the system: if a decision is made just to optimize the score at a specific moment, it can negatively influence the results obtained in the future. Bosman's works proposed an algorithmic framework integrating evolutionary computation with machine learning and statistical learning techniques to estimate future situations. Predictions were made based on information collected from the past. The used predictor was a learning algorithm that approximated either the optimization function or several parameters.

Hatzakis et al. [38] used prediction techniques to forecast the location of the Pareto front in multi objective problems. This approach stored the location of previous solutions and used autoregressive models to predict the position of the new optimal solution in the next time step. The changes on the environment were known à priori in order to decide when to make the next prediction.
A similar technique for multiobjective optimization was investigated by Zhou et al. [129]. This approach explored prediction technics to re-initialize the population of an EA. The proposed method used information from the past to guide future search. Each individual in the population was tracked and its history was modeled through a time series model. Predictions about each individual's position at the next time step were made using a linear model. These predictions were used to re-initialize the population after a change was detected. Two strategies for population re-initialization were investigated. The first, predicted the new location of individuals from the location changes that had occurred in the past. Then, the current population was updated using new individuals generated based on that prediction. The second strategy consisted of perturbing the current population with a Gaussian noise, whose variance was estimated according to the previous changes.

Rossi et al. [72] compared different techniques to improve the search for tracking a moving optimum using the information provided by a predictor mechanism based on Kalman filters. The used predictor assumed that the changes in the environment were not random and could be learned, helping the EA to keep track of the current optimum.

4.5 Hybrid methods

Although some of the referenced work have been included in one of the previous sections, some of them used a combination of more than one method. Branke [14] stated that memory should always be used in combination with some mechanism to preserve the population's diversity. The combination of memory, either implicit or explicit, and diversity was used in [68], [53], [57, 58], [59], [103], [87], [118], [51].
The multi-populations approach described before also combined memory and methods for promoting the diversity.

Chapter 5

Improving Memory

When using memory-based EAs some questions can be asked concerning memory:

1. When and which individuals should be stored in memory?
2. Which individuals should be selected from the memory and introduced into the population when a change happens?
3. Which individuals should be replaced when the memory is full?
4. What is the best size for the memory?

In this chapter we focus in providing answers to the last two questions.

5.1 Memory-based Evolutionary Algorithms

Memory-based EAs work by storing explicitly or implicitly good solutions of the current population. In explicit memory approaches, besides the search population, an extra space - called memory - is used. The stored information can be reused later in new environments. When the environment changes, old solutions in the memory that are fit for the new environment are reactivated allowing the EA to readapt to the new environment. These approaches are suitable when the environments change cyclically: if good solutions from the past are memorized, they can be reused in similar future situations. After the retrieval of memory individuals, the EA readapts easier to the new environment.
To see if our methods for improving memory could be efficiently used in different types of memory-based EAs, we implemented three different algorithms: a direct-memory approach, the Memory-Enhanced Genetic Algorithm (**MEGA**), an immigrant memory algorithm, the Memory-Immigrants Genetic Algorithm (**MIGA**) and an associative memory method, the Associative-Memory Genetic Algorithm (**AMGA**).

In **MEGA** [114], the population and the memory were initialized at random. The memory was updated as follows: after being updated, the algorithm decided the next time the memory should be updated using a random integer between 5 and 10: if the memory was updated at generation t, the next update would occur at generation $t + rand(5, 10)$. In order to store the most relevant information to an environment in the memory, each time an environmental change was detected, the memory was also updated. When the memory was updated, the current best individual of the population (if the memory update was due to $t = t_m$) or the elite from the previous population (if the memory update was because a change detection) was stored replacing a random individual, if any was still present. If not, the best individual or the elite replaced the closest memory point if it was a better solution, according to the current environment or the previous environment. The memory was also used to detect environmental changes: a change was detected when at least one individual of the memory changed its fitness. At this moment, a new set of individuals was formed by merging the memory and the search population. Then, these individuals were evaluated in the context of the new environment, and the best p (population size) individuals were selected to become the new search population, which evolved through selection, crossover and mutation. Through this process, the memory remained unchanged. The best individual from the previous population was preserved and transferred to the next population replacing the worst individual (elitism of size 1). The pseudo code for **MEGA** is shown in Figure 5.1.

```
 Function $MEGA$
      $n$: global number of individuals
      $m$: memory size , $p$: population size
      $t$: current generation
      $t_m$: time to update memory
─────────────────────────────────────────────
$t = 0$
$t_m = rand(5, 10)$
$P(0)$ = initialize population randomly
$M(0)$ = initialize memory randomly
repeat
    evaluate memory $M(t)$ and population $P(t)$
    replace the worst in $P(t)$ by the best individual
        from $P(t-1)$
    if change is detected
         merge $P(t)$ and $M(t)$
         select the $p$ best individuals to form $P'(t)$
    else $P'(t) = P(t)$
    if $t = t_m$ or change detected
         if $t = t_m$
             $B_p(t)$ is the best individual from $P'(t)$
         if change detected
             $B_p(t)$ is the best individual from $P(t-1)$
         if still any random point stored in $M(t)$
             replace random individual with $B_p(t)$
         else
            if $t = t_m$
                 select memory individual $S_m(t)$
                 if $fitness(S_m(t))$ < $fitness(B_p(t))$
                       $S_m(t)$ is replaced by $B_p(t)$
            if change detected
                 select memory individual $S_m(t-1)$
                 if $fitness(S_m(t-1))$ < $fitness(B_p(t))$
                       $S_m(t-1)$ is replaced by $B_p(t)$
        $t_m = t + rand(5, 10)$
    $P''(t) = Selection(P'(t))$
    $Crossover(P''(t))$
    $Mutation(P''(t))$
    $P(t+1) = P''(t)$
    $t = t + 1$
until stop_condition
```

Figure 5.1: Pseudo code for the **MEGA** algorithm

MIGA [114] used the same memory updating mechanism as **MEGA**. However, the memory retrieval did not depend on the detection of environmental changes. This algorithm used the memory individuals to create immigrants. The memory was reevaluated every generation and the best individual from the memory was used to create immigrants that were introduced into the main population. The number of immigrants was a percentage (r_i) of the total number of individuals (n). The immigrants were created by mutating the best memory individual using an established mutation probability (p_i). Those immigrants were introduced into the population replacing the worst $r_i * n$ ones. The pseudo code for **MIGA** is shown in Figure 5.2.

AMGA was first introduced in [116] and was inspired in the Population-Based Incremental Learning (PBIL) algorithm proposed by Baluja [6]. The **AMGA** was a PBIL-like algorithm using an explicit memory scheme which stored the current best individual of the population as well as the environmental information given by a real-value probability vector. Each memory point consisted of a pair $< S, V >$ where S was the stored individual and V was the associated allele distribution vector. This vector gave the frequency of ones over the population at each gene locus. As in **MEGA**, the **AMGA** reevaluated the memory every generation and if a change was detected, the best individual in the memory $< B_m, V_m >$ was extracted and used to create a set of $\alpha * n$ new individuals using the vector V_m. Those new individuals were inserted into the population replacing the worst ones. The parameter $\alpha \in [0, 1]$ - called associative factor - determined the number of individuals created from the memory when a change occurred. Each new individual $I = \{g_1, g_2, ...g_l\}$ was created from $V_m = \{d_1, d_2, ..., d_l\}$ (l was the chromosome length) as follows:

$$g_i = \begin{cases} 1 & \text{if } rand(0.0, 1.0) < d_i \\ 0 & \text{otherwise} \end{cases}$$

The memory updating mechanism was analogous to the one used in **MEGA**. Every time the memory was updated, the pair $< B_p, V_p >$ was created and stored in the memory. B_p was the current best individual of the population and V_p was the allele distribution vector of the actual population. This pair replaced a random point in the memory if one still existed, or the closest point, otherwise. When a change was detected, a set of new individuals was created, replacing the worst ones of the population. The pseudo code for **AMGA** is shown in Figure 5.3.

```
Function $MIGA$
    $n$: global number of individuals
    $m$: memory size , $p$: population size
    $t$: current generation
    $t_m$: time to update memory
-----------------------------------------------------------

$t = 0$
$t_m = rand(5, 10)$
$P(0)$ = initialize population randomly
$M(0)$ = initialize memory randomly
repeat
    evaluate memory $M(t)$ and population $P(t)$
    replace the worst in $P(t)$ by the best from $P(t-1)$

    if $t = t_m$ or change detected
        if $t = t_m$: $B_p(t)$ is the best from $P(t)$
        if change detected: $B_p(t)$ is the best from $P(t-1)$
        if still any random point stored in $M(t)$
            replace random individual with $B_p(t)$
        else
            if $t = t_m$
                select memory individual $S_m(t)$
                if $fitness(S_m(t)) < fitness(B_p(t))$
                    $S_m(t)$ is replaced by $B_p(t)$
            if change detected
                select memory individual $S_m(t-1)$
                if $fitness(S_m(t-1)) < fitness(B_p(t))$
                    $S_m(t-1)$ is replaced by $B_p(t)$
        $t_m = t + rand(5, 10)$
    //create immigrants
    $B_m$ is the best individual of the $M(t)$
    $P_I(t)$ = create $r_i * n$ immigrants from $B_m$ ($p_i$)
    evaluate memory–based immigrants
    replace worst $r_i * n$ individuals in $P(t)$ with $P_I(t)$

    $P'(t) = Selection(P(t))$
    $Crossover(P'(t))$
    Mutation(P'(t))
    $P(t+1) = P'(t)$
    $t = t + 1$
until stop_condition
```

Figure 5.2: Pseudo code for the **MIGA** algorithm

```
 Function $AMGA$
     $n$: global number of individuals
     $m$: memory size , $p$: population size
     $t$: current generation
     $t_m$: time to update memory
----------------------------------------------------------------
$t = 0$
$t_m = rand(5, 10)$
P(0) = initialize population  randomly
M(0) = initialize memory randomly
        ($V$ starts with 0.5 at every loci)
repeat
     evaluate memory $M(t)$ and population $P(t)$
     replace the worst in $P(t)$ by the best individual
          from $P(t-1)$
     if change is detected
           $< B_m(t), V_m(t) >$ is the best individual in $M(t)$
           $P_I(t)$ = create $\alpha * n$ individuals from $V_m(t)$
           $P'(t)$ = replace worst $\alpha * n$ individuals
                        in $P(t)$ with $P_I(t)$
     else
          $P'(t) = P(t)$
     if $t = t_m$
           $t_m = t + rand(5, 10)$
           $B_p(t)$ is the best individual of the $P'(t)$
           $V_p(t)$ is the allele distribution vector of $P(t)$
           if still any random point stored in $M(t)$
               replace random individual with $< B_p(t), V_p(t) >$
           else
               select memory point $<S_m(t), V_m(t) >$
               if $fitness_{S_m(t)} < fitness_{B_p(t)}$
                    $< S_m(t), V_m(t) >$ is replaced by $< B_p(t), V_p(t) >$
     $P''(t) = Selection(P'(t))$
     $Crossover(P''(t))$
     $Mutation(P''(t))$
     $P(t+1) = P''(t)$
     $t = t + 1$
until stop_condition
```

Figure 5.3: Pseudo code for the **AMGA** algorithm

5.2 Replacing Strategies

Since memory has limited size, it is necessary to maximize its capacity and carefully select the individuals to be replaced when it is full. In chapter 4 we saw that this issue was already explored and different replacing schemes have been proposed [13], [8]. In this thesis we introduced three new mechanisms for replacing memory individuals when the maximum capacity has been reached.

5.2.1 Aging

The first replacing strategy based on age was called **age1**. Every individual in the memory had an attribute that indicated its age. When the memory was initialized, all individuals started with an age equal to zero. In every generation, the age of all the memory individuals increased one unit. Also, if a memory individual was selected to be inserted into the population when a change was detected, its age was increased by a specific value. Finally, there was a maximum value for the individual's age. If a memory individual reached this maximum, its age was reset to zero. When the memory was full and it was necessary to start replacing the memory individuals, the youngest individual was selected to be deleted and replaced by a new member (if this individual was better). The underlying idea was to replace an individual with less contribution to the EA's performance: one that was never selected to the population or that was in the memory for a long time and its age was set to zero. Figure 5.4 describes the pseudo code for this replacing strategy.

The second replacing strategy based on age was called **age2**. Every individual in memory had an attribute that indicated its age. When the memory was started, all individuals had age equal to zero. In every generation, the age of all the memory individuals was computed using a linear combination of its actual age and a contribution of its fitness. This contribution was set using a parameter called fit_rate. More formally, the age of an individual was calculated using equation 5.1.

$$age_i = age_i + 1 + fit\_rate * fitness_i \text{ , with i = 1, 2, ..., } m \qquad (5.1)$$

where m was the memory size.

In this strategy, the individuals' age was not set to zero, so the older individuals were not penalized. When the memory was full and it was necessary to start replacing memory individuals, again the youngest was selected to be deleted and replaced by a new member if this individual was better.
Figure 5.5 describes the pseudo code for this replacing strategy.

```
Function $Age_1$ $Replacing\ Strategy$
    $m$: memory size
    $C$: age's increment
    $MAX\_AGE$: maximum allowed age
    $i = 1, .., m$
------------------------------------------------------------
initialize memory randomly
$age_i = 0$
every generation: $age_i = age_i + 1$
if change detected
   $M_k$ is the selected memory individual
   Insert $M_k$ into population (replaces worst)
   $age_{M_k} = age_{M_k} + C$
if $age_i \geq MAX\_AGE$
   $age_i = 0$
if it is time to update memory
   $B_p$ is the best individual of the population
   if there are random individuals in memory
      select a random memory individual $M_{rnd}$
      memory individual $M_{rnd}$ is replaced by $B_p$
   else
      select memory individual $M_{sel}$ with the lowest age
      if $fitness(M_{sel}) <$ fitness($B_p$)
         memory individual $M_{sel}$ is replaced by $B_p$
```

Figure 5.4: Pseudo code of the **age1** replacing strategy

```
Function $Age_2$ $Replacing\ Strategy$
    $m$: memory size
    $fit\_rate$: fitness contribution. $fit\_rate \in (0, 1)$
    $i = 1, .., m$
------------------------------------------------------------
initialize memory randomly
$age_i = 0$
every generation: $age_i = age_i + 1 + fit\_rate \times fitness_i$
if it is time to update memory
    $B_p$ is the best individual of the population
    if there are random individuals in memory
        select a random memory individual $M_{rnd}$
        memory individual $M_{rnd}$ is replaced by $B_p$
    else
        select individual $M_{sel}$ with the lowest age
        if $fitness(M_{sel}) < fitness(B_p)$
            individual $M_{sel}$ is replaced by $B_p$
```

Figure 5.5: Pseudo code of the **age2** replacing strategy

5.2.2 Generational

This replacing strategy, denominated by **generational**, aimed to maximize the number of different individuals present in the memory by storing, if possible, an individual for each different environment already known. In order to have a different individual per environment, this strategy, when updating the memory, checked if any random individual still existed or if a solution was stored since the last change. In those cases, the best individual of the population was memorized: a random individual was always replaced; an individual from the same environment was replaced if it was worst than the current best. Finally, if no individual had been stored since the last change and there were no random individuals to replace, we selected the individual that was genetically similar to the current best and replaced it, if it was worse than the current best.
Figure 5.6 describes the pseudo code for this replacing strategy.

```
  Function *Generational Replacing Strategy*
      $m$: memory size
      $t$: current generation
      $i = 1, .., m$
---------------------------------------------
t = 0
initialize memory randomly
if it is time to update memory
    $B_p$ is the best individual of the population
    if there is a memory individual $M_i$
       of the same environment stored in memory
       if $fitness(M_i) < fitness(B_p)$
           memory individual $M_i$ is replaced by $B_p$
    else
       if there are random individuals in memory
           select a random memory individual $M_{rnd}$
           memory individual $M_{rnd}$ is replaced by $B_p$
       else
           select memory individual $M_{sel}$
            (the most similar with $B_p$)
           if $fitness(M_{sel}) < fitness(B_p)$
              memory individual $M_{sel}$ is replaced by $B_p$
```

Figure 5.6: Pseudo code of the **generational** replacing strategy

Example

In the following example, the memory individuals were randomly generated at the beginning and the environment changed every 20 generations, but this information was unknown to the algorithm. The time to update memory was decided at random.

Some considerations about the notation found in the next tables: the value '1' in the *rnd* column indicates that the individual is one of the initial random solutions; a '0' in this column indicates that the memory individual is not random. Initially, the attribute *gen* was set to '-1' and afterwards this value changed to the generation number where the solution was memorized. In the column *fitness*, a '-1' value indicates that the memory individual has not yet been evaluated. Table 5.1 represents the initial configuration of the memory.

Initial memory			
genotype	is rnd?	gen	fitness
0011000000	1	-1	-1
1110000000	1	-1	-1
0000000000	1	-1	-1
1111100000	1	-1	-1
0000011111	1	-1	-1
1010101010	1	-1	-1
1100110011	1	-1	-1

Table 5.1: Initial memory

Assume that the first memory update occurred at generation 6. At this point the best individual of the main population was stored and replaced one of the random memory individuals. The individual to be replaced was selected randomly (Table 5.2).

Memory at generation 6			
genotype	is rnd?	gen	fitness
0011000000	1	-1	2
1110011110	**0**	**6**	7
0000000000	1	-1	0
1111100000	1	-1	5
0000011111	1	-1	5
1010101010	1	-1	5
1100110011	1	-1	6

Table 5.2: Memory updated at generation 6

If the next memory update occurred at generation 15, and since the environment didn't change yet, we replaced the last memorized individual, if the current population best had a higher fitness (Table (5.3).
If the next memory update occurred at generation 25, for instance, and since the environment changed every 20 generations, we were in the presence of a new environment. In this situation, the current best was stored in the memory replacing one of the random individuals (see Table 5.4).
When no random individual were present in the memory and a new environment appeared since the last storage, the Branke's similar strategy was used to decide which individual to replace. The memory individual which was the most similar with the current best individual of the population was replaced (if it was a worse

Memory at generation 15			
genotype	is rnd?	gen	fitness
0011000000	1	-1	2
1111111110	**0**	**15**	9
0000000000	1	-1	0
1111100000	1	-1	5
0000011111	1	-1	5
1010101010	1	-1	5
1100110011	1	-1	6

Table 5.3: Memory updated at generation 15

Memory at generation 25			
genotype	is rnd?	gen	fitness
0011000000	1	-1	2
1111111110	0	15	9
0001111111	**0**	**25**	7
1111100000	1	-1	5
0000011111	1	-1	5
1010101010	1	-1	5
1100110011	1	-1	6

Table 5.4: Memory updated at generation 25

solution). For instance, suppose that at generation 150 the memory was updated and no random individuals were present in the memory (see Table 5.5).

Memory at generation 150 (before update)			
genotype	is rnd?	gen	fitness
1111100010	0	58	6
1111111110	0	15	9
1001111111	0	36	8
1111100111	0	75	8
1100111000	0	116	5
1111111000	0	139	7
1111101111	0	97	9

Table 5.5: Memory at generation 150

Assume that the current best of the population had a fitness equal to 10 and its genotype was 1111111111. One of the most similar individuals from the memory was selected and since the current best had better fitness, this individual was stored replacing the most similar (Table 5.6).
This replacing strategy maximized the memory's diversity and minimized the number of redundant individuals, improving the memory's usage.

Memory at generation 150	(after update)		
genotype	is rnd?	gen	fitness
1111100010	0	58	6
1111111111	0	**150**	**10**
1001111111	0	36	8
1111100111	0	75	8
1100111000	0	116	5
1111111000	0	139	7
1111101111	0	97	9

Table 5.6: Memory updated at generation 150

5.3 Population and Memory Sizes

In the standard EA, the parameter settings are usually decided at the beginning and remain constant during the entire process. During the last years, this approach changed and extensive research in the field of setting the parameters of the EAs was made. Those investigations focused in two main issues: the run-time adjustment of probabilities of the genetic operators and of the population size. The first topic was studied by several authors: [41], [34], [3], [4], [22], [23], [24]. Concerning the population size, usually it is considered an unchanging parameter, which value is constant during the run. Choosing this parameter off-line can be problematic: if it is too small the EA may not be able to find good solutions; if it is too large, there is a high computational overhead. Although finding an appropriate population size is a difficult task, several adaptive population sizing methods have been suggested to be used in EAs solving stationary problems: [2], [5], [30], [52].

When the EAs are used to deal with **dynamic environments**, as stated before, some modifications have to be introduced. In spite of these modifications, the global functioning of the EAs designed to deal with dynamic applications inherited most of the characteristics of the EAs used for static domains. So, typically, the EA uses constant values for the crossover and mutation probabilities and a constant population size. In the specific domain of this thesis, when an explicit memory is used, its size is also set at the beginning and is usually a small percentage of the global number of individuals. Not much attention has been devoted to the study of the influence of population and memory size in the performance of the EA for dynamic environments. It is also a fact that, in approaches using memory, no reasons are referred to support the choice of a certain value for the population/memory sizes. However, if we look to natural systems that inspire EAs, the number of individuals of a certain species changes over time and tends to become stable around appropriate values, according to environmental characteristics or natural resources [33].
Little research has been made about this issue: Schönemann [78], [79] studied the impact of population size in the context of Evolutionary Strategies for dynamic environments, and concluded that the choice of the population size could be a determinant factor in certain classes of problems. Recently, Richter et

al. [70] proposed a memory-based abstraction method using a grid to memorize useful information and the results obtained suggested that an optimal grid size depended on the type of dynamics. The authors claimed that the use of an adaptive grid size would increase the performance of the abstraction memory, indicating this issue as future research. As far as we know, no more research has been done concerning this subject.
Looking to the different approaches using explicit memory schemes for EAs coping with dynamic environments it is evident that the memory size (m) is always set as a small percentage of the global number of individuals (n). Next, we present the choices for population and memory sizes made in the most relevant works using explicit memory-based EAs for dynamic environments:

- Mori et al. [58] presented a memory-based thermodynamical GA (TBGA) which used a population of 46 individuals and a memory of size 8;
- The enhanced memory EA proposed by Branke [13] used a population of 90 individuals and a memory with size 10;
- Bendtsen's approach [8] used a population with 100 individuals and a memory of 10;
- Karaman et al. [43] studies used population of 50 individuals, while the memory size was 10;
- Simões et al. [87] proposed a GA inspired in the natural immune system. The algorithm used a search population of size 100 and a memory of size 20;
- Other memory-based approaches proposed by Simões et al. used population with 110 chromosomes and memory of size 10 [89].
- The approach suggested by Trojanowski et al. [100] used a population with 100 individuals and a memory size equal to 20;
- EAs using associative memory schemes studied in [126], [116] used population of 110 individuals and memory of size 10;
- Different memory approaches proposed by Yang used population of size 90 and memory of 10 individuals [114], [115], [117], [118], [121], [123], [127];
- The memory/search EAs proposed in [123] used two populations of variable size and a memory with constant size. The sum of the two populations was always 108 and the memory used 12 individuals.
- Liu's approach [51] used a global number of individuals equal to 110 and memory of size 10.

Table 5.7 gives a synthesis of the relation between the global number of individuals (n) and memory sizes (m) and the corresponding percentage of memory size relative to n. On average, the dimension of memory was chosen to be 10.5% of the global number of individuals. This corresponds to memory proportions

Algorithm	Ref.	n	m	$\frac{m}{n}$
$TBGA$	[58]	$n = 54$	$m = 8$	15%
Mem/Search EA	[13]	$n = 100$	$m = 10$	10%
Mem/Search2 EA	[13]	$n = 100$	$m = 10$	10%
$AMIGA$	[100]	$n = 120$	$m = 20$	17%
$DMEA$	[8]	$n = 110$	$m = 10$	9%
$ISGA$	[87]	$n = 120$	$m = 20$	17%
$MIEA$	[43]	$n = 60$	$m = 10$	17%
$MPBIL1$	[116]	$n = 120$	$m = 10$	8%
$MPBIL2$	[116]	$n = 120$	$m = 10$	8%
$MEGA$ $MIGA$ $MRIGA$	[114]	$n = 100$	$m = 10$	10%
$MUMDA$ $MUMDA_i$ $MEGAi$	[115]	$n = 100$	$m = 10$	10%
$AMGA$	[117]	$n = 100$	$m = 10$	10%
$DAMGA$	[117]	$n = 100$	$m = 10$	10%
$ISGAs$	[118]	$n = 100$	$m = 10$	10%
$MIGA$	[89]	$n = 120$	$m = 10$	8%
$MEGA$	[89]	$n = 120$	$m = 10$	8%
$DMGA$ $AMGA$ $HMGA$	[121]	$n = 100$	$m = 10$	10%
$MIGA$ $MRIGA$	[123]	$n = 100$	$m = 10$	10%
$MPBIL$ $MEGA$ $MSGA$	[123]	$n = 120$	$m = 12$	10%
$MPBILi$ $ISGA$	[127]	$n = 110$	$m = 10$	9%
$PISGA$	[51]	$n = 110$	$m = 10$	9%
			Average	***10.5%***

Table 5.7: Summary of population and memory sizes

between 9% and 20% of n, with 10% being the most selected. As we can see, choosing a constant value for population and memory sizes was widely used in memory-based EAs. Memory was always used with a smaller dimension, but no justification was given for that choice.

In this thesis we are concerned in obtaining more insight about this topic. Does size really matter? Is it correct to choose a smaller memory size and keep this value constant during the entire evolutionary process? Can the EA evolve the best memory/population size according to the characteristics of the environment or the specifications of the problem? To find answers to those questions we divided the study in two parts. First, we used different memory-based EAs and made an exhaustive empirical study, keeping constant the population and memory size, but changing it in different intervals. The next section explains this experimentation. Second, we proposed a new EA that dynamically changed the population and memory sizes. The global number of individuals could not surpass a given maximum, but the algorithm changed the size of population and memory in order to obtain higher performances. This algorithm, called Variable-size Memory Evolutionary Algorithm (**VMEA**), is explained in section 5.3.2.

5.3.1 Size matters?

The first part of our investigation consisted of using the three memory-based EAs, described earlier, to solve the same set of problems using population and memory with different sizes. The global number of individuals was chosen to ensure the same number of function evaluations. The distribution of those individuals in memory and population was made with different proportions.

The three memory-based algorithms were run using the same parameter settings except for the population and memory sizes. The global number of individuals was always $n = 100$, the memory size m was used as a different percentage of n as follows:

$$m = K\% \times n \text{ , with } K = 10, 20, 30, ..., 90$$

All the variations of the algorithms were run solving four different dynamic optimization problems with different characteristics.
All the details about the parameters and problems used is given in chapter 8 and the results obtained are reported in chapter 9.

5.3.2 Variable-size Memory Evolutionary Algorithm

The second step in understanding the importance of population and memory sizes was dedicated to the development of an algorithm that could find the best choice for the memory and the population sizes during the run. This new algorithm used a memory to store good solutions and a main population that was responsible for finding the current optimum. The innovation introduced in that algorithm was that the population and the memory sizes could change during the run. The sizes were set to initial values, those values could change during the run, but the global number of individuals was always the same. This algorithm used a population that searched for the optimum and evolved, as usual, through selection, crossover and mutation. A memory population was responsible for storing good individuals of the evolved population at several moments of the search process. The basic idea of **VMEA** was to use the limited resources (total number of individuals) in a flexible way.
In order to get room to add new individuals into memory or into population two cleaning processes were performed when the global number of individuals was attained. These mechanisms looked for individuals of equal genotype and if they were found they were removed from the population ($CleanPopulation$) or from the memory ($CleanMemory$). These mechanisms were executed, if necessary, when a change was detected or when the memory was updated.
The memory was evaluated every generation and a change was detected if at

least one individual in the memory changed its fitness. If an environmental modification was detected, the best memory individual for the new environment was introduced into the population. If there was no room for adding this individual into the population, after the cleaning of repeated chromosomes, the best individual in memory replaced the worst one of the population.
The memory was updated from time to time and if the established limit was not reached, the best individual of the current population was stored in the memory, increasing its size. If there was no room to keep this new solution, then the best individual of the current population was introduced replacing a memory individual chosen accordingly to the replacing scheme. The moment chosen to update memory (t_m) was calculated using a random pattern. The first update time was computed by $t_m = rand(5, 10)$ and the next updating times used the same pattern: $t_m = t + rand(5, 10)$, where t was the actual generation and $rand(5, 10)$ was a random generator of numbers between 5 and 10. The memory was also updated when a change was detected in the environment. The individual to be stored in memory, in the first situation, was the best individual of the current population; in the second case, the memorized solution was the best individual from the population before the change had occurred.
When the memory was updated, first, the randomly initialized individuals were replaced and the memory size was kept constant. If there were no more random individuals, and if after adding this new individual, the global number of individuals was beyond the maximum, the best individual was introduced into memory and its size was increased accordingly. If the maximum value was reached, then the cleaning process was executed to remove repeated chromosomes from the population. If this cleaning was successful, the best individual of the population was added to the memory, the population size was decreased by one and the memory size was increased by one. If the clean of redundant solutions in the population failed, the cleaning process was executed to the memory. If a repeated individual was removed from the memory, the best individual was added to the memory and the sizes were kept constant. If all this failed, i.e., if there was no space for a new individual, the individual was stored replacing one of the existing. The replacement occurred if the selected individual had lower fitness than the current best. The individual to be replaced was selected using the replacing strategy that was being used.
The first **VMEA** was proposed in [91] and the process of cleaning the individuals with equal genotype was executed only in the memory. In [80] the cleaning of the population was incorporated. The cleaning processes removed a maximum of one memory individual or two individuals from the population.
This final version [93] was used in the experiments carried out in this thesis. The pseudo code for **VMEA** is in Figure 5.7. This algorithm was compared with other memory-based schemes using the standard dimensions for population and memory and the results are reported in chapter 9.

```
Function $Variable-size\ Memory\ Evolutionary\ Algorithm$
$p$: population size , $m$: memory size
$n = p + m$ , $n$ is updated during the run
$t_m$: memory update time

set initial values for $p$ and $m$
$t = 0$ , $t_m = rand(5, 10)$
random initialize $M(0)$ and $P(0)$
repeat
    evaluate memory $M(t)$and population $P(t)$
    replace the worst of $P(t)$ by the best of $P(t-1)$
    if change is detected
        $B_m$ is the best individual from $M(t)$
        if $n + 1 \leq n$ then
            Add $B_m$ to $P(t)$ , $p = p + 1$
        else if CleanPopulation is successful
                Add $B_m$ to $P(t)$
             else if CleanMemory is successful
                    Add $B_m$ to $P(t)$
                    $p = p + 1$ , $m = m - 1$
                  else
                    $B_m$ replaces the worst of $P(t)$
    if $t = t_m$ or change detected
        if $t = t_m$: $B_p$ is the best of $P(t)$
        if change detected: $B_p$ is the best of $P(t-1)$
        if still any random point stored in $M(t)$
            replace random memory individual by $B_p$
        else
            if any memory point of same cycle
                replace it
            else
                if $(n + 1) \leq n$
                    Add $B_p$ to $M(t)$ , $m = m + 1$
                else if CleanPopulation is successful
                        Add $B_p$ to $M(t)$
                        $p = p - 1$ , $m = m + 1$
                    else if CleanMemory is successful
                            Add $B_p$ to $M(t)$
                         else
                            select $M_{sel}$ from $M(t)$
                            if $fitness(M_{sel}) < fitness(B_p)$
                                Replace $M_{sel}$ by $B_p$

        $t_m = t + rand(5, 10)$
    $P'(t) = Selection(P(t))$ , $Crossover(P'(t))$ , $Mutation(P'(t))$
    $P(t + 1) = P'(t)$
    $t = t + 1$
until $stop\_condition$
```

Figure 5.7: Pseudo code of the Variable-size Memory Evolutionary Algorithm

Chapter 6

Promoting Diversity

The use of mechanisms to promote diversity is widely used in EAs dealing with dynamic environments. We know that the premature convergence is an important drawback in traditional EAs used in non-stationary problems, so it makes sense to hypothesize that the use of mechanisms to promote the population's diversity can help the EAs in this type of optimization problems. As we saw in Chapter 4, different approaches can be used to reach this goal. The most used are (1) the increasing of the mutation rate after a change is detected or (2) strategies based on immigrants, which introduce new individuals into the population before or after the change has occurred.

In our work we looked for inspiration in natural systems and proposed two different recombination operators that, used as crossover substitutes, allow the promotion of the population's diversity at different levels. We investigated two operators, called transformation and conjugation, and proposed the corresponding computational approaches of mechanisms for rearranging the genetic material present in living organisms such as viruses and bacteria.

6.1 New genetic operators

The genetic operators are essential to the EA's performance. Much work has been done in this field, concerning the definition and improvement of traditional genetic operators. Particularly, the crossover operator was assumed as the most important operator for the effectiveness of the EA. Bio-inspired crossover has been used in three main variants (one point, two point and uniform), and has become the standard genetic operator used in the EA to mix the genetic material of the individuals of the population. Nevertheless, if we look at nature we see a wide variety of processes responsible for creating genetic diversity among the individuals: transposition, transduction, translocation, transformation or conjugation, to name a few. Computational approaches of some of these operators were already studied and used with success in EAs solving static problems (transposition, translocation, conjugation). In this thesis we explored two of those genetic operators and used them as the main genetic operator (replacing the standard crossover) in EA dealing with dynamic optimization problems. Our goal was to see if they provided more diversity in the population and if this was always advantageous for the EA dealing with dynamic optimization problems. The analyzed genetic operators were conjugation and transformation and will be explained in the next sections.

6.1.1 Transformation

Biological Transformation

Transformation is the transfer of genetic material between organisms by means of extracellular pieces of DNA. These strains of DNA, or gene segments, are extracted from the environment and added to recipient cells [33]. Bacterial transformation was first observed by Frederick Griffith in 1928, an English bacteriologist searching for a vaccine against bacterial pneumonia. He discovered that a non-virulent strain of *Streptococcus pneumoniae* could be transformed into a virulent one by exposure to strains of virulent S. pneumoniae *Streptococcus pneumoniae* that were killed with heat. Later, in 1944 Oswald Avery, Colin MacLeod, and Maclyn McCarty showed that the transforming factor was genetic, because of the gene transfer in *Streptococcus pneumoniae.* This process of uptake and incorporation of DNA by bacteria was called **transformation**.
After the transfer of small pieces of extra cellular DNA between organisms, there are two possibilities, failure or success, known technically as restriction and recombination. Restriction is the destruction of the incoming foreign DNA, since those bacteria assume that foreign DNA is more likely to come from an enemy, such as a virus. In this case, transformation fails. Recombination is the physical incorporation of some of the incoming DNA into the bacterial chromosome. If this happens, genes from the assimilated segment replace some of the host cell's genetic information and bacteria are permanently transformed. Once integrated in the chromosome, the DNA segment is able to survive. Organisms that have the capability of absorbing foreign DNA are called competent.
There are two types of bacterial transformation:

- *natural transformation*: bacteria naturally have the capacity to take up foreign DNA and become genetically transformed by it;
- *engineered transformation*: bacteria are altered through genetic engineering to make them competent, allowing them to be genetically transformed by the foreign DNA ;

Bacterial transformation is widely used in genetic engineering, where biologists manipulate genetic material: fragments of DNA are isolated, cut into discrete pieces and rejoined to create novel genes and other genetic constructs. This technology allows scientists to study the activity of genes in order to understand their function. For instance, one of the applications of this technology is the potential to treat genetic diseases, such as cancers, by gene replacement. The genetic manipulations described above require large quantities of DNA. One of the easiest ways to get large amounts of DNA is to use *engineered transformation*: place the desired DNA into bacteria, grow the bacteria, then harvest the bacteria, and isolate the DNA [73].

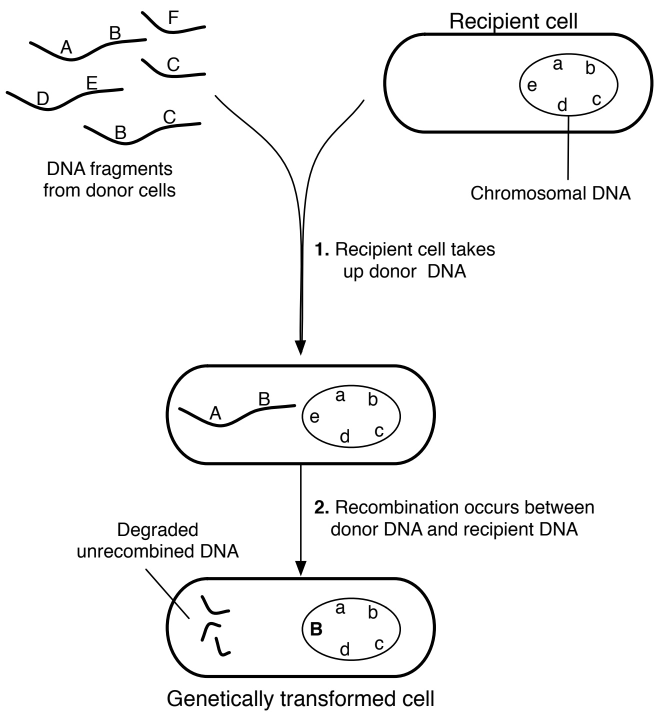


Figure 6.1: Biological transformation

Computational Transformation

To mimic this biological mechanism, at the beginning of the process, a pool of segments of different sizes was created. Each segment consisted of a binary string and could be selected to be incorporated into the individuals. Since this operator replaced crossover it was used in an analogous manner: first, the individuals to be transformed were selected using the chosen selection method. Then, transformation was applied to each one of those individual with a fixed probability. Besides changing the individuals of the population, the pool of segments was also updated: every generation a percentage of the existing segments was selected and randomly modified using the genetic information of the individuals of the population. This percentage was controlled by the parameter α_t. The pseudo code of a memory-based EA using this mechanism is described in Figure 6.2.

```
Function $Memory - based\ EA\ with\ Transformation$
     $n$: global number of individuals
     $p$: population size
     $m$: memory size
     $s$: gene segment pool size
     $t$: current generation
     $t_m$: time to update memory
--------------------------------------------------------------

$t = 0$
$t_m = rand(5, 10)$
$P(0)$ = initialize population randomly
$M(0)$ = initialize memory randomly
$G(0)$ = initialize gene segment pool randomly

repeat
    evaluate memory $M(t)$
    evaluate population $P(t)$
    replace the worst in $P(t)$ by the best individual
        from $P(t-1)$
    if change is detected
         $P'(t)$ = retrieve information from $M(t)$
    else $P'(t) = P(t)$
    if $t = t_m$ or change detected
         update memory
         $t_m = t + rand(5, 10)$

    $P''(t) = Selection(P'(t))$
    $Transformation(P''(t), G(t))$
    $Mutation(P''(t))$
    $G(t+1) = Update\ gene\ segment\ pool\ (P'(t))$
    $P(t+1) = P''(t)$
    $t = t + 1$
until stop_condition
```

Figure 6.2: Pseudo code for the Memory-based EA with Transformation

Each individual of the mating pool was transformed, with a certain probability, following the steps described in Figure 6.3.

```
Function Transformation
    $p_t$: transformation probability
---------------------------------------------
For each individual of the mating pool
    if $rand() \leq p_t$
        Select random segment
        Select random transformation point
        Replace individual genes with segment genes
```

Figure 6.3: Computational Transformation

The main aspects to consider in the implementation of transformation were the origin of the gene segments that transformed each individual, how the process of transformation occurred, and the updating of the gene segments pool.
The EA started with an initial population of p individuals and an initial pool of p_t gene segments, both created at random. In each generation, the p individuals were selected to be transformed using the gene segments of the gene segment pool. After that, the gene segment pool was changed: α_t of the current segments were replaced by new ones created from the individuals of the old population, the remaining segments were created randomly (see Figure 6.4).

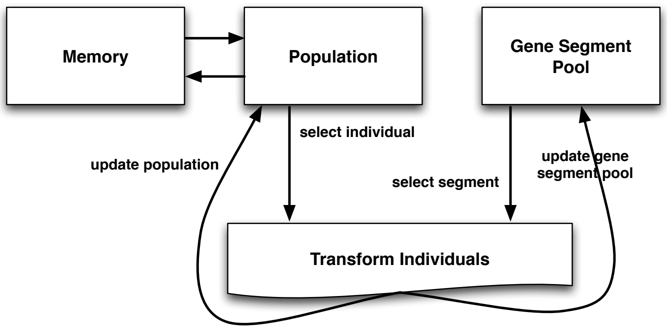


Figure 6.4: Computational transformation

The EA followed the traditional steps and a selection method was used to choose which individuals to transform. These selected individuals were grouped in a mating pool and the transformation mechanism was applied to each individual with a fixed probability. Transformation was seen as a form of asexual reproduction, since there was no exchange of genetic material between the individuals of the population. Each individual generated a new one through the process of transformation.
The transformation of each selected individual followed these steps:

1. randomly selection of a segment from the segment pool
2. randomly choice of a point of transformation in the selected individual
3. incorporation of the segment in the genome of the individual

The incorporation of the segment into the selected individual was made by replacing the genes after the transformation point. The chromosome was seen as a circle, in order to keep the chromosome length constant. This corresponded to the biological process where the gene segments, when integrated in the recipient's cell DNA, replaced some genes in its chromosome. Figure 6.5 illustrates the process of transforming an individual.

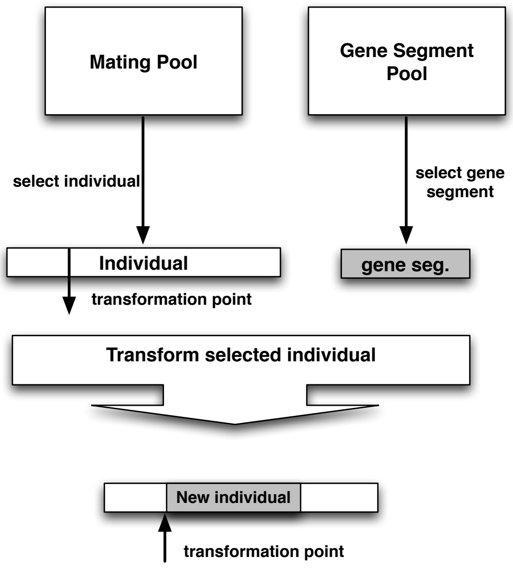


Figure 6.5: Transforming an individual

The segments used to change the individual proceed, mostly, from the individuals existing in the previous generation. In the used experimental setup, the segment pool was changed in every generation. The modifications were made replacing a percentage α_t of the segments with new ones, created from the individuals from the old population. The remaining 1-α_t were created randomly. The size of each gene segment was also chosen at random. The update of the gene pool was made according to the steps indicated in Figure 6.6 .

```
Function Update gene segment pool
    $\alpha_t$: the percentage of segments created
                from previous population
    $p$: the population size
    $l$: the chromosome length
    $sl_i$: the length of segment $i$
    $P$: previous population
    $G_i$: the $i^{th}$ segment
-----------------------------------------------------------
$i = 0$
repeat
  select a random individual from $P$
  select two random points in the individual -> $p_1$, $p_2$
  $G_i$ = genetic material contained between $p_1$ and $p_2$
  $i = i + 1$
until (i == $\alpha_t \times n$)
repeat
  $sl_i$ = random(1, l)
  $G_i$ = randomly generated bits of size $sl_i$
  $i = i + 1$
until (i == $p$)
```

Figure 6.6: Updating the gene segment pool

6.1.2 Biological Conjugation

Bacterial conjugation was discovered in 1946 by Joshua Lederberg and Edward Tatum and consists of the transfer of genetic material between bacteria through cell-to-cell contact. To make conjugation possible, two bacterial cells must come together and a cytoplasmic bridge called *pilus* is built. This contact is temporary and allows to transfer genetic material via the plasmid from the donor cell to the recipient cell. The plasmid consists of a linear or circular double-stranded DNA that is capable of replicating independently of the chromosomal DNA.
As the donor replicates its chromosome, the copy is injected into the recipient. At any time that the donor and recipient become separated, the transfer of genes stops. The genes that successfully make the trip replace their equivalents in the recipient's chromosome. This mechanism only occurs between cells of opposite mating types. The donor (or "male") carries a fertility factor ($F+$), which doesn't exist in the recipient cell, the "female" ($F-$). The factor F is a set of genes originally acquired from a plasmid and integrated into the bacterial chromosome. A donor cell can become later a recipient one. The cells produced by conjugation are always $F+$. Nevertheless, the population of bacteria never become 100% $F+$ because conjugation is very time-consuming. Thus, an $F-$ cell can undergo one or more fissions while an $F+$ cell engages in one conjugation. Hence, the population of $F+$ cells can actually decline with conjugation. Another factor that limits the $F+$ cells is the *pilus* that only $F+$

cells can build. This bridge provides an attachment point for certain viruses, which thus kill only the $F+$ cells. Figure 6.7 illustrates biological conjugation. In biology, bacterial conjugation is a beneficial process to bacteria since it allows them to acquire a gene that confers survival or a novel characteristic which enables them to thrive in harmful conditions or to utilize a new metabolite. It is through this process that resistance to antibiotics can be transferred from one bacterial cell to another. 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 [33].

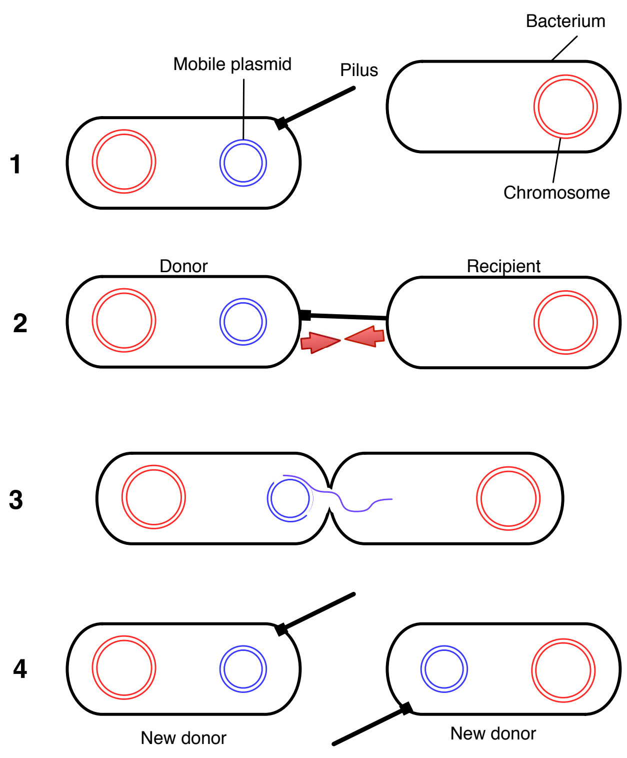


Figure 6.7: Biological conjugation

6.1.3 Computational Conjugation

Computational conjugation tried to mimic the biological mechanism and was introduced independently by Smith [94], [95] and Harvey [37].
Smith 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 [37] investigated a tournament-based conjugation: two parents were selected at random, and the winner of the tournament (fittest individual) became the donor and the loser the recipient of the genetic material. That way, the

conjugation operator could be applied repeatedly by different donors to a single recipient. Both authors used this operator as a substitute for the crossover operator in a GA to solve different stationary problems.
In this thesis we proposed a different version of computational conjugation where the donor and the recipient cells were not chosen at random but selected according to their fitness. This operator was integrated in the memory-based EA, replacing crossover and was used to create the offspring from the selected parents.
The mating pool was created using the appropriate selection method. The individuals were selected to be donors or recipients according to their current fitness: the $\frac{p}{2}$ best individuals became the 'donors' while the remaining became the 'recipients' (p was the current size of the population). Then, using a fixed probability, the i_{th} donor transferred part of its genetic material to the i_{th} recipient (i=1, ..., $\frac{p}{2}$). Following that, all offspring created by this process was mutated and joined with the donor individuals becoming the next population of size p.
The pseudo code of an EA using this mechanism is described in Figure 6.8.

```
  Function $Memory-based\ EA\ with\ Conjugation$
      $n$: global number of individuals
      $p$: population size
      $m$: memory size
      $t$: current generation
      $t_m$: time to update memory

t = 0
$t_m = rand(5,10)$
$P(0)$ = initialize population randomly
$M(0)$ = initialize memory randomly
repeat
    evaluate memory $M(t)$
    evaluate population $P(t)$
    replace the worst in $P(t)$ by the best individual
        from $P(t-1)$
    **if** change is detected
        $P'(t)$ = retrieve information from $M(t)$
    **else** $P'(t) = P(t)$
    **if** $t = t_m$ **or** change detected
        update memory
        $t_m = t + rand(5,10)$

    $P''(t) = Selection(P'(t))$
    $Conjugation(P''(t))$
    $Mutation(P''(t))$
    $P(t+1) = P''(t)$
    $t = t + 1$
until stop_condition
```

Figure 6.8: Pseudo code for the Memory-based EA with Conjugation

Conjugation was applied to $\frac{p}{2}$ pairs of individuals following these steps (see Figure 6.9):

1. select i^{th} donor
2. select i^{th} recipient
3. select two random points
4. select the donor's genes contained between the two points
5. replace the corresponding recipient's genes
6. the donor remains unchanged

$i = 1, 2, ..., \frac{p}{2}$

```
Function Conjugation
  p: the population size
  p_c: conjugation probability
  P: current population

  Select p parents from P
  Select the p/2 best individuals (donors)
  Select the p/2 worst individuals (recipients)
  for i = 1 to p/2
      if rand() ≤ p_c
          select i^th donor
          select i^th recipient
          select two different random points p_1 and p_2
          insert donor's genes into recipient
```

Figure 6.9: Pseudo code for conjugation

Figure 6.10 shows how conjugation was applied to one pair of individuals of the mating pool.

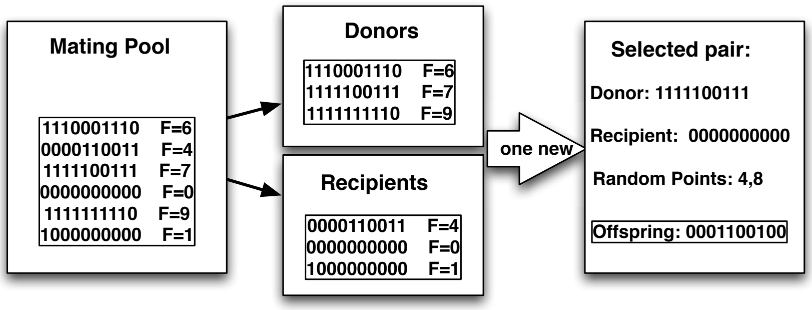


Figure 6.10: Computational conjugation

Transformation and conjugation were used in different memory-based EAs replacing crossover. The EA's performance and its adaptability in different types of dynamic problems was analyzed and compared with the EA using crossover. The population's diversity was also measured, in order to investigate its influence on the algorithms' performance. The results obtained and the main conclusions are reported in Chapter 10.

Chapter 7

Prediction

This chapter describes the prediction mechanisms incorporated in the memory-based EA. The proposed mechanisms provide predictions for two situations: when the next change will occur and how the environment will change. For the first predictor, to estimate **when** the next change will take place, we propose two different approaches, one using linear regression, another using nonlinear regression. The second predictor is based on Markov chains and is used to estimate **how** the next environments will look like. The combined application of these two predictors can significantly increase the EA's performance. Knowing the time when the next change will happen and how the environment will change, it is possible to retrieve useful information from the memory and introduce it into the population **before** the alterations in the environment happen.

7.1 Predicting *when*

Usually, the memory-based EAs for dynamic environments detect the changes when they occur. **After** the change is detected the information is retrieved from the memory and inserted into the population. In certain types of dynamic environments some repeated behavior can be observed and it is possible to make predictions about when the next change will happen. For instance, if the environment changes periodically after a fixed number of generations, the generation when the next change will occur can be correctly predicted. Even in non periodic environments, if some repeated pattern is present, prediction methods can be successfully applied. The next sections describe the contributions given in this thesis that allow the EA to accurately predict the generation **when** the next change will be observed in the environment.

7.1.1 Linear Regression Predictor

Basics of Linear Regression

Simple linear regression studies the relationship between a response variable y and a single explanatory variable x. This statistical method assumes that for each value of x, the observed values of y are normally distributed around a mean that depends on x. These means are usually denoted by μ_y. In general the means μ_y can change according to any sort of pattern as x changes. In simple linear regression it is assumed that they all lie on a line when plotted against x. The equation of that line is:

$$\mu_y = \beta_0 + \beta_1 \times x \tag{7.1}$$

with intercept β_0 and slope β_1. This is the linear regression line and describes how the mean response changes with x. The observed y values will vary around the mean and it's assumed that this variation, measured by the standard deviation, is the same for all the values of x [21]. Linear regression allows inferences not only for samples for which the data is known, but also for those corresponding to x's not present in the data. Three types of inferences are possible:

1. estimate the slope β_1 and the intercept β_0 of the regression line;
2. estimate the mean response μ_y, for a given value of x;
3. predict a future response y for a given value of x.

In general, the goal of linear regression is to find the line that best predicts y from x. Linear regression does this by finding the line that minimizes the sum of the squares of the vertical distances of the points from the line. The estimated values for β_0 and β_1 called b_0 and b_1 are obtained using previous observations as stated by equations 7.2 and 7.3. The intercept b_0 is given by:

$$b_0 = \overline{y} - b_1 \times \overline{x} \tag{7.2}$$

The slope b_1 is given by:

$$b_1 = cr * \frac{s_y}{s_x} \tag{7.3}$$

where $\overline{y}$ is the mean of the observed values of y, $\overline{x}$ is the mean of the observed values of x, cr the correlation between x and y given by equation 7.4, s_x and s_y the standard deviations of the observed x and y, respectively, given by equation 7.5.

$$cr = \frac{1}{n-1} \sum_{i=1}^{n} \left(\frac{x_i - \overline{x}}{s_x} \right) \left(\frac{y_i - \overline{y}}{s_y} \right) \tag{7.4}$$

$$s_x = \sqrt{\frac{1}{n-1} \sum_{i=1}^{n} (x_i - \overline{x})^2} \texttt{ and } s_y = \sqrt{\frac{1}{n-1} \sum_{i=1}^{n} (y_i - \overline{y})^2} \tag{7.5}$$

with n, the number of previous observations from x and y.
After the slope and the intercept of the regression line are estimated, it is possible to predict the value of y (called $\hat{y}$) from a given x using the regression line equation:

$$\hat{y} = b_0 + b_1 \times x \tag{7.6}$$

Note that linear regression does not test if the data are linear. It assumes that the data are linear, and finds the slope and the intercept values for a straight line that best fits the known data. If the error in measuring x is large, other inference methods are needed.

Linear Regression Prediction in the EA

If the environment changed periodically at fixed time steps, the generation when the next change would occur could be successfully predicted by linear regression. The prediction of the moment of the next change was calculated as follows:

- the first two changes of the environment were memorized after they happen (no prediction could be made yet);
- change k, $k > 2$, could be predicted using the equation 7.6.

As we saw, the linear regression needs at least two previous observations to construct the regression line that will be used to make predictions. This is the reason why the first two changes in the environment could not be predicted. After the first two changes, and using the values where those changes occurred, an approximation of the regression line was built and predictions about the next possible moment of change were provided. Then, each time a change occurred, new values of b_0 and b_1 were computed and, using equation 7.3, the regression line was updated.

Example

For example, suppose that three observations were already made ($n = 3$):

Observation	x	y
1	1	50
2	2	100
3	3	150

The first change occurred at generation 50, the second change at generation 100 and the third change at generation 150. Could we predict when will occur the fourth change?
Using equations 7.3 and 7.2, the estimated values for the slope and intercept of the regression line were:

$$b_1 = 1 * \frac{50}{1} = 50$$
$$b_0 = 100 - b_1 * 2 = 100 - 50 * 2 = 0$$

Using equation 7.6 the predicted value for the fourth change ($x = 4$) would be:

$$\hat{y}_4 = b_0 + b_1 * x = 0 + 50 * 4 = 200$$

In this example, the changes in the environment were determined with a fixed change period of size $r = 50$, so the predictions were exact and there were no related errors.

7.1.2 Nonlinear Regression Predictor

Usually, linear regression is used to model relationships between variables that follow a linear correlation. Curved patterns can be modeled using linear regression (e.g. polynomial regression) but nonlinear regression is often used for these patterns because it allows for modeling a wide range of functions.

Basics of Nonlinear Regression

The basic idea of nonlinear regression is the same as that of linear regression, namely to relate a response y to a vector of predictor variables x [21]. Nonlinear regression is characterized by the fact that the prediction equation depends nonlinearly on one or more unknown parameters. The basic form for a nonlinear model between the response y and a predictor x is given as:

$$y_i = f(x, \theta) + \epsilon_i \tag{7.7}$$

where y_i and x_i are the data, f is a nonlinear function involving the predictor and the parameter vector θ and ϵ_i is a random error.
For instance, let's assume the asymptotic regression model:

$$f(x) = \theta_1 - \theta_2 * \theta_3^x \texttt{ where } 0 < \theta_3 < 1 \tag{7.8}$$

if we change the values for θ_1, θ_2 and θ_3 we can model different types of nonlinear curves. Figure 7.1 shows two examples of curves using different values for the parameters θ_i. Although nonlinear regression is less intuitive and more complicated to use, it is more powerful because it allows predictions in both linear and nonlinear data. Thus, this technic is more suitable to model information of real world which is almost all nonlinear. The difficult task in nonlinear regression is to estimate the correct values for the parameter vector θ_i. Once the estimation of the parameters is done, predictions can be performed using the nonlinear function. The different techniques for estimating the nonlinear parameters are briefly described in the next section.

Parameter Estimation in Nonlinear regression

The task of parameter estimation for nonlinear regression is not straightforward. Usually, statistical software using numerical algorithms is used to analyze the data and produce the best parameter's choice for that data [21]. A nonlinear parameter estimation problem is an optimization problem whose goal is to minimize the sum of squared errors given by equation 7.9.

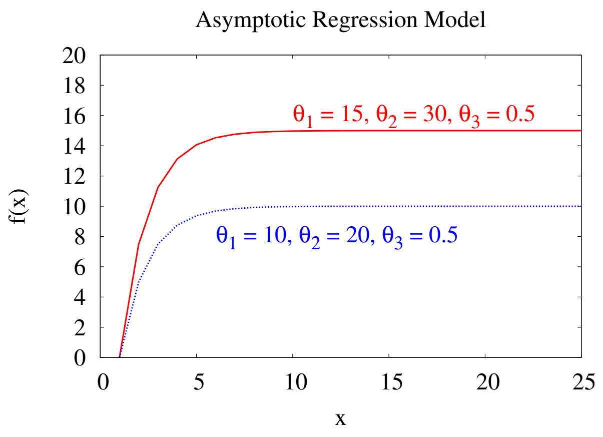


Figure 7.1: The asymptotic regression model

$$Sum_{e_i} = \sum_{i=1}^{n} (y_i - f(x_i, \theta_i))^2 \tag{7.9}$$

Rather than minimizing the sum of squared errors, other techniques minimize the sum of absolute deviations. Several function minimization methods are used in parameter estimation, for instance, weighted least squares, maximum likelihood, Quasi-Newton method, Simplex procedure or Hooke-Jeeves pattern moves [21], [63]. In general, these methods are not easily controllable and require much auxiliary information to work correctly.
Another option, more general and easy to apply, is to use a genetic algorithm to evolve a population of individuals that minimize an objective function. This approach was introduced and successfully tested by Pan et. al [66] and was used in this thesis to estimate the parameter vector θ_i. This GA is described in the section 7.1.2.

Nonlinear Regression Prediction in the EA

In this thesis four different functions were used in the nonlinear predictor. The four functions are defined by the equations 7.10 through 7.13.

$$y = \theta_1 + \theta_2 \times x + \theta_3 \times x^2 \tag{7.10}$$

$$y = \frac{\theta_1 \times x}{\theta_2 + \theta_3 \times x} \tag{7.11}$$

$$y = \frac{\theta_1}{1 + e^{(\theta_2 - \theta_3 \times x)}} \tag{7.12}$$

$$y = \frac{(\theta_1 \times x)^{\theta_4} + \theta_2 \times \theta_3}{\theta_3 + x^{\theta_4}} \tag{7.13}$$

Each one of these functions, using different values for the vector θ can model a wide set of data. Figure 7.2 shows how the different functions can be used to define the change period, using a specific set for the vector θ:

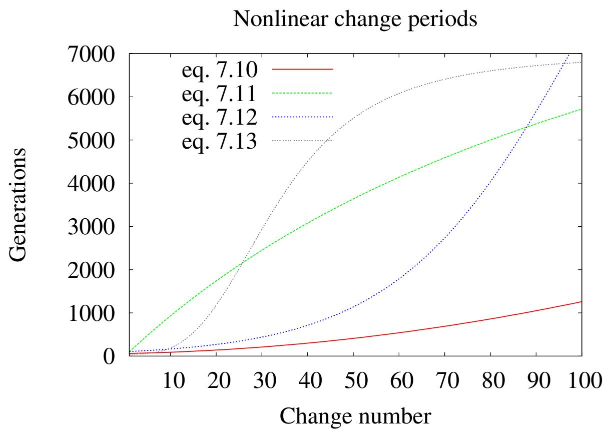


Figure 7.2: Different types of nonlinear change periods

As we can see, different types of change periods are created with the nonlinear functions: function 7.10 creates a rapid change period, i.e, with few generations between two changes. With function 7.12 we have a change period that initially changes very quickly but slows down as time proceeds.
The main limitation of nonlinear regression is that the function that is used to model the information must be known. The parameter vector θ is unknown and is estimated during the run. Different values for θ allow to model different types of curves.

The nonlinear regression module was used in the EA with a set of n functions $f1, f2, ..., fn$, that could be used to give the predictions [1]. At time t, only one function was active. The choice of that function was made measuring the prediction errors of all functions. The function with lower error was the selected one. As we said, the vector parameter θ was estimated using a standard GA. Every time a change occurred in the environment and additional information was available, the GA was executed to find a vector θ that better fitted the data. Thus, the vector θ was estimated using only the **known data**. Using these estimated parameters and the selected function, the predictor estimated **when** the next change would occur. After the real change had occurred, the prediction error was computed. If the error was superior to an established threshold α_p, the module analyzed all the available functions to see if this error could be reduced. If so, a different function was used for future predictions. Figure 7.3 shows how the proposed module works. At the beginning, all functions were evaluated and the function with least prediction error was used to make the next prediction. If all the functions had equal errors, this choice was made at random.

[1]Currently, as we said, we are only using four.

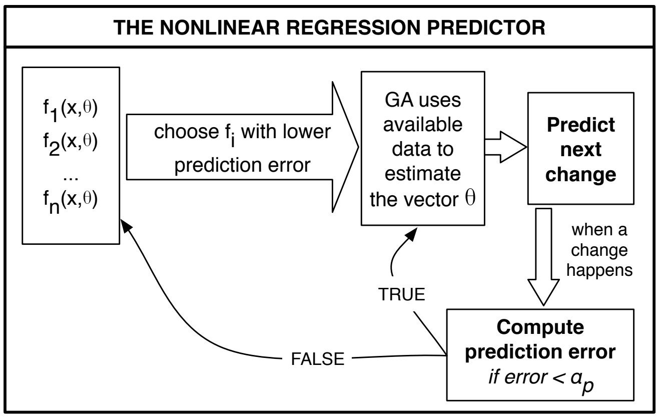

THE NONLINEAR REGRESSION PREDICTOR
$f_1(x,\theta)$
$f_2(x,\theta)$
...
$f_n(x,\theta)$
choose f_i with lower prediction error
GA uses available data to estimate the vector θ
Predict next change
when a change happens
Compute prediction error *if error < a_p*
TRUE
FALSE

Figure 7.3: The nonlinear regression predictor

Genetic Algorithm for Estimating the Parameters

A standard GA, as suggested by [66], was used to estimate the vector parameter θ used in the nonlinear functions.

The GA used a population of binary strings which corresponded to different values of θ_i. The required number of genes was determined using the desired precision and the domain size for each parameter. If the used precision was s, and the lower an upper limits for the parameter k were ${b_k}^L$ and ${b_k}^U$, respectively, the number of genes to encode the parameter k was the minimum m_k, satisfying the following condition:

$$2^{{m_k}^{-1}} < {b_k}^U - {b_k}^L \leq 2^{m_k}$$

Before evaluating the individuals, a decode of the binary string into a real value was performed. The real value for the parameter θ_k was found using equation 7.14.

$$\theta_k = {b_k}^L + d_k(\frac{{b_k}^U - {b_k}^L}{2^{m_k} - 1}) \tag{7.14}$$

where d_k corresponded to the integer value represented by the corresponding m_k genes.

After the decoding operation, the individuals were evaluated using the fitness function. This function aimed to minimize the sum of the least squares errors (equation 7.9). Because individuals with a higher fitness were selected more often, after some generations the best individual represented the optimal solution for θ. The initial population was generated at random and was evolved using tournament selection, uniform crossover and flip mutation. The best individual of the previous population was transferred to next population to guarantee that the best solution found so far wasn't lost. The parameters used in this GA were the following: population of size 50, crossover rate of 75% and mutation rate of 1%. The GA was run for 1000 generations, or until the fitness function had no alterations for 10 generations - this usually meant that the optimum values were found. The domain of the parameters, and consequently the chromosome size, depended on the nonlinear function as shown in Table 7.1. The precision used for each parameter θ was six places after the decimal point. The initial

f	θ_i	${b_k}^L$	${b_k}^U$	m_k	m_{total}
	θ_1	-10	65	27	
f_1	θ_2	0	200	28	80
	θ_3	0	5	25	
	θ_1	0	65	26	
f_2	θ_2	0	1	20	65
	θ_3	0	0.5	19	
	θ_1	14000	15000	34	
f_3	θ_2	0	5	23	76
	θ_3	0	0.5	19	
	θ_1	9000	10000	30	
f_4	θ_2	50	100	26	103
	θ_3	50	100	26	
	θ_4	0	2	21	

Table 7.1: Upper and lower limits for the parameters' domain

parameters' domains, presented in Table 7.1 were chosen to enclose a wide set of nonlinear curves. When the GA was run, in order to provide faster and correct estimations, using the known data, an alteration could be made on these initial domains. This task was controlled by a parameter s_d ($s_d \in [0.0, 1.0]$) that was changed during the run, and applied to some of the chromosomes. If the individuals using the new domain size had the same or better fitness than the best solution found so far, the domains size were adjusted for all the individuals of the next generation.

Example for the nonlinear predictor

For example, suppose that twenty observations were already made ($n = 20$) and the change period followed a nonlinear behavior given by function 7.11. Using this function, changes followed a nonlinear curve. For instance, the first change occurred at generation 100, the second change at generation 150, the third at generation 180, and so on, as illustrated in table 7.2 and figure 7.4. Could we predict when the 21st change would occur?

The first step to predict when the next change would occur was to estimate the values for the vector θ that better fitted the known data. The GA was run and, for the 20 known observations, it estimated the following values:

$$\theta_1 = 15$$
$$\theta_2 = 0.1$$
$$\theta_3 = 0.05$$

Observation	x	y
1	1	100
2	2	150
3	3	180
4	4	200
5	5	214
6	6	225
7	7	233
8	8	240
...	...	...
17	17	268
18	18	270
19	19	271
20	20	273

Table 7.2: Nonlinear change period

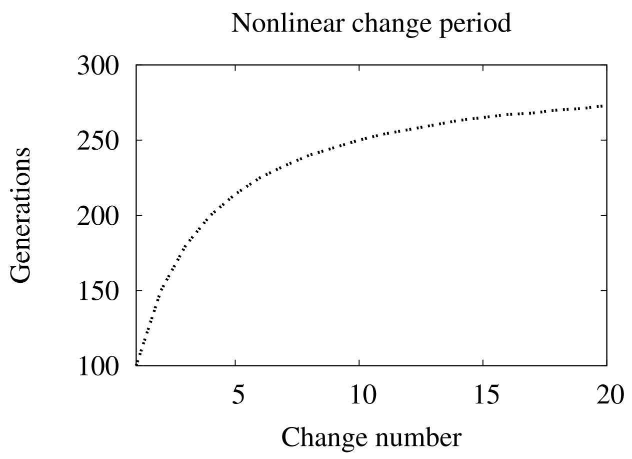


Figure 7.4: Nonlinear change period

If function f_2 was being used, these values were introduced in equation 7.11 and the predicted value for the 21^{st} change was generation 274:

$$y = \frac{15 \times 21}{0.1 + 0.05 \times 21} = 274$$

Then, the prediction error was calculated after the change occurred. If this error was small, f_2 continued to be used for future predictions, otherwise, a different function was selected.

7.2 Predicting *how*

The prediction of how the environment will be modified in the next change, combined with the prediction about the generation where that change will happen, makes possible the preparation of the population **before** the change. If we know how the next environment will look we can introduce the appropriate individuals into the population. This way, when the change effectively happens, the EA will be prepared to face the new environmental conditions.
To gather information about the characteristics of the known environments we used a Markov chain. The information stored in the Markov chain was also used to estimate which environment(s) could appear in the next change.

7.2.1 Markov Chain Predictor

Basics of Markov Chains

A Markov chain can be defined as a sequence of random variables X_1, X_2, X_3, ... (countable set of states) that do not keep memory of the whole past. In fact, Markov chains are memoryless, meaning that the present state is enough to predict future states, i.e.:
$Pr(X_{n+1} = x|X_n = x_n, ..., X_1 = x_1) = Pr(X_{n+1} = x|X_n = x_n)$

A discrete Markov chain model can be defined by the tuple S, P, λ:

- S is the state space, a finite or countable infinite set of possible values for a sequence of random variables X_1, X_2, X_3, ...
- P is a matrix representing transition probabilities between states. In the matrix P, the element p_{ij} is the probability of going from state X_i to state X_j;
- λ is the initial probability distribution for all the states in S. $\lambda = p_0, p_1, p_2, ...$ with p_i, the probability of starting at state X_i.

Markov chains are often described by a directed graph, where the edges are labeled by the probabilities of going from one state to another state. Figure 7.5 shows a graphical representation of a Markov chain with 5 states and the following matrix probability:

$$P = \begin{pmatrix} 0.00 & 1.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.25 & 0.75 \\ 1.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.25 & 0.00 & 0.25 & 0.00 & 0.50 \\ 0.00 & 0.00 & 1.00 & 0.00 & 0.00 \end{pmatrix}$$

Markov Chain Prediction in the EA

In our approach, we used two markov chains:

- System Markov chain (**SMC**): was created off-line and used to define the dynamics of the environment during the entire run.

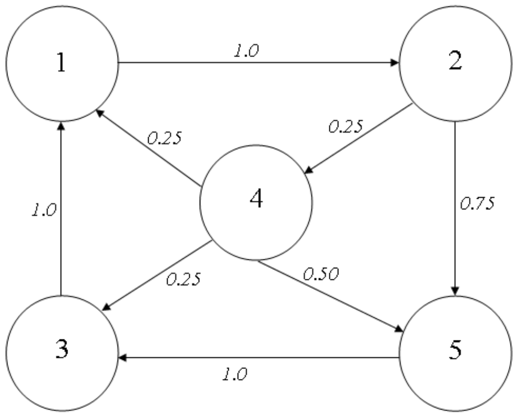


Figure 7.5: Markov chain with 5 states

- Algorithm Markov chain (**AMC**): was empty at the beginning, and was created on-line as new information was gathered from the evolutionary process.

Each state of the Markov chain corresponded to an environment. The **SMC** was created at the beginning with all the possible states and probabilities transitions. In this model, the initial probabilities vector λ was also created. This component was used to decide how the environment changed, but all the information contained in it was **unknown** to the evolutionary algorithm. The **AMC** was built on-line by the algorithm and was used to make predictions. Initially, this component was empty, and if a new environment appeared, a new state was added to the **AMC** and the corresponding matrix P was updated. In a perfect scenario, at the end of the run, the **AMC** was equal to the **SMC**. Section 7.4 gives more detail about this topic.

Example

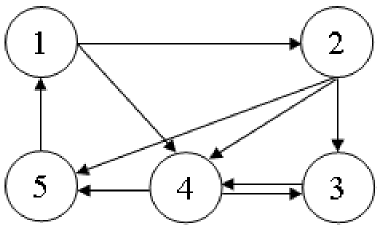
In this example we assume that the maximum number of states is 5 and thee **SMC** defined *a priori* is:

- $\lambda = 1.0, 0.0, 0.0, 0.0, 0.0$
- transition matrix:

$$P = \begin{pmatrix} 0.00 & 0.50 & 0.00 & 0.50 & 0.00 \\ 0.00 & 0.00 & 0.50 & 0.25 & 0.75 \\ 0.00 & 0.00 & 0.00 & 1.00 & 0.00 \\ 0.00 & 0.00 & 0.25 & 0.00 & 0.75 \\ 1.00 & 0.00 & 0.00 & 0.00 & 0.00 \end{pmatrix}$$

- each state corresponds to a different environment:

All this information is unknown to the EA and is used to decide how the environment will change.



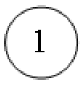
The **AMC** starts without any information:

$$P_{AMC} = \begin{pmatrix} 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \end{pmatrix}$$

Let's see step by step how the **AMC** is built and how the Markov predictor works:

Step 1:

- Initial state: 1



- Next states: 2 or 4

Step 2:

- Randomly chooses state 2

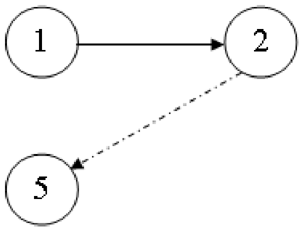
- The **AMC** transition matrix is updated:

$$P_{AMC} = \begin{pmatrix} 0.00 & \mathbf{1.00} & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \end{pmatrix}$$

- Next states (predicted): none
- Next states (real): 3, 4 or 5

Step 3:

- Randomly chooses state 5



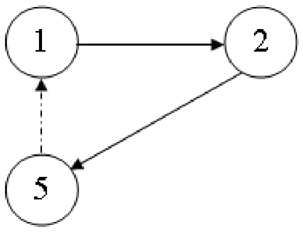
- The **AMC** transition matrix is updated:

$$P_{AMC} = \begin{pmatrix} 0.00 & 1.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & \mathbf{1.00} \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \end{pmatrix}$$

- Next states (predicted): none
- Next states (real): 1

Step 5:

- Go to state 1



- The **AMC** transition matrix is updated:

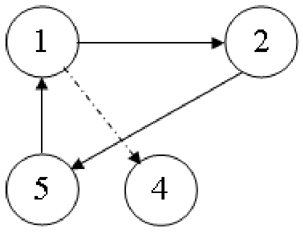
$$P_{AMC} = \begin{pmatrix} 0.00 & 1.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 1.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ \mathbf{1.00} & 0.00 & 0.00 & 0.00 & 0.00 \end{pmatrix}$$

- Next states (predicted): 2

- Next states (real): 2 or 4

Step 6:

- Randomly chooses state 4



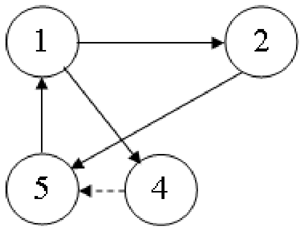
- The **AMC** transition matrix is updated:

$$P_{AMC} = \begin{pmatrix} 0.00 & \mathbf{0.50} & 0.00 & \mathbf{0.50} & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 1.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 1.00 & 0.00 & 0.00 & 0.00 & 0.00 \end{pmatrix}$$

- Next states (predicted): none
- Next states (real): 3 or 5

Step 7:

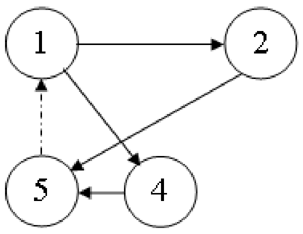
- Randomly chooses state 5



- The **AMC** transition matrix is updated:

$$P_{AMC} = \begin{pmatrix} 0.00 & 0.50 & 0.00 & 0.50 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 1.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & \mathbf{1.00} \\ 1.00 & 0.00 & 0.00 & 0.00 & 0.00 \end{pmatrix}$$

- Next states (predicted): 1



- Next states (real): 1

Step 8:

- Go to state 1
- The **AMC** transition matrix is updated (no modifications are made this time):

$$P_{AMC} = \begin{pmatrix} 0.00 & 0.50 & 0.00 & 0.50 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 1.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 1.00 \\ 1.00 & 0.00 & 0.00 & 0.00 & 0.00 \end{pmatrix}$$

- Next states (predicted): 2 or 4
- Next states (real): 2 or 4

On the last two steps, predictions were 100% precise. Continuing this process, the matrix P_{AMC} would evolve towards the values of matrix P defined off-line at the beginning.

7.3 Anticipation

The efficacy of the two predictors described before was only achieved if they were used at the right moment. In order to prepare the population before the changes happen, the anticipation mechanism must be robust and capable of dealing with erroneous predictions. The prediction errors should be used to improve the next predictions' values.
The "right moment" to start preparing the population was decided using the values predicted either by the linear predictor or by the nonlinear predictor. Both estimated the generation when the next change would be observed. Knowing this value, the system started the preparation for the change *some generations before*. If the prediction mechanisms were accurate and the correct information was introduced in the main population **before** the change, the EA's performance was not affected by the changes in the environment and it continued evolving quickly readapting to the new conditions. When the prediction mechanisms failed and no anticipation was made, when a change occurred, the EA's performance suffered from a sudden decrease and the EA took some time to readapt to the new environment.

A parameter, called Δ, was used to decide how many generations before the predicted moment of change the anticipation started. The value of this parameter was also used to cover minor prediction errors. The value of Δ should be correctly chosen in order to assure that the population was prepared before the change.
Different approaches to assign a value to Δ were tested:

- Δ constant
- Δ adjustable
  - using the maximum prediction error
  - using the average of the positive prediction errors
  - using the average of all the prediction errors (absolute value)
  - using the maximum and the average of the positive prediction errors

Prediction error at time t (e_t) was given by equation 7.15:

$$e_t = g - g' \tag{7.15}$$

where g was the predicted generation for the occurrence of next change and g' was the generation where the change actually happened. Those prediction errors could be positive or negative. A negative error indicated that the predicted value for the next change (g) was smaller than the real value (g'), i.e, $g < g'$ and thus the anticipation of the change was successfully made. A positive error meant the opposite. In this situation, if the value of Δ was greater than this error, the anticipation was made before the change; otherwise, the change was detected only when it occurs and no effective anticipation was executed.
The value of Δ should be chosen in order to cover the prediction error and to guarantee that the preparation for the next change was made before it happens. More explicitly, if the next change was estimated to happen at generation g, at generation g - Δ, the Markov model was used to predict the set of possible future environments. At that time, individuals from the memory were retrieved and introduced into the population, replacing the worst ones. In order to be effective, the Markov model should act before the change. Therefore, if the change was observed at generation g', the value of Δ should assure that the condition $\Delta > |g - g'|$ was observed. In addition, the value of Δ should minimize the computational costs guaranteeing that the anticipation was as close to the change as possible. Thus, the choice of the best value of Δ assumed soaring importance.
The use of a constant value for Δ was a weakness of the system: on one hand, it required preliminary experimentation to decide what value to choose, on another, if the conditions of the change period were altered, a new value should be chosen [92]. To overcome this limitation we proposed several mechanisms that used the previously observed prediction errors to continuously change the value of Δ. In all studied approaches the parameter Δ was initialized with the value 5 and this value was used for the first two changes when no predictions could be made. After that, the value of Δ was updated according to the studied

methods, but it could not be lower than a minimum value of 2. This restriction assured that the preparation for next change was made at least two generations before it has occurred.
All the proposed approaches used, in different ways, the previously observed errors to learn how to adapt the value of Δ. The methods described next adjusted the value of Δ during the run aiming to make the system more efficient and robust.

7.3.1 Using the maximum prediction error

This method was called *Max_Err* and updated the value of Δ in the following way: Δ was initialized with the value 5 and this value was used for the first two changes when no predictions could be made. Thereafter, the value of Δ was changed using the maximum observed prediction error. In a more formal way, at the k_{th} change, Δ was updated as follows:

$$\Delta_1(k) = \begin{cases} 5 & \text{if } k = 1,2 \\ max\{2, e_0, e_1, ..., e_k\} & \text{if } k > 2 \end{cases} \tag{7.16}$$

where e_k was the observed error at the k^{th} change.

7.3.2 Using the average of the positive prediction errors

This method was called *Av_Err (+)* and updated the value of Δ in the following way: Δ started with the value 5 and this value was used for the first two changes when no predictions could be made. Thereafter, the value of Δ was changed using the average of the positive errors given by the linear predictor. As previously stated, a positive error meant that the predicted change generation was after the real change. In a more formal way, at the k^{th} change, Δ was updated as follows:

$$\Delta_2(k) = \begin{cases} 5 & \text{if } k = 1,2 \\ max\{2, \frac{\sum_{i=1}^{k} e_i}{k}\} & \text{if } k > 2 \text{ and } e_i > 0 \end{cases} \tag{7.17}$$

where e_i was the observed error at the i^{th} change.

7.3.3 Using the average of all the prediction errors (absolute value)

This method, called Av_Err(all) updated the value of Δ in the following way: Δ started with the value 5 and this value was used for the first two changes when no predictions could be made. Afterward, the value of Δ was changed using the average of the absolute values of all measured errors. In a more formal way, at the k^{th} change, Δ was updated as follows:

$$\Delta_3(k) = \begin{cases} 5 & \text{if } k = 1,2 \\ max\{2, \frac{\sum_{i=1}^{k} |e_i|}{k}\} & \text{if } k > 2 \end{cases} \tag{7.18}$$

where e_i was the observed error at the i^{th} change.

7.3.4 Using the maximum and the average of the positive prediction errors

This method, called Max_Av_Err combined the first two described techniques. The value of Δ was updated in the following way: Δ was initialized with the value 5 and this value was used for the first two changes when no predictions could be made. Thereafter, the value of Δ was computed using the average of the sum of the maximum observed error and the average of all positive errors.

$$\Delta_4(k) = \begin{cases} 5 & \text{if } k = 1, 2 \\ max\{2, \frac{\Delta_1(k)+\Delta_2(k)}{2}\} & \text{if } k > 2 \end{cases} \tag{7.19}$$

7.4 Putting it all together: prediction in the EA

The proposed computational model was called **PredEA** and used a traditional **EA** that evolved a population of individuals aiming to optimize the current fitness function. A memory was used to store useful information from the past that was used in future changes. This traditional memory-based EA was extended with the two above mentioned prediction modules described in sections 7.1 and 7.2. The first module used information about **when** the previous changes occurred to estimate the generation when the next change would be observed. This module was tested using a linear regression predictor and a nonlinear regression predictor. The second module used a Markov chain to keep track of previous environments and provided predictions on **how** the environment would look like in the next change step. The two predictor modules were managed by a third component, the anticipation module, that used the information provided by the previous two modules and prepared the EA for the next change. Figure 7.6 illustrates the proposed architecture.

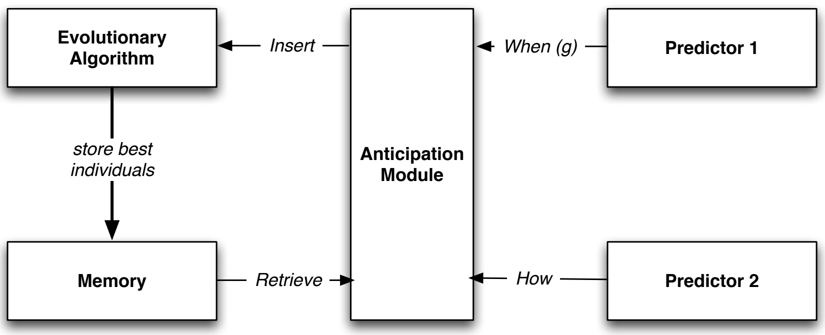


Figure 7.6: Prediction modules in the memory-based Evolutionary Algorithm

Here is a brief description about each one of the components of Figure 7.6:

- ***Evolutionary Algorithm***: standard evolutionary algorithm which evolved a population of individuals through the application of selection, crossover and mutation;

- ***Memory***: stored the best individual of the population in a certain moment;
- ***Predictor 1*** ($P1$): stored the generations where different changes occurred and used this information to foresee when the next change would take place. Two approaches were tested, based on linear and nonlinear regression techniques;
- ***Predictor 2*** ($P2$): every time a different environment appeared, this module stored the environmental information. It consisted of a set of states, a matrix of state transition probabilities and the initial probability vector. Each state corresponded to a different environment. The initial probability vector was initialized by choosing the initial state randomly. The state transition probability matrix started filled with zeros and was updated on-the-fly when different environments appeared. When this module was called, it used all the available information to estimate to which environment(s) the system would change;
- ***Anticipation module*** (A): this module managed all the information provided by the two predictors, and computed the value of Δ which was used to decide when to activate the mechanisms to prepare the **EA** to the next change. At that time, information from the memory was retrieved and inserted into the population. This information corresponded to those individuals that could be useful to the next predicted environments.

Next sections detail each one of these modules.

7.4.1 Evolutionary Algorithm

Consisted of a standard memory-based EA. A main population of individuals evolved by means of selection, crossover and mutation and was used to find the best solution for the current environment. Another population was used as memory to store the best current individual from time to time. When a change happened or was predicted, the information stored in memory was retrieved and used to help the EA readapt to the new environment. This algorithm was similar to **MEGA**, described in a previous chapter, empowered with the prediction modules.

7.4.2 Memory

Memory was used to store the best individuals of the current population. It started empty and had a limited size (20 individuals). The update time, $t_m(t)$ was computed using the following equation:

$$\begin{aligned} t_m(0) &= rand(5,10) \\ t_m(t) &= t_m(t-1) + rand(5,10) \end{aligned}$$

An individual was stored using the **generational** replacing strategy, described in chapter 5.
Memory was also used to **detect changes** in the environment: a change occurred when at least one individual in the memory had its fitness changed. The memorized individuals were associated with the environments where they were the best solution.

7.4.3 Predictor 1 module (P1)

This predictor used information about when previous changes were observed to estimate when the next change would occur. If the change period was periodic or follows a linear behavior, this module could be used with a simpler predictor based on linear regression techniques. If the changes in the environment occurred in generations following a nonlinear function, this predictor should be based on nonlinear regression methods. If the type of change period was unknown to the EA, the second predictor, based on nonlinear regression, should be used since it can accurately predict both situations.

7.4.4 Predictor 2 module (P2)

This module consisted of two parts. One was hidden from the system (unknown to the algorithm) and was built off-line with the following information: the maximum number of different states that may appear and a probability matrix of transitions that was used to model how the environments changed. The second part was built on-line by the algorithm and was used to make predictions. It kept track of the different environments and estimated which environments should appear in the next change. Those predictions were made using only the information known so far about the previous environmental changes. Each state of the Markov chain corresponded to a different environment. If two states were linked, it meant that a change happened from one state to the other. Associated to each transition was a probability value which was updated every time a change was detected. The initial state was randomly chosen among the existing states. Again we stress that this information was unknown to the algorithm and the model was updated throughout time. The information that was stored about each environment was problem dependent. For instance, in the dynamic bit-matching problem, each state corresponded to a different template.

7.4.5 Anticipation module (A)

This module received the information provided by the two predictors and decided when to start the preparation of the EA for the next change. This activation should be done at the correct time in order to prepare the population to the next environment(s) predicted by the $P2$ module. The $P1$ module estimated the generation when the next change would be observed and the A module started the anticipation some generations before. If the prediction mechanisms were accurate and the correct information was introduced in the main population before the change, the EA's performance wa not affected by the changes in the environment. When the prediction mechanisms failed and no anticipation was made, when a change occurred, the EA's performance was affected and the EA took some time to readapt to the new environment. The Δ parameter described previously was used by this module to decide how many generations before the predicted moment of change (g) the A module should be activated. The value of this parameter was also used to cover minor prediction errors associated with the $P1$'s estimations. The anticipation process consisted of retrieving from memory individuals that were good solutions in the environments that the $P2$ module indicated as the next to appear. These individuals were inserted into the main population at the generation $g - \Delta$, replacing the worst individuals. If the $P2$ module didn't provide any prediction, five random individuals from memory were inserted into the population, replacing five randomly selected individuals. Figure 7.7 shows the pseudo code of the **PredEA** algorithm.

```
Function $PredEA$
    $t_m$: time to update memory
    $max$: maximum number of states of the Markov chain
    $markov$: Markov model defined off-line (SMC)
    $initial\_state$: initial state
```

```
$t = 0$
$t_m = rand(5, 10)$
$P(0)$ = initialize population randomly
$M(0)$ = initialize memory randomly
$\Delta(0) = 5$
Initialize the $AMC$ information
repeat
   evaluate memory $M(t)$ and population $P(t)$
   replace the worst in $P(t)$ by the best individual
       from $P(t-1)$
   if a change is detected
         Store performance measures
         Activate the $P1$ module
             Update $P1$ information
             Predict when next change will occur ($g$)
         Update the value of $\Delta(t)$
         Update the $AMC$ information
         if prediction was unsuccessful
            $P'(t)$ = retrieve information from $M(t)$
  else $P'(t) = P(t)$

  if $g$ (next_change) is close (as defined by $g - \Delta(t)$)
         Activate the $P2$ module
             Predict next state(s)
             Activate the $A$ module
                 $P'(t)$ = retrieve information from $M(t)$
  else $P'(t) = P(t)$

  if $t = t_m$ or change detected
         update memory
         $t_m = t + rand(0, 5)$
  $P''(t) = Selection(P'(t))$
  $Crossover(P''(t))$, $Mutation(P''(t))$
  $P(t+1) = P''(t)$
  $t = t + 1$
until $stop\_condition$ is true
```

Figure 7.7: Pseudo code of the **PredEA**

Chapter 8

Experiments

This chapter introduces the experiments that were completed in order to study the performance of all the contributions given in this thesis. It provides a description of the problems used, the general parameter settings common to all the techniques, and the statistical validation applied to the results.

8.1 Benchmark problems

Our study relied on the use of well-known benchmark problems. The decision about which ones to use, and how to use them, was made having in mind the two classes of improvements analyzed in this thesis: prediction, diversity and memory. As far as we know, there are no studies proposing solutions that can predict *how* and *when* the environment changed. Thus, for this topic, we analyzed the results obtained with our approaches an no comparisons with other studies were made. For the second class of questions that we addressed (diversity ad memory), in order to compare our results with other analogous studies, we used the Yang's Dynamic Optimization Problem (DOP) generator [126]. This DOP generator allows parameterizing of different aspects of the environment, such as the change period, the severity of the change and the predicability of the environment. The DOP generator can construct different dynamic environments from any binary-encoded stationary function using the bitwise exclusive-or (XOR) operator. We used four problems under the DOP generator: the knapsack problem, the onemax problem and the royal road functions $F1$ and $F2$.
To test the performance of the prediction module we needed to have control over the number of different environments. We also needed to make different types of transitions between different states. As a result, we used a slightly modified version of the dynamic knapsack problem and of the dynamic bit matching problem. The next sections describe the DOP generator and all the problems used.

8.1.1 Knapsack Problem

The knapsack problem is a NP-complete combinatorial optimization problem often used as benchmark. It consists of 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 (C), i.e.,:

$$\texttt{max}\ profit(x) = \sum_{i=1}^{m} v_i x_i \tag{8.1}$$

subject to the following constrain:

$$\sum_{i=1}^{m} w_i x_i \leq C \tag{8.2}$$

The initial values (v_i), weights (w_i) and capacity (C) were created using strongly correlated sets of randomly generated data [55]:

$$w_i = rand(1, 50) \tag{8.3}$$

$$v_i = w_i + rand(1, 5) \tag{8.4}$$

$$C = 0.6 \times \sum_{i=1}^{m} w_i \tag{8.5}$$

$rand(min, max)$ uniformly generates a random number between min and max, and m is the number of items.
The fitness of an individual x using binary representation 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. The fitness function is defined as follows ([126]):

$$f(x) = \begin{cases} \sum_{i=1}^{m} v_i x_i & \text{if } \sum_{i=1}^{m} w_i x_i \leq C \\ \\ 10^{-10} \times (\sum_{i=1}^{m} w_i - \sum_{i=1}^{m} w_i x_i) & \text{otherwise} \end{cases} \tag{8.6}$$

The time-varying knapsack was created by using the DOP generator (described next) or by changing the value of the knapsack capacity. The number of items used in our experimentations was $m = 100$.

8.1.2 Bit-matching Problem

The bit-matching problem is an unimodal problem whose goal is to find a solution that matches a given template. Using a binary template, the fitness function of an individual x is:

$$\texttt{max}\ match(x) = \sum_{i=1}^{l} x_i\ \texttt{is equal to}\ t_i \tag{8.7}$$

where l is the chromosome length and t_i is the bit of the actual template at gene i.
Changing the template from time to time turns this problem dynamic. The number of bits that change in the template defines the severity of the change. The difficulty of the problem can be increased using templates with larger dimensions.
The onemax problem, where the EA has to find a solution that maximizes the number of ones, is a particular case of the bit-matching problem.
Both bit-matching template and onemax problems were transformed from static to dynamic using the DOP generator, described next.
We used chromosomes of 300 bits for the onemax problem and binary templates of length 100 for the dynamic bit-matching problem.

8.1.3 Royal road functions

The royal road functions were introduced by Mitchell et al. [56] and are suitable for testing the EA's performance concerning recombination and schema processing. They consist of a list of partially specified bit strings (schemas) s_i, in which '*' denotes a wild card (i.e., allowed to be either 0 or 1). A bit string x is said to be an instance of a schema s, i.e., $x \in s$, if x matches s in all non-'*' positions.
Each schema s_i contributes with a coefficient c_i which is equal to the schema's order, i.e. $c_i = o(s_i)$. The order of a schema s_i, is the number of defined bits in s_i.
For the royal road F1 ($RR1$), $c_i = 8$ for all s_i, with $i = 1..8$. For the royal road F2 ($RR2$), $c_i = 8$ for s_i $(i = 1..8)$, $c_i = 16$ for s_i $(i = 9..12)$ and $c_i = 32$ for s_i $(i = 13, 14)$.

Equations 8.8 and 8.9 define the fitness of a binary string for the royal road functions $RR1$ and $RR2$:

$$RR1(x) = \sum_{i=1}^{8} c_i \delta_i(x) \tag{8.8}$$

$$RR2(x) = \sum_{i=1}^{14} c_i \delta_i(x) \tag{8.9}$$

where $\delta_i(x) = 1$, if $x \in s_i$ and $\delta_i(x) = 0$, otherwise.

Figures 8.1 and 8.2 provide examples for the two royal road functions. In practice, this means that the fitness of a binary string x is calculated by summing the coefficients c_i corresponding to each of the given schema s_i of which x is an instance.

s_1 =11111111**

s_2 =********11111111**

s_3 =****************11111111**

s_4 =************************11111111********************************

s_5 =********************************11111111************************

s_6 =**11111111****************

s_7 =**11111111********

s_8 =**11111111

s_{opt} =1111111111111111111111111111111111111111111111111111111111111111

Figure 8.1: Royal Road Function 1

s_1 =11111111**

s_2 =********11111111**

s_3 =****************11111111**

s_4 =************************11111111********************************

s_5 =********************************11111111************************

s_6 =**11111111****************

s_7 =**11111111********

s_8 =**11111111

s_9 =1111111111111111**

s_{10} =****************1111111111111111********************************

s_{11} =********************************1111111111111111****************

s_{12} =**1111111111111111

s_{13} =11111111111111111111111111111111********************************

s_{14} =********************************11111111111111111111111111111111

s_{opt} =1111111111111111111111111111111111111111111111111111111111111111

Figure 8.2: Royal Road Function 2

Both royal road functions used binary representations with chromosomes of size 64. They were transformed from static to dynamic using the DOP generator, described next.

8.1.4 Dynamic Optimization Problem Generator

The dynamic optimization problem (DOP) generator, proposed in [112, 126], is used to construct different types of dynamic environments from any binary-encoded stationary function using the exclusive-or operator. The full description of the generator is given bellow according to the explanation provided in [121].

Two parameters are used to define the characteristics of the environment: r and ρ. The speed of the change is controlled by r, which defines the number of generations between changes. The severity of the change is controlled by the parameter $\rho \in [0.0, 1.0]$. At $\rho = 0.0$ the environment is static; the highest severity occurs when $\rho = 1.0$.
The DOP generator can construct three types of dynamic environments: cyclic, cyclic with noise and random dynamic environments.
For cyclic dynamic environments, $2K$ XOR masks are created:

$M(0), M(1), ..., M(2K-1)$ and the environment will cycle among those masks in a fixed logical loop. K is the number of different templates. The evaluation of an individual x at generation t in cyclic dynamic environments is made according to the following equation:

$$f(x,t) = f(x \otimes M(I_t)) = f(x \otimes M(k\%(2K))) \tag{8.10}$$

$k = t/r$ gives the index of the current environmental period: 1, 2, 3, ... and $\otimes$ is the exclusive-or (xor) operation.
$I_t = k\%(2K)$ is the index of the mask M that encodes the environmental characteristics for generation t.

The $2K$ XORing masks are generated using K binary templates T previously created: $T(0), ...T(K-1)$. Each template T contains a number of ones equal to $\rho = 1/K$, with $\rho \in [1/l, 1.0]$ and l the length of the chromosome. This means that, using a higher severity, the number of different environments is smaller. Besides that, the number of templates and the value of ρ also depend on the chromosome length. For instance, for a chromosome of length equal to 100, a $\rho = 0.8$ can not be used, because the result is not an integer value.
The initial mask is $M(0) = \{0\}$ and the remaining masks are created as follows:

$$M(i+1) = M(i) \otimes T(i\%K) \tag{8.11}$$

with $i = 0, 1, ..., 2K-1$.

The templates T are first used to create K masks until $M(K) = \{1\}$ and then are used again in the same order until $M(2K) = \{0\}$.

Cyclic dynamic environments with noise are created as described before, but the mask $M(i)$ is created by mutating the previous mask with a small probability per gene. For random dynamic environments, the XOR mask $M(i)$ applied to the individuals is always randomly generated every time the environment is changed.

Example
In the next example we show how the DOP generator can construct different cyclic dynamic environments using chromosomes of size $l = 10$, $\rho = 0.1$. These two parameters are used to determine the number of templates (K) to create. So, $2K = 2/\rho = 20$ $(K = 10)$.

The first step consists in creating the $K = 10$ templates T, each one containing a number of ones equal to the product $\rho \times l = 1$:
$T(0) = 1000000000$
$T(1) = 0100000000$
$T(2) = 0010000000$
$T(3) = 0001000000$
...
$T(9) = 0000000001$

The $2K$ M masks are created in the second step. The first mask $M(0)$ is set to zero and the remaining are generated using the previous templates and the $\otimes$ operator:

$M(0) = 0000000000$

$M(1) = M(0) \otimes T(0) = 1000000000$
$M(2) = M(1) \otimes T(1) = 1100000000$
$M(3) = M(2) \otimes T(2) = 1110000000$
...
$M(10) = M(9) \otimes T(9) = 1111111111$

$M(11) = M(10) \otimes T(0) = 0111111111$
$M(12) = M(11) \otimes T(1) = 0011111111$
$M(13) = M(12) \otimes T(2) = 0001111111$
...
$M(20) = M(19) \otimes T(9) = 0000000000$

When $r = 10$, the environment changes every 10 generations at which time the mask $M(k/2K)$ - where k is the index of the environmental cycle - is selected to change the environment.

8.2 Experimentation plan

The experimentation plan was divided in two groups: one for testing the approaches concerning diversity, replacing strategies and population/memory sizes, (**Plan I**) and another for evaluating the performance of the prediction modules (**Plan II**). Because we were interested in testing different aspects in each approach, the experimentation plan was different for each one.
In the experimentation **Plan I**, we used the DOP generator problem on four different problems: the knapsack problem, the onemax problem, and the royal road functions 1 and 2.
We used linear change periods, with changes observed at every r generations, with $r = 10$, $r = 50$, $r = 100$ and $r = 200$. We studied cyclic environments where four different values of ρ were applied to the severity of change: $\rho = 0.1$, $\rho = 0.2$, $\rho = 0.5$ and $\rho = 1.0$.
Table 8.1 summarizes the experimentation **Plan I**, which included, for each tested algorithm, 16 different types of environments. Each environment was run for 56 different algorithms, resulting in a total of 896 different tested situations.

The 56 different algorithms tested on **Plan I** are listed on table 8.2.

	Benchmarks	Change period type	Change period size	Environmental changes	Severity of change
Plan I	DOP-KP	linear (periodic)	$r = 10$	cyclic	$\rho = 0.1$
	DOP-OM		$r = 50$		$\rho = 0.2$
	DOP-RR1		$r = 100$		$\rho = 0.5$
	DOP-RR2		$r = 200$		$\rho = 1.0$

Table 8.1: Experimentation Plan I

Methods		Algorithms			
		MEGA	MIGA	AMGA	VMEA
Diversity	Cx	MEGA-Cx	MIGA-Cx	AMGA-Cx	VMEA-Cx
	Cj	MEGA-Cj	MIGA-Cj	AMGA-Cj	VMEA-Cj
	Tf	MEGA-Tf	MIGA-Tf	AMGA-Tf	VMEA-Tf
Replacing strategies	sim	MEGA-sim	MIGA-sim	AMGA-sim	VMEA-sim
	age1	MEGA-age1	MIGA-age1	AMGA-age1	VMEA-age1
	age2	MEGA-age2	MIGA-age2	AMGA-age2	VMEA-age2
	gen	MEGA-gen	MIGA-gen	AMGA-gen	VMEA-gen
Mem/Pop sizes	$m = K\% \times n$	MEGA-m	MIGA-m	AMGA-m	variable
	$p = n - m$	m is the memory size; p is the population size; n is the total number of individuals; $K = 10, 20, 30, ..., 90$			

Table 8.2: Algorithms tested on experimentation Plan I

In the experimentation **Plan II**, besides the periodic change period allowed by DOP, we were also interested in creating other types of dynamics. Therefore, we used the knapsack and the dynamic bit-matching problem, independently from the DOP generator.
We set a maximum number of different environments: $max = 3, 5, 10, 20$ or 50 and, depending on the problem, we created max different capacities (for the dynamic knapsack problem) or max different templates (for the dynamic bit-matching problem). The different capacities were generated from the initial capacity of the knapsack (created using equation 8.5) and making variations of 20%, following the next equation:

$$C(t) = \begin{cases} 0.6 \times \sum_{i=1}^{m} w_i\texttt{, if } t = 0 \\ C(t-1) - 0.2 \times C(t-1)\texttt{, if } t \texttt{ is odd} \\ C(t-1) + 0.2 \times C(t-1)\texttt{, if } t \texttt{ is even} \end{cases}$$

The different templates were created using an initial template $T(0) = 0$. The following templates were created by changing $\frac{l}{max}\%$ of the bits from the previous template (l was the chromosome length).
According to the classification provided in chapter 3, three different types of change period were used:

- **linear** (periodic): every r generations, with $r = 10$, $r = 50$, $r = 100$ and $r = 200$
- **patterned**: the moments of change were decide by repeating an established pattern. We used four different patterns: 5-10-5, 10-20-10, 50-60-70

and 100-150-100. The generations when the environment changed were calculated as follows:

$$change(i) = change(i-1) + pattern(k)$$

where k was the pattern index, 0, 1 or 2, since all the patterns had size equal to three and i was the change index. For the first change, $change(0) = 0 + pattern(0)$.

For example, for the pattern 5-10-5, the generations when the environment changed were:

$$\begin{aligned}
\text{change}(0) &= 0 + \text{pattern}(0) &&= 5\\
\text{change}(1) &= 5 + \text{pattern}(1) &&= 15\\
\text{change}(2) &= 15 + \text{pattern}(2) &&= 20\\
\text{change}(3) &= 20 + \text{pattern}(0) &&= 25\\
\text{change}(4) &= 25 + \text{pattern}(1) &&= 35\\
\text{change}(5) &= 35 + \text{pattern}(2) &&= 40
\end{aligned}$$

- **nonlinear**: the change period was defined by a nonlinear function. We used four different types of nonlinear functions, as explained in chapter 7.

For each one of the change period types, two different types of environmental changes were defined: cyclic and probabilistic. The environments changed between *max* different states.
For the probabilistic type, the probabilities associated to each different state were set at the beginning of the run and corresponded to the system Markov model (**SMC**) transition matrix (see chapter 7). The severity of the change was not tested with different values, since the Markov model stored information about the fitness function, which was the same, independently of the used severity. So, different severities would not influence the model's performance.
The **PredEA** algorithm was compared with its counterpart used without prediction (**noPredEA**). No further algorithms were compared in this experimental plan, since we didn't find in the literature any memory-based EA providing prediction for the moment of the next change and for the trend of that change. The prediction methods described in Chapter 4 were used in different domains or after the occurrence of the change. In our work, we used a similar technique as proposed by [38] and [129], but its application was made differently: linear and nonlinear methods were used to predict when next change would occur, instead of assuming that the moment of next change was known, as in [38] and [129]. Besides, an additional predictor (the Markov model) was used to analyze and predict the trend of the change.

Table 8.3 summarizes the experimentation **Plan II**, which included, for each tested algorithm, 252 different types of environments. Each environment was run for 10 different algorithms, resulting in a total of 2520 different tested situations.

The 10 different algorithms tested on **Plan II** are listed on table 8.4.
In all the experiments, the information concerning the type of change period,

	Bench.	Change period type	Change period size	Env. changes	nº of states
Plan II	dyn KP dyn BM	linear	10, 50, 100, 200		
		patterned	5-10-5 10-20-10 50-60-70 100-150-100	cyclic probabilistic	3 5 20 50
		nonlinear	(variable, using 4 functions)		

Table 8.3: Experimentation Plan II

Methods		Algorithms	
		PredEA-lr	PredEA-nlr
Prediction	Δ const	PredEA-lr-const	PredEA-nlr-const
	Δ_MaxErr	PredEA-lr-MaxErr	PredEA-nlr-MaxErr
	Δ_AvErr	PredEA-lr-AvErr	PredEA-nlr-AvErr
	Δ_AvErr2	PredEA-lr-AvErr2	PredEA-nlr-AvErr2
	$\Delta\_AvMaxErr$	PredEA-lr-AvMaxErr	PredEA-nlr-AvMaxErr

Table 8.4: Algorithms tested on experimentation Plan II

the type of environmental change, the change period size, and the number of different states were **unknown** to the **EA**.

8.3 Settings

Table 8.5 lists the general parameters used in all the algorithms involved in the experiments. A total of 30 runs were performed with each technique for each problem. Binary representation was used for all the studied problems. In order to have the same number of function evaluations per generation, n was set as follows: MEGA, AMGA and VMEA used $n = 100$. For VMEA, since the sizes of the memory and population change during the run, the value of n can be less than 100 in some situations. MIGA used a value of n computed using equation 8.12, since that the $r_i \times n$ immigrants were also evaluated every generation. The memory size for MEGA, MIGA and AMGA was $m = 20\% \times n$.

$$n = \frac{100}{1 + r_i} \tag{8.12}$$

The EA was allowed to evolve for as many generations as necessary so as to result in 200 environmental changes. The different recombination operators were applied with a probability of 70% and flip mutation was used with 1% rate.

Individual's representation	binary
Initialization	uniform randomly created
Chromosome length	64, 100 or 300
Runs	30
Generations	based on 200 environmental changes ($r \times 200$)
Global number of individuals (n)	100, 84 for MIGA
Memory size (m)	20% $\times n$, variable for VMEA
Population size (p)	$n - m$
Memory replacing strategy	similar, age_1, age_2, generational
Selection method	tournament, size 2
Survivors selection	generational with elitism of size one
Recombination	uniform crossover, transformation or conjugation
Recombination probability	70%
Mutation	flip
Mutation probability	1%

Table 8.5: General settings used in the experiments

Table 8.6 lists the specific parameters to certain algorithms involved in the experiments.

MAX_AGE (used in $age1$ replacing strategy)	100
C (age's increment used in $age1$ replacing strategy)	5
fit_rate (used in $age2$ replacing strategy)	0.1
r_i (used in **MIGA**)	20%
p_i (used in **MIGA**)	1%
α (used in **AMGA**)	0.5
α_t (used in transformation)	0.75
p_t (gene segment pool size used in transformation)	50
α_p (used in prediction, module P2)	10

Table 8.6: Specific parameters used in the experiments

The initial populations and memories were randomly created and the selection of parents was made using the tournament selection method. We used tournament of size two and the winner of the tournament was selected to the mating pool. The next population was formed using the generated offspring, through recombination and mutation, and the best individual (the elite) of previous population was preserved.

8.4 Measures, plots and tables

Two different measures were used to evaluate the different algorithms:

- The **overall performance measure** gives the global score achieved by each algorithm.

$$overall = \frac{1}{G}\sum_{t=1}^{G} best_t$$

G is the number of generations and $best_t$ is the fitness of the best individual at generation t.

- The **off-line performance measure** is used to evaluate how the algorithms evolved during the entire run. At generation t the off-line performance is:

  $$\textit{off-line}(t) = \frac{1}{t}\sum_{i=1}^{t} best'_t$$

  where t is the actual generation and $best'_t$ is the maximum observed fitness since the last time step at which a change in the environment occurred.

The diversity of the population was measured using the formula:

$$Diversity = \frac{1}{l \times p \times (p-1)}\sum_{i=1}^{p}\sum_{j=1}^{p} HD(p_i, p_j)$$

where l is the length of the chromosome, p, the population size, p_i and p_j, the i^{th} and the j^{th} individuals of the population, and HD the hamming distance. The diversity measure presented in next chapters was the averaged diversity over the 30 runs.

8.5 Statistical validation

All the results obtained were statistically validated. The normality of data was verified using the Shapiro-Wilk test with α set to 0.01. In almost all situations, data followed a normal distribution, so the paired two-tailed t-test, at a 0.01 level of significance, was used for assessing the statistical difference of the means over 30 runs of each pair of algorithms. In a few cases, were the normality test failed, the nonparametric Friedman test, also at a 0.01 level of significance, was applied. After this test, the multiple pair wised comparisons were performed using the Nemenyi procedure.
For multiple comparisons, the p-value (0.01) used either in the t-test or in the Nemenyi test was adjusted using the Bonferroni correction method. The correction was performed using the following equation:

$$p_B = \frac{p}{nc}$$

where nc corresponded to the number of comparisons.

The hypotheses for comparing two independent algorithms were:

- $H_o : u1 = u2$ (means of the two algorithms were equal), this was the null hypothesis.

- $H_a : u1 \neq u2$ (means of the two group were not equal), this was the alternative hypothesis.

The statistical tests yield the following results: if the p-value provided by the statistical test was smaller than the critical value (p_B), there was evidence to reject the null hypothesis in favor of the alternative. In other words, there was evidence that the means were significantly different at the significance level reported by the corrected p-value. Otherwise, there was not enough evidence to reject the null hypothesis, and we concluded that there was evidence that the means were not significantly different.

In the statistical tables presented in the next chapters, each line compares a pair of algorithms using the notation "++" or "$--$", when the first algorithm is significantly better than, or significantly worse than the second one, respectively. The use of "-" or "+" indicates that the first algorithm is better than, or worse than the second, respectively, but without statistical evidence.

Chapter 9

Memory: Experimental Results

This chapter sets forth the results concerning the solutions for memory management problems, i.e., the memory (and population) size and the replacing methods. First, we analyze the influence that different settings of population and memory sizes have in different types of dynamic environments. Different scenarios, using constant population and memory sizes, are analyzed for the three algorithms described in chapter 5. The three algorithms with constant populations are compared with VMEA that uses the same global number of individuals, but with variable proportions.
Second, we study the performance of the different memory-based EAs, using different replacing schemes for updating the memory.

9.1 Population and memory sizes

In this section we analyze the results concerning the influence that different population and memory sizes can have in memory-based EAs for cyclic dynamic environments. No comparisons, between the algorithms that use population and memory of constant sizes, were made. These memory-based EAs were already studied and compared in many different works [90, 114, 117, 121, 123]. Our goal was to see if, for each one of the studied algorithms, different population and memory sizes lead to different performances. Each algorithm, using constant values for the population and memory sizes, was compared with the **VMEA**, which used variable sizes for population and memory. In all the situations, the global number of individuals (n) was kept constant.

9.1.1 Analysis of the results for MEGA

The results obtained with **MEGA** for the four studied benchmarks, using different sets for the population and memory sizes, are shown on Figure 9.1 through Figure 9.4. Each figure corresponds to a different problem (Knapsack, Onemax, Royal Road F1 and Royal Road F2).

The statistical results are in Table 9.1; they compare the best and worst choices for population and memory sizes for the **MEGA** ($MEGA_b$ and $MEGA_w$, respectively), and the best results obtained with **MEGA** compared with **VMEA**. $MEGA_b$ and $MEGA_w$ corresponded to the best and the worst overall averages, respectively, chosen from all the studied cases.
The results obtained show that different choices for the population and memory sizes had a great impact on **MEGA**'s performance. For the Knapsack problem, memory sizes of $m = 60\% \times n$ or $m = 70\% \times n$ achieved the best results; the worst results were observed using smaller memories (10% or 20% of n). In general, larger memory size improved **MEGA**'s performance. The worst results were observed using populations with 90 or 10 individuals. For larger change periods, the influence of the population/memory sizes on the algorithm's performance became smaller, since the algorithm had more time to evolve and to find the best solution.
For the Onemax problem, the conclusions were similar: the best results were achieved using memory proportions of 50% to 70% of n. The worst performance was obtained with smaller memory (10% of n). The results were also more stable for larger change periods where the influence of the number of individuals was not so evident. Analyzing the results obtained in the Royal Road functions F1 and F2, we conclude that the best choices for the proportion of memory size ranged between 30% to 60% of n. The worst results were obtained using larger memory size. When the change period increased, the variations on the population/memory sizes had less impact on the algorithm's performance. **VMEA** significantly outperformed **MEGA** in most situations as the statistical tests confirmed. This was a consistent observation for all the problems and all situations. Only a few exceptions were observed on the Royal Road functions: for larger change periods, the **VMEA** was better than **MEGA**, but the difference was not statistically significant.

		Knapsack				Onemax				Royal Road F1				Royal Road F2			
		ρ															
T-test results	r	0.1	0.2	0.5	1.0	0.1	0.2	0.5	1.0	0.1	0.2	0.5	1.0	0.1	0.2	0.5	1.0
$MEGA_b - MEGA_w$	10	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++
$VMEA - MEGA_b$		++	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++
$MEGA_b - MEGA_w$	50	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++
$VMEA - MEGA_b$		++	++	++	++	++	++	++	++	+	++	++	++	++	++	++	++
$MEGA_b - MEGA_w$	100	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++
$VMEA - MEGA_b$		++	++	++	++	++	++	++	++	+	+	+	+	+	+	+	+
$MEGA_b - MEGA_w$	200	++	++	++	++	++	++	++	+	++	++	++	++	++	++	++	++
$VMEA - MEGA_b$		++	++	++	++	++	++	++	++	+	+	+	+	+	+	+	+

Table 9.1: Statistical results of comparing **MEGA** and **VMEA**

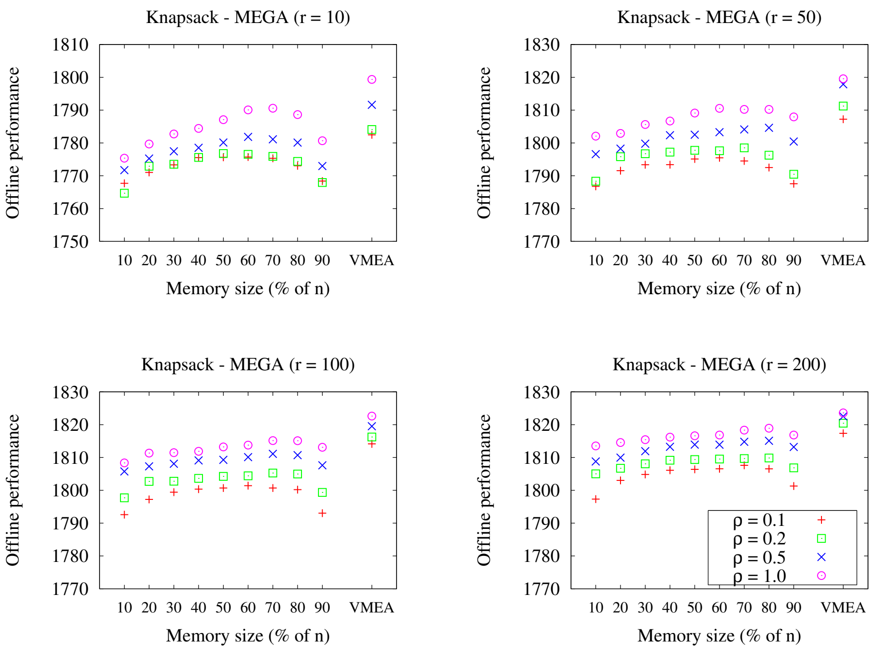


Figure 9.1: Global results obtained in the dynamic **Knapsack** problem using **MEGA** with different population and memory sizes

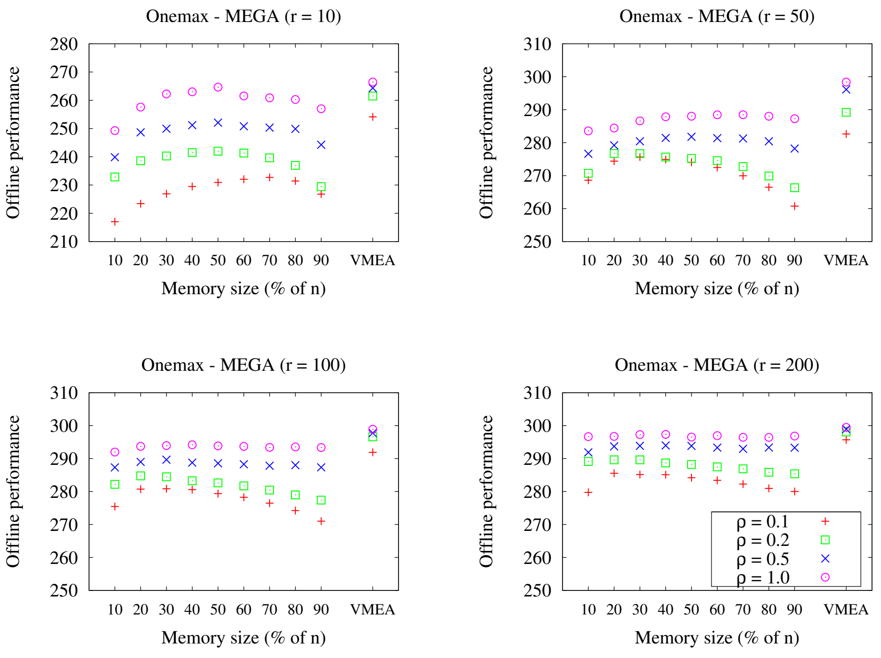


Figure 9.2: Global results obtained in the dynamic **Onemax** problem using **MEGA** with different population and memory sizes

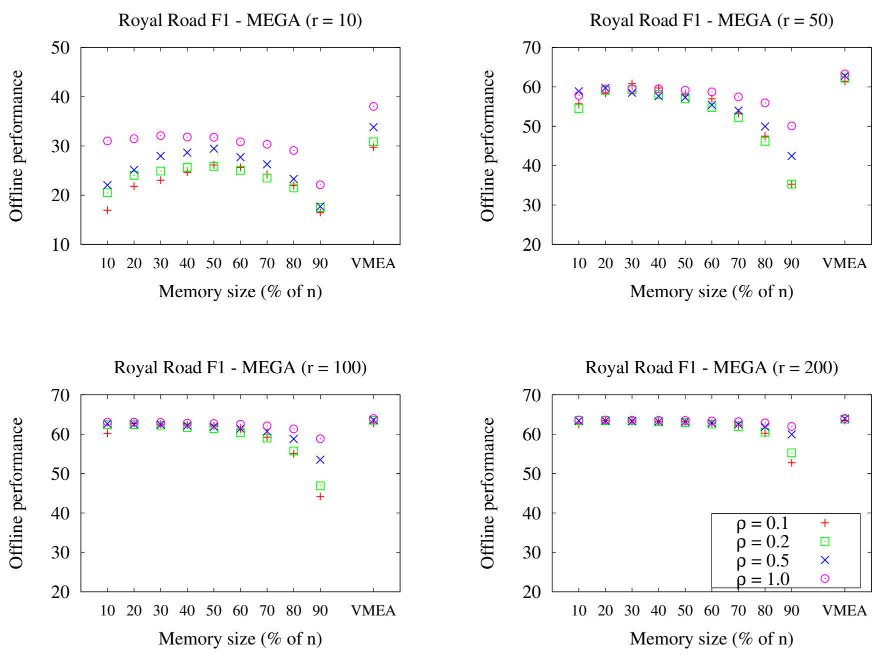


Figure 9.3: Global results obtained in the dynamic **Royal Road F1** problem using **MEGA** with different population and memory sizes

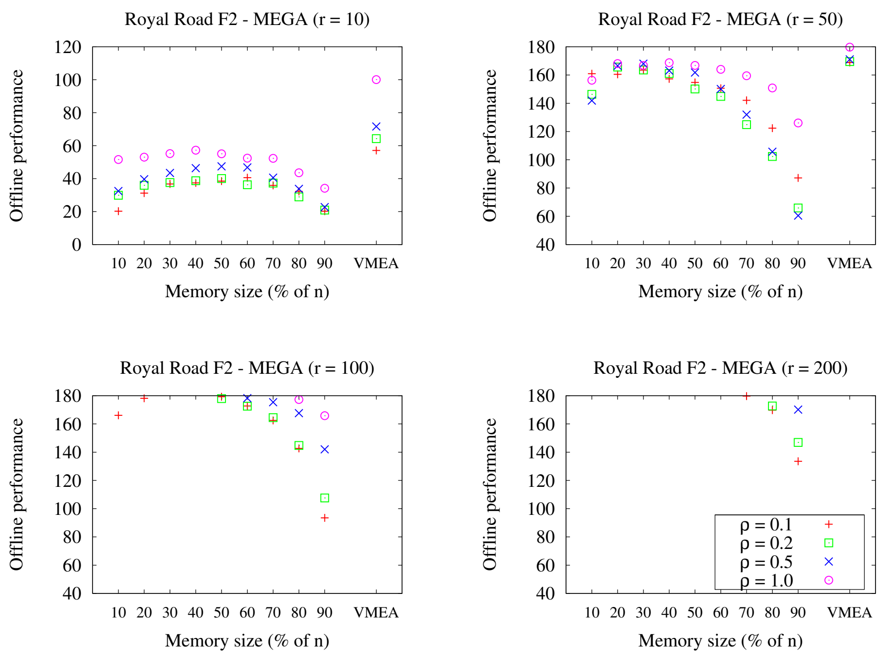


Figure 9.4: Global results obtained in the dynamic **Royal Road F2** problem using **MEGA** with different population and memory sizes

9.1.2 Analysis of the results for MIGA

Figures 9.5 through 9.8 set forth the results for the four studied problems using **MIGA** (with different population and memory sizes) and **VMEA**.
The first plot refers to the Knapsack problem and reveals that, when the environment changed more rapidly ($r = 10$), the best results were obtained using a memory size equal to $m = 30\% \times n$ to $m = 60\% \times n$, depending on the severity of the change. When the severity was higher ($\rho = 0.5$ and $\rho = 1.0$), the best choice corresponded to $m = 50\% \times n$ or $m = 60\% \times n$. For the remaining cases, smaller memories (30% or 40% of n) allowed the algorithm to get the best solutions. For $r = 50$, $r = 100$ and $r = 200$ the best results were achieved using $m = 20\% \times n$ or $m = 30\% \times n$. The worst choices for the population and memory sizes, in general, were those corresponding to the extremes: large memory (90%) or small memory (10%).
Analyzing the results for the Onemax problem, similar conclusions are drawn: rapid environments ($r = 10$) required smaller populations (and larger memory size) than slower changing environments ($r = 50$, $r = 100$ and $r = 200$). In fact, for the first situation the best results corresponded to a memory size of $m = 60\% \times n$. For the remaining cases, the highest marks were obtained using smaller memory sizes (20% or 30% of n). Once more, the use of $m = 90\% \times n$ was the worst option for **MIGA**.
Analyzing the results of the Royal Road functions F1 and F2, we discern the same behavior observed on the previously discussed problems. In fact, Figures 9.7 and 9.8 show that **MIGA** obtained the best results using a memory equal to $20\% \times n$ and the performance of the algorithm decreased as the population became smaller. The worst marks were obtained with memory with $90\% \times n$ individuals.
Analyzing the results and the statistical information presented on Table 9.2, we conclude that **VMEA** performed significantly better than **MIGA**. Few exceptions were found for the Onemax problem: **VMEA** performed better than **MIGA**, but the difference was not statistically significant.

		Knapsack				Onemax				Royal Road F1				Royal Road F2			
		ρ															
T-test results	r	0.1	0.2	0.5	1.0	0.1	0.2	0.5	1.0	0.1	0.2	0.5	1.0	0.1	0.2	0.5	1.0
$MIGA_b - MIGA_w$	10	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++
$VMEA - MIGA_b$		++	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++
$MIGA_b - MIGA_w$	50	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++
$VMEA - MIGA_b$		++	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++
$MIGA_b - MIGA_w$	100	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++
$VMEA - MIGA_b$		++	++	++	++	++	++	+	+	++	++	++	++	++	++	++	++
$MIGA_b - MIGA_w$	200	++	++	++	++	++	++	++	+	++	++	++	++	++	++	++	++
$VMEA - MIGA_b$		++	++	++	++	+	+	+	+	++	++	++	++	++	++	++	++

Table 9.2: Statistical results of comparing **MIGA** and **VMEA**

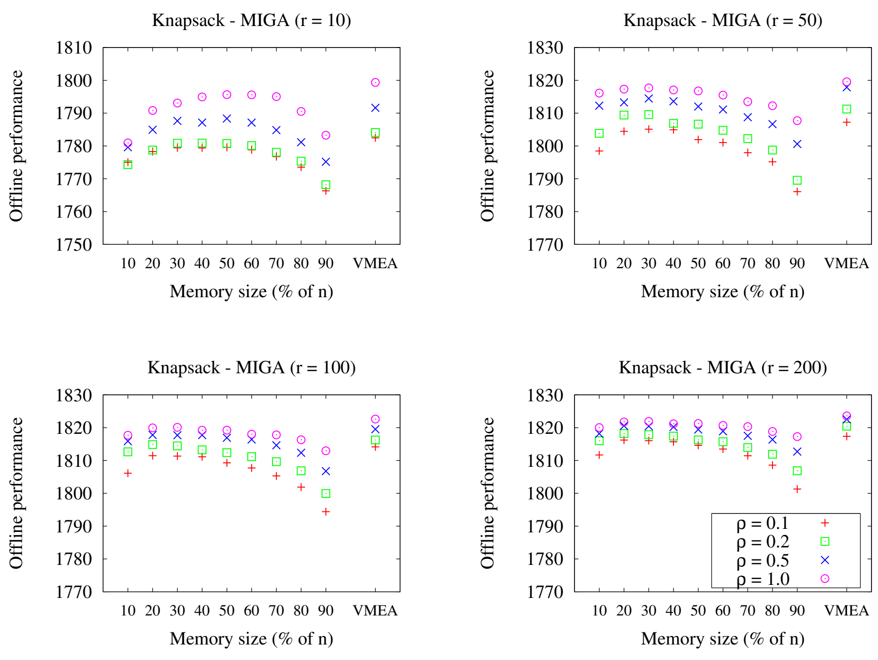


Figure 9.5: Global results obtained in the dynamic **Knapsack** problem using **MIGA** with different population and memory sizes

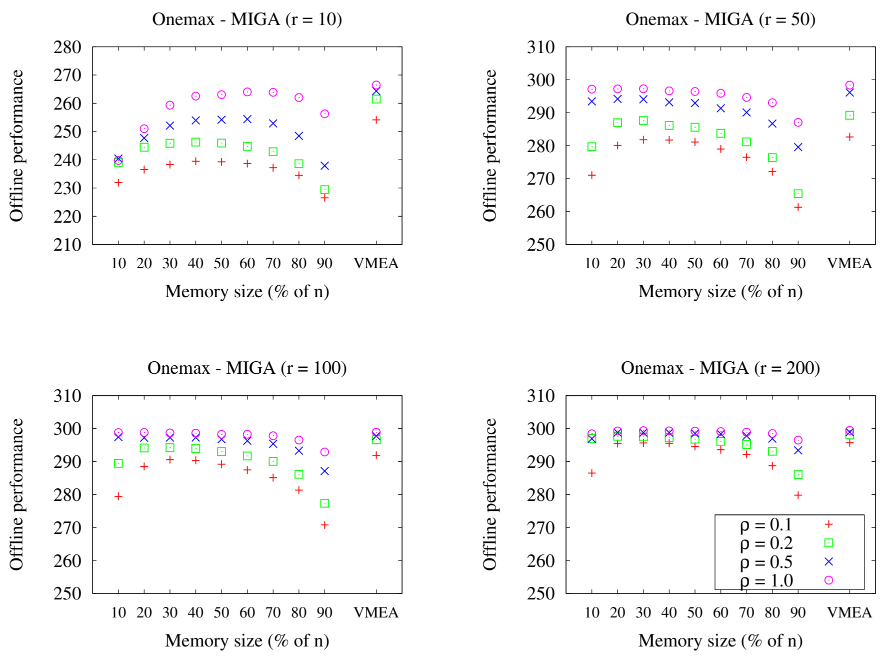


Figure 9.6: Global results obtained in the dynamic **Onemax** problem using **MIGA** with different population and memory sizes

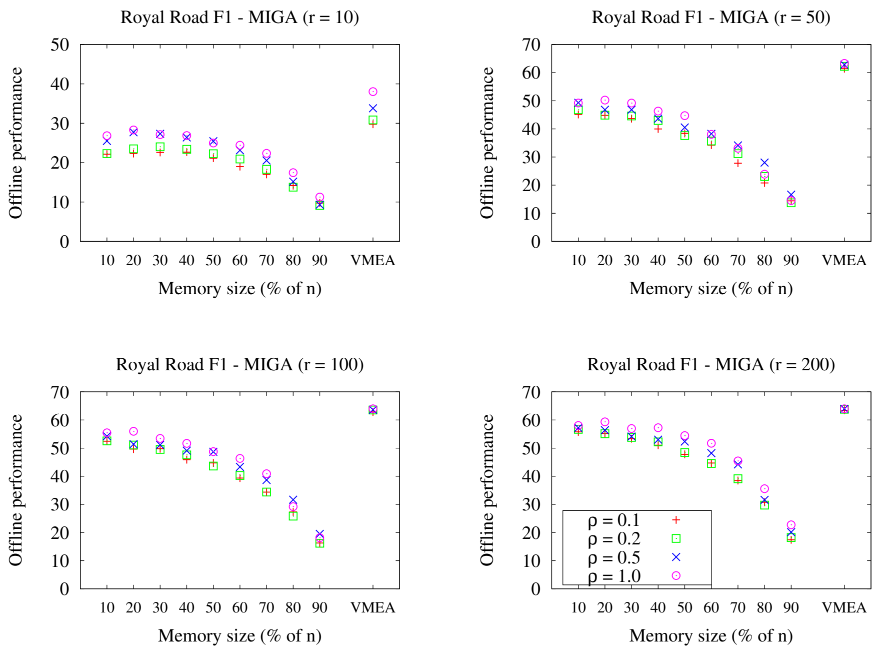


Figure 9.7: Global results obtained in the dynamic **Royal Road F1** problem using **MIGA** with different population and memory sizes

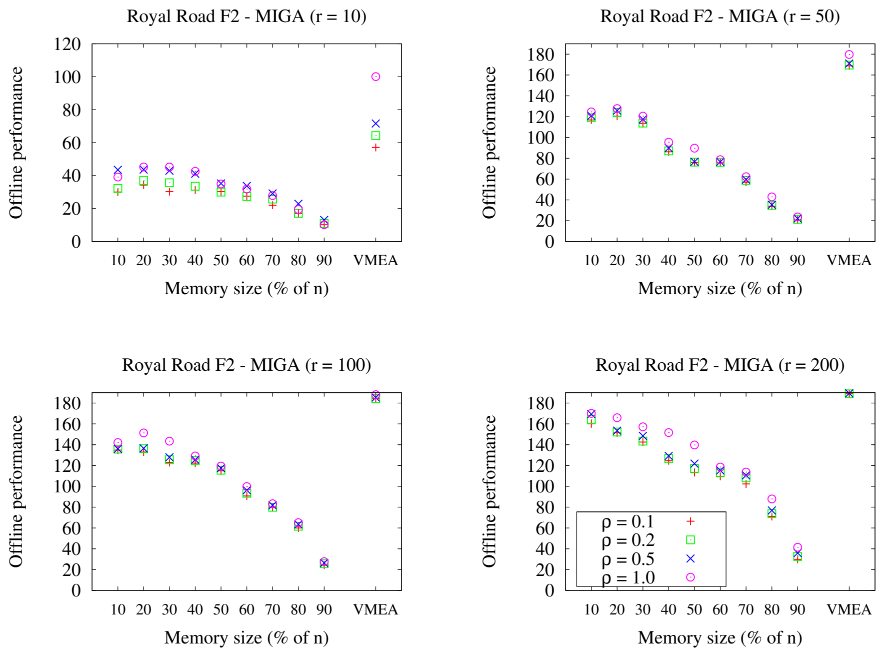


Figure 9.8: Global results obtained in the dynamic **Royal Road F2** problem using **MIGA** with different population and memory sizes

9.1.3 Analysis of the results for AMGA

The global results obtained for the **AMGA**, using different proportions for the memory size, are displayed in Figure 9.9 through Figure 9.12.
The statistical results shown on Table 9.3 compare the best and worst choices using different memory sizes for the **AMGA** ($AMGA_b$ and $AMGA_w$, respectively); the best results of the **AMGA** are compared with **VMEA**.
As observed, depending on the change period and the change ratio, the choice of population and memory sizes affected the algorithm's performance.
For the Knapsack problem, in environments with slower changes (r = 50, r = 100, r = 200), the chosen proportion for the memory size had a larger impact in the algorithm's performance. On those situations, the best results were achieved using $m = 70\% \times n$ and $m = 60\% \times n$. This observation was consistent for all the values of ρ. The worst results were observed using larger memory sizes ($m = 80\% \times n$ and $m = 90\% \times n$). When $r = 10$ **AMGA** obtained similar results using memory sizes from 10% to 70% of n. As before, the use of smaller populations ($m = 90\% \times n$) lead to a significant decrease in the algorithm's performance.

For the Onemax problem, **AMGA**, for $r = 10$ and $r = 50$, obtained better results using a memory size of 50% or 60% of n. On those situations, either smaller or larger populations lead to a decrease of the EAs' performance. For $r = 100$ and $r = 200$, **AMGA** achieved the best results using larger memory (90%) and smaller populations. In general, for those situations, the algorithm's performance increased using larger memory sizes.

The performance of the **AMGA** in the Royal Road F1 and F2 problems was similar in environments with slower changes: using $r = 50$, $r = 100$ and $r = 200$. In these cases, the best results obtained with the **AMGA** were reported with memory of 10% or 80% of the global number of individuals. On those situations, smaller memory sizes were enough to obtain the best results. In addition, as the change period increased, the impact of the population size on the algorithm's performance was smaller, and only with smaller populations was the algorithm's performance seriously affected.
When $r = 10$, the **AMGA** for the Royal Road F1 obtained the best results using $m = 20\% \times n$ or $m = 30\% \times n$, in environments with lower severity of change ($\rho = 0.1$, $\rho = 0.2$ and $\rho = 0.5$), or larger memory ($m = 70\% \times n$) in environments with higher severity ($\rho = 1.0$). In all the cases, either larger (90%) or smaller (10%) memory size presented the worst results.
When $r = 10$, the results for **AMGA** for the Royal Road F2 were similar to Royal Road F1. The algorithm obtained the best results using $m = 30\% \times n$ or $m = 40\% \times n$ in environments with lower severity of change ($\rho = 0.1$, $\rho = 0.2$ and $\rho = 0.5$), or smaller populations ($m = 60\% \times n$) in environments with higher severity ($\rho = 1.0$). In all the cases, either larger or smaller populations presented the worst results.

Analyzing all the situations studied for the **AMGA**, no general behavior was found. The impact of the population and memory sizes was significant but de-

pended on the problem and on the environmental characteristics.
Table 9.3 indicates that a *good* or a *bad* choice for the population and memory sizes had a significant impact on the **AMGA**'s performance. Comparing the **VMEA**'s and the **AMGA**'s performance in all situations, **VMEA** always outperformed **AMGA**. Table 9.3 corroborates this observation. In general, as the change period increased, the global performance of **VMEA** became more robust and was similar for all values of ρ.

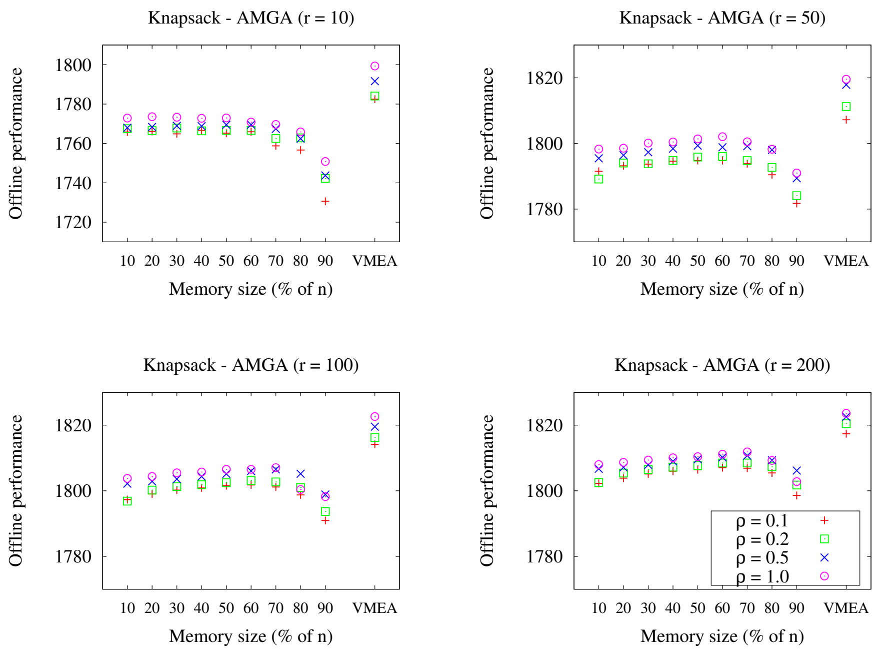


Figure 9.9: Global results obtained in the dynamic **Knapsack** problem using **AMGA** with different population and memory sizes

		Knapsack				Onemax				Royal Road F1				Royal Road F2			
		ρ															
T-test results	r	0.1	0.2	0.5	1.0	0.1	0.2	0.5	1.0	0.1	0.2	0.5	1.0	0.1	0.2	0.5	1.0
$AMGA_b - AMGA_w$	10	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++
$VMEA - AMGA_b$		++	++	++	++	++	++	++	++	+	+	+	++	++	+	+	+
$AMGA_b - AMGA_w$	50	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++
$VMEA - AMGA_b$		++	++	++	++	++	++	++	++	+	++	++	++	++	++	+	+
$AMGA_b - AMGA_w$	100	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++
$VMEA - AMGA_b$		++	++	++	++	++	++	++	++	+	++	++	++	++	++	++	++
$AMGA_b - AMGA_w$	200	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++
$VMEA - AMGA_b$		++	++	++	++	++	++	++	++	+	+	+	+	++	++	++	++

Table 9.3: Statistical results of comparing **AMGA** and **VMEA**

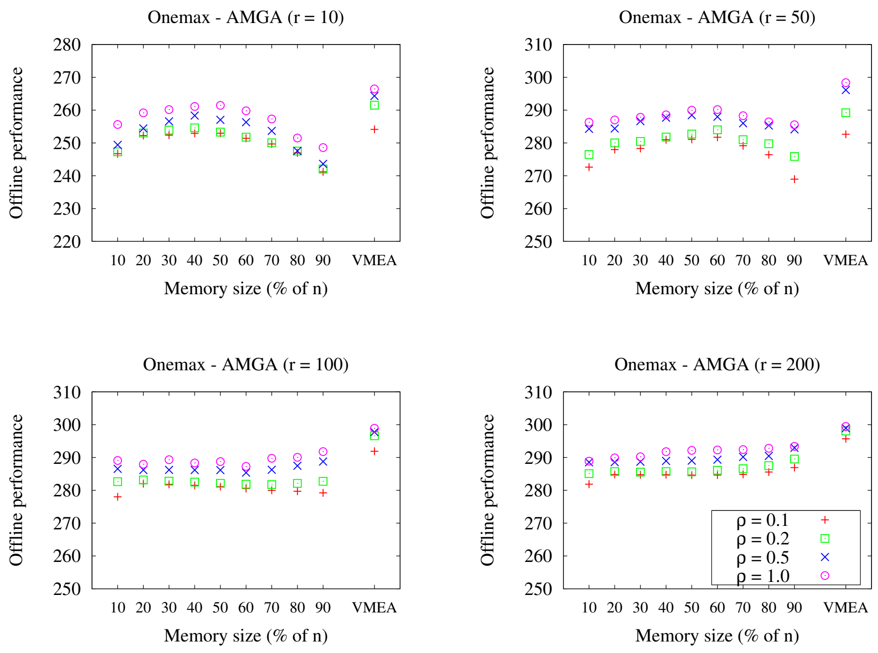


Figure 9.10: Global results obtained in the dynamic **Onemax** problem using **AMGA** with different population and memory sizes

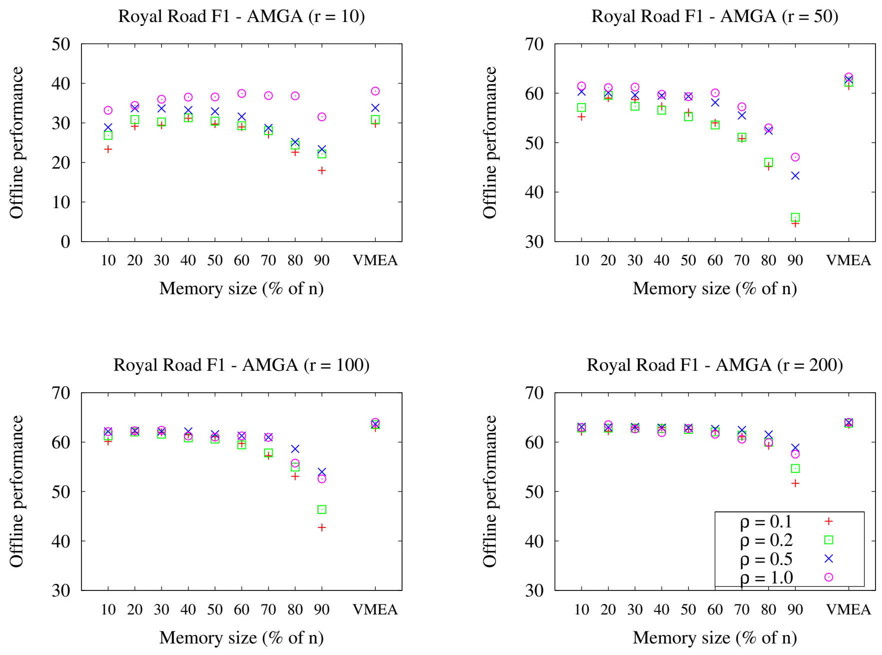


Figure 9.11: Global results obtained in the dynamic **Royal Road F1** problem using **AMGA** with different population and memory sizes

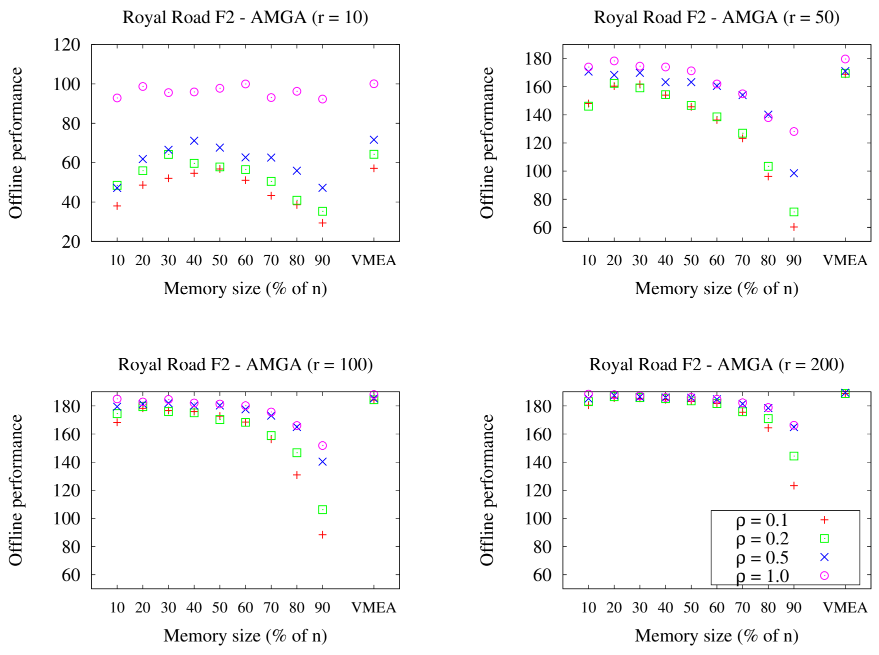


Figure 9.12: Global results obtained in the dynamic **Royal Road F2** problem using **AMGA** with different population and memory sizes

9.1.4 Analysis of the VMEA sizes

From the analysis of the results, it is evident that **VMEA** achieved significant better performances compared to all the other algorithms. Using constant sizes for **MEGA**, **MIGA** or **AMGA**, different settings yield to different performances and no typical values were found as the optimal choices for the population/memory sizes. On the other hand, **VMEA**, using different population and memory sizes during the run, was able to achieve the best scores. In order to better understand **VMEA**'s better performance, we analyzed the variations of population and memory sizes in this algorithm.
Figure 9.13 shows the population sizes achieved at the end of the run. Note that the global number of individuals (n) was the same in all the algorithms; for example, if p was set to 90, the corresponding memory size was 10. Using the information provided by the plots we can establish some general conclusions:

(a) **VMEA** evolved larger population sizes (and smaller memory) for environments using larger change periods. The only observed exception corresponded to the Onemax problem for $\rho = 0.1$, where the population was almost the same for every values of r.

(b) when the severity of the change was higher, larger population sizes were attained.

Comparing the settings that allowed the other peer algorithms to achieve the best results, it is not possible to say that the same conclusions were observed in

all circumstances. Nevertheless, in most situations, the above described pattern was present. We can see that for the Royal Road Functions F1 and F2, the three algorithms had results in accordance with the general principles stated above. As for the remaining problems, similar observations were generally present for **MEGA** and **MIGA**, but the same was not true for **AMGA**.
It is important to point out that the values presented on Figure 9.13 are the final sizes for the population reached at the end of the run using **VMEA**. So, it is understandable that the results cannot be exactly the same obtained by the other algorithms. In fact, **VMEA** changed the population and memory sizes during the run in different ways, depending on the change period. Figure 9.14 represents a typical example, obtained for the Onemax problem for $r = 10$ and $r = 200$.
We can see that, for both situations, there was a consistent pattern: as time passed, the population size tended to decrease as the memory size became larger. Also, in rapid changing periods ($r = 10$) those variations occurred faster. When $r = 200$, the decrease of the population size was slower. This happened because we used a replacing strategy that replaced the memory individuals belonging to the same cycle. Consequently, in larger cycles, there was no need to increase the memory size as often because a previously stored individual, belonging to the same period, was being replaced. For rapid change periods, this situation was not so frequent and the memory size increased faster to store a wider variety of individuals.

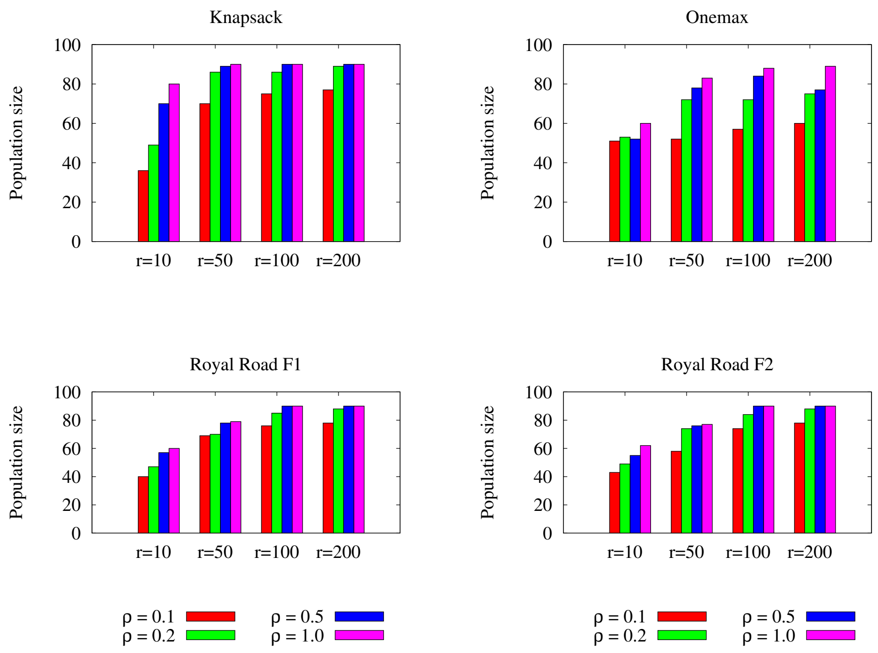


Figure 9.13: Population sizes at the end of the run obtained using **VMEA**

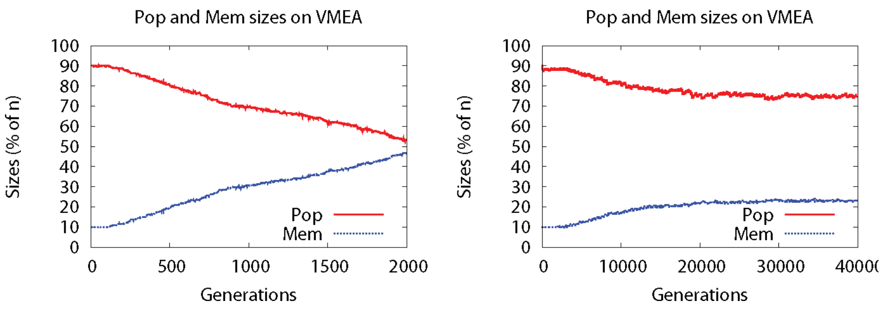


Figure 9.14: Population sizes during the run obtained using **VMEA** for the Onemax problem using $r = 10$ (figure on the left) and $r = 200$ (figure on the right) and $\rho = 0.5$

9.1.5 Discussion

The results obtained show that the use of different values for the population and memory sizes can have a significant influence in the efficacy of the EAs. The values used as standard for population and memory sizes, corresponding to larger populations and smaller memories (10% of n), rarely achieved the best results in the cases studied.

The results were not consistent in the three algorithms tested using constant sizes, but some patterns occurred often.

Looking at the population and memory sizes that allowed **MEGA**, **MIGA** and **AMGA** to achieve the best scores, the following was generally observed:

(a) when the changes in the environment (r) occurred faster, it was important to use a larger memory size than for situations where the changes occurred at a slower rate.

(b) as the severity of the change (ρ) increased, the best size for the memory became smaller.

AMGA was the only algorithm where those situations were not so regular.

The first observation can be explained by the memory replacing scheme that was used. Using the *generational* scheme, if an individual of the current period was previously stored in memory, when the memory was updated and the period was the same, one memory individual of the same cycle was replaced. When the change period was larger, this situation occurred often and smaller memory sizes were enough to provide a good performance. The opposite happened when the change period was smaller and the replacing of an individual of the same cycle rarely happened. As a result, the memory capacity was attained faster and larger memory sizes provided better results.

The second conclusion is explained as follows: when the severity of the change was higher, a small number of different environments appeared. For instance, when $\rho = 1.0$ the environment changed between two different states, for $\rho = 0.1$, ten or more different environments appeared during the run. So, for higher values of ρ, less information was needed in the memory to assure a good performance.

The results obtained don't allow us to say which population and memory size is better for a particular problem. The best choice of values depends on the environmental characteristics, the problem to solve, and the algorithm used. Therefore, this choice is not simple or easy and trying to tune the population size before running the algorithm is practically impossible, since the combinations are huge and time consuming.
The proposed **VMEA**, capable of controlling the population and memory sizes during the run, obtained superior results, and proved to be effective and robust in all environments and problems analyzed. This algorithm used the same number of individuals but managed their distribution among the population and memory differently. The results show that, at the end of the run, the algorithm obtained population and memory sizes that were in accordance with the observations (a) and (b) made before. Moreover, the evolution of the population and memory sizes during the run was different for different environments. Without many restrictions, **VMEA** was able to manage the global number of individuals "intelligently" by using population and for the memory proportions that allowed it to obtain significantly better results.

9.2 Replacing strategies

This section presents the results about the impact that different replacing strategies had in the performance of the memory-based EAs. We show the results obtained with Branke's most used replacing method, called *similar*, and compare those results with the replacing strategies proposed in this work previously described in Chapter 5. The four replacing methods - *similar*, *age1*, *age2* and *generational* - were tested and compared on the four implemented EAs: **MEGA**, **MIGA**, **AMGA**, and **VMEA**.

9.2.1 Analysis of the results for MEGA

The results obtained with **MEGA** are reported on Figures 9.15 through 9.18. The statistical results provided by the statistical tests are displayed on Table 9.4.
The results obtained show that, when $r = 10$, the *age2* replacing method allowed the algorithm to achieve better results. The remaining techniques obtained comparable performances.
For the remaining values of r, the results obtained by **MEGA** were as follows: the best results were obtained using the *generational* replacing method and *age1* achieved the worst results; *age2* was better than *age1* for $r = 50$ and $r = 100$ and had equivalent performances when $r = 200$.

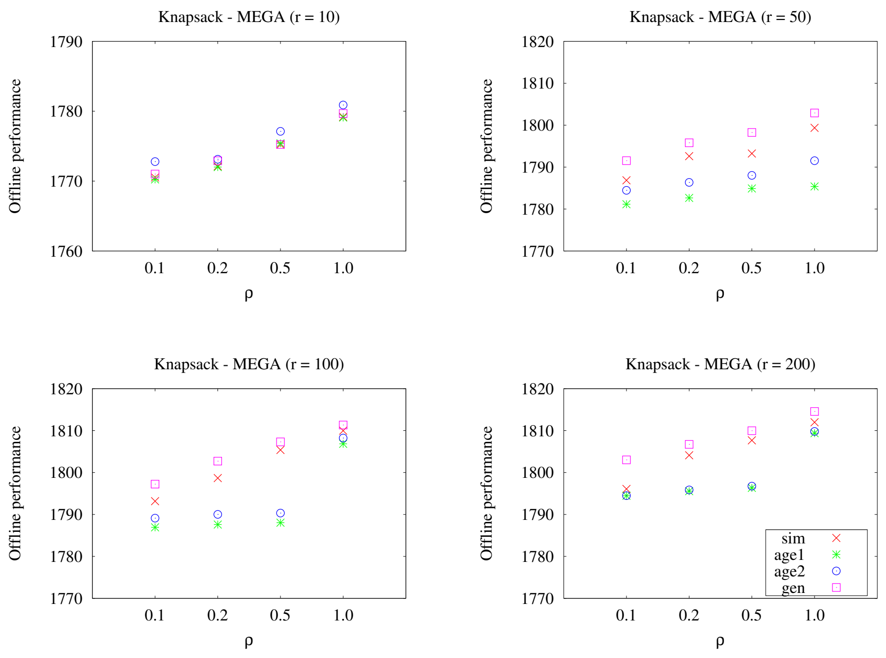


Figure 9.15: Global results obtained in the dynamic **Knapsack** problem using **MEGA** with different replacing strategies

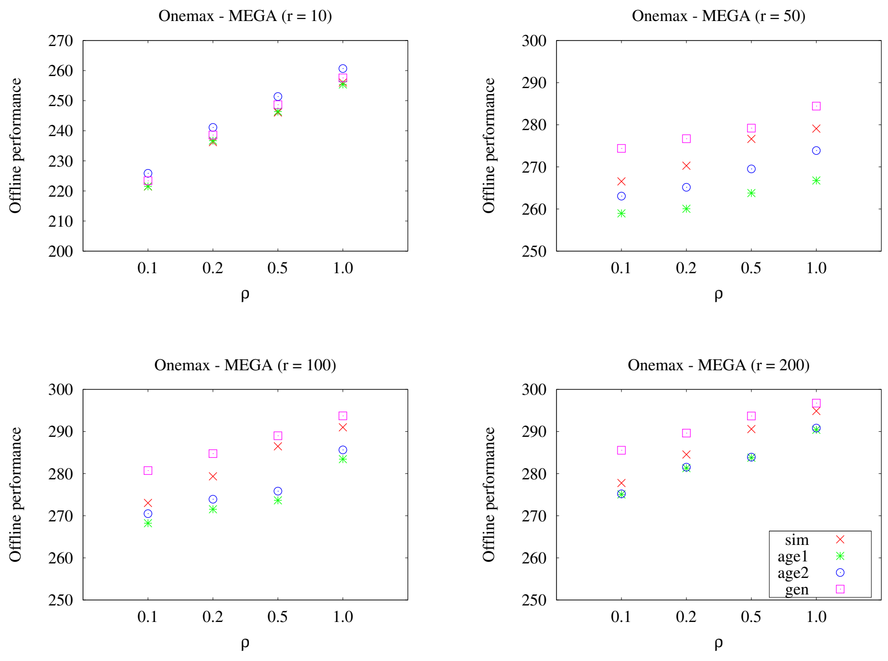


Figure 9.16: Global results obtained in the dynamic **Onemax** problem using **MEGA** with different replacing strategies

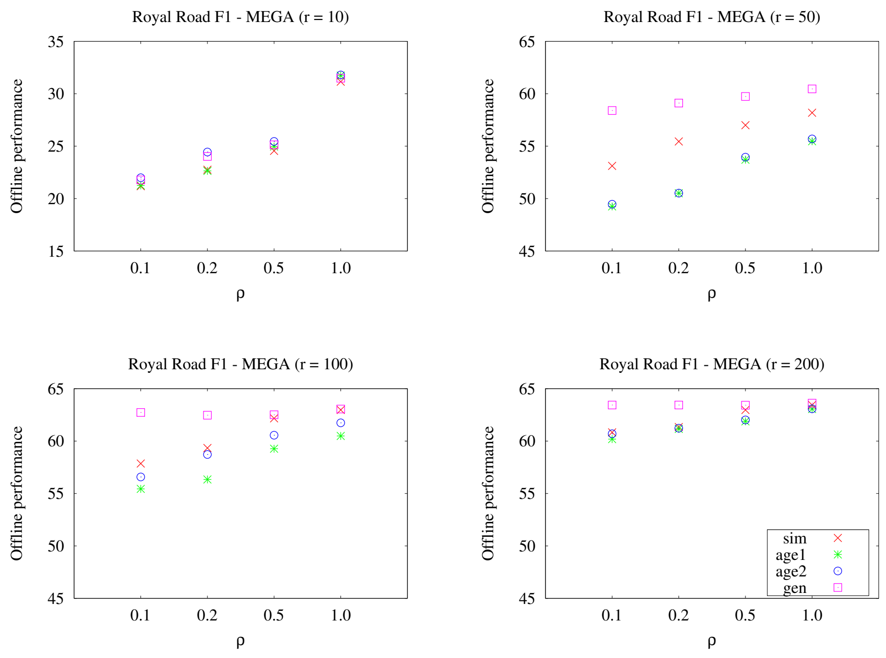


Figure 9.17: Global results obtained in the dynamic **Royal Road F1** problem using **MEGA** with different replacing strategies

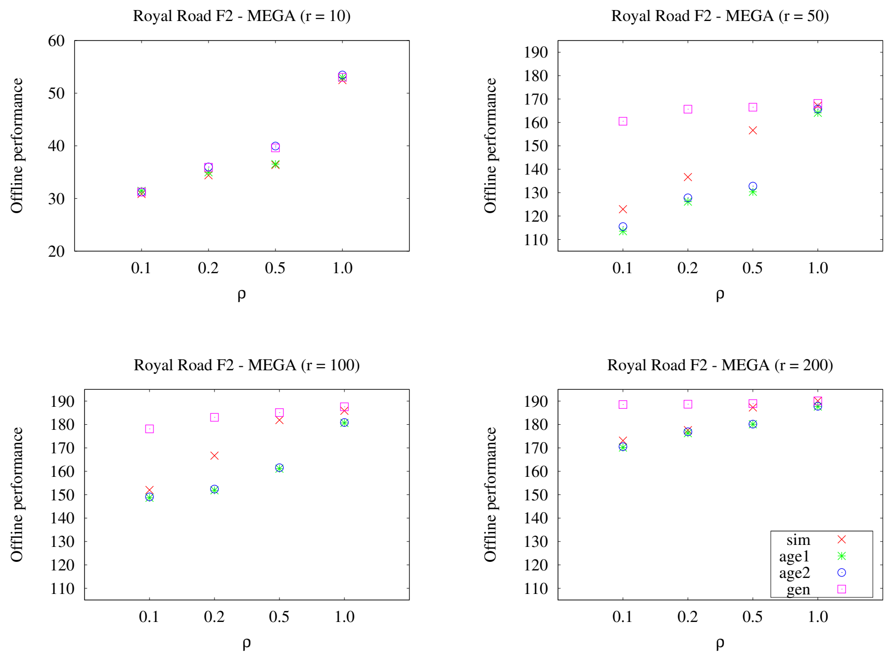


Figure 9.18: Global results obtained in the dynamic **Royal Road F2** problem using **MEGA** with different replacing strategies

		Knapsack				Onemax				Royal Road F1				Royal Road F2			
		ρ															
T-test results	r	0.1	0.2	0.5	1.0	0.1	0.2	0.5	1.0	0.1	0.2	0.5	1.0	0.1	0.2	0.5	1.0
gen – age1		+	+	−	+	++	++	++	++	+	++	+	+	−	−	+	−
gen – age2		−−	−−	−−	−−	−−	−−	−−	−−	−	−	−	−	−	−	−	−
gen – sim	10	+	+	+	+	++	++	++	+	+	++	++	+	++	++	++	++
age1 – age2		−−	−−	−−	−−	−−	−−	−−	−−	−	−−	−	−	+	−−	−−	−−
age1 – sim		−	−	+	−	+	+	+	−	+	−	+	+	+	+	+	+
age2 – sim		++	++	++	++	++	++	++	++	+	++	++	+	+	++	++	++
gen – age1		++	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++
gen – age2		++	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++
gen – sim	50	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++
age1 – age2		−−	−−	−−	−−	−−	−−	−−	−−	−	−	−	−	−−	−−	−−	−−
age1 – sim		−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−
age2 – sim		−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−
gen – age1		++	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++
gen – age2		++	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++
gen – sim	100	++	++	++	++	++	++	++	++	++	++	+	+	++	++	++	++
age1 – age2		−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−	−−	−	-
age1 – sim		−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−
age2 – sim		−−	−−	−−	−	−−	−−	−−	−−	−−	−−	−−	−	−−	−−	−−	−−
gen – age1		++	++	++	++	++	++	++	++	++	++	++	+	++	++	++	++
gen – age2		++	++	++	++	++	++	++	++	++	++	++	+	++	++	++	++
gen – sim	200	++	++	++	++	++	++	++	++	++	++	++	+	++	++	++	+
age1 – age2		−	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−
age1 – sim		−−	−−	−−	−−	−−	−−	−−	−−	−	−	−	−	−−	−−	−−	−−
age2 – sim		−−	−−	−−	−−	−−	−−	−−	−−	−	−	−	−	−−	−−	−−	−−

Table 9.4: Statistical results of comparing **MEGA** using different replacing strategies

9.2.2 Analysis of the results for MIGA

Figures 9.19 through 9.22 show the comparison results between the different memory replacing techniques on **MIGA**. The statistical validation of the results is reported on Table 9.5.

As happened for **MEGA**, for $r = 10$ the best scores were attained using the *age2* technique and the remaining methods performed equivalently. For larger change periods ($r = 50$, $r = 100$ and $r = 200$), the results were analogous to the ones obtained by the other algorithms: the best technique was the *generational*. The *similar* method conferred better scores than *age1* and *age2*, with *age2* slightly superior to *age1*.

Table 9.5 shows that all these comparisons were statistically significant. Statistical equivalence was observed when $r = 10$, for all the methods except *age2* and, in some comparisons, between *age1* and *age2*, when $r = 200$.

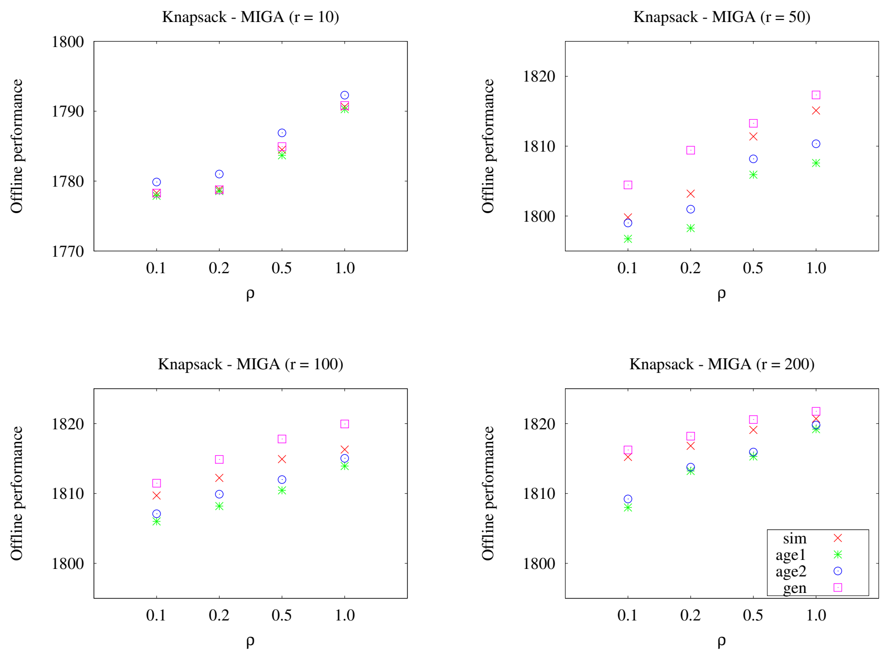


Figure 9.19: Global results obtained in the dynamic **Knapsack** problem using **MIGA** with different replacing strategies

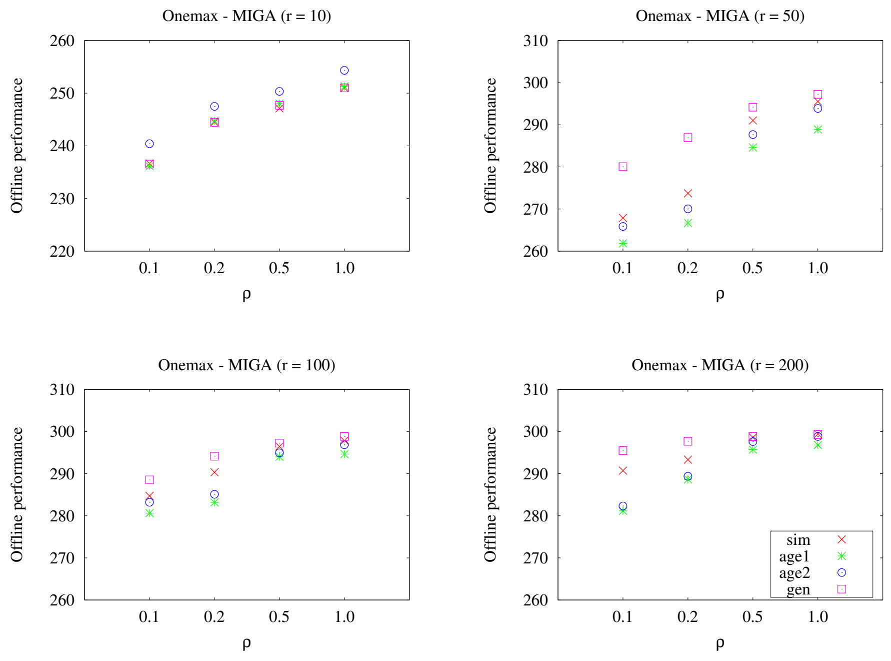


Figure 9.20: Global results obtained in the dynamic **Onemax** problem using **MIGA** with different replacing strategies

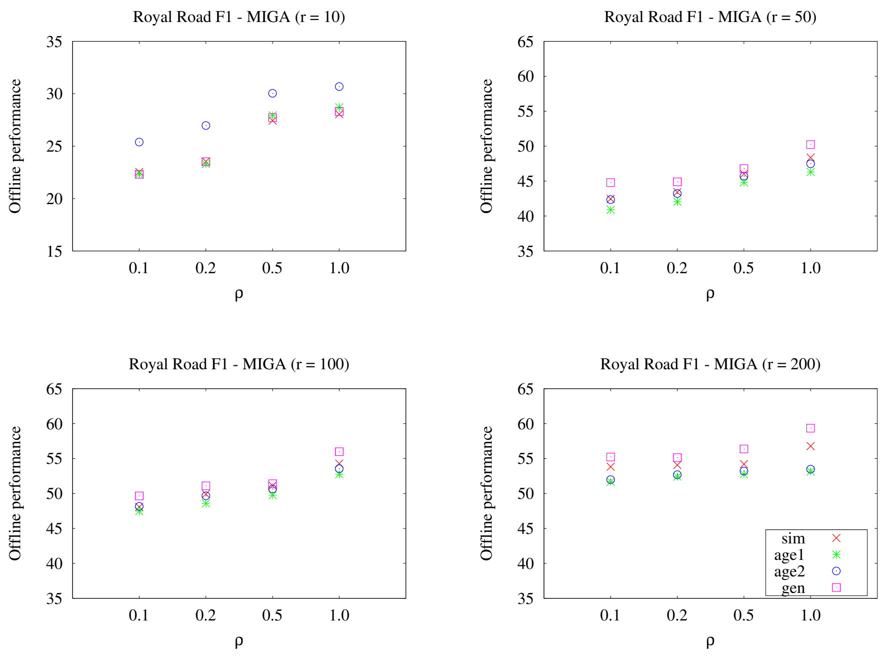


Figure 9.21: Global results obtained in the dynamic **Royal Road F1** problem using **MIGA** with different replacing strategies

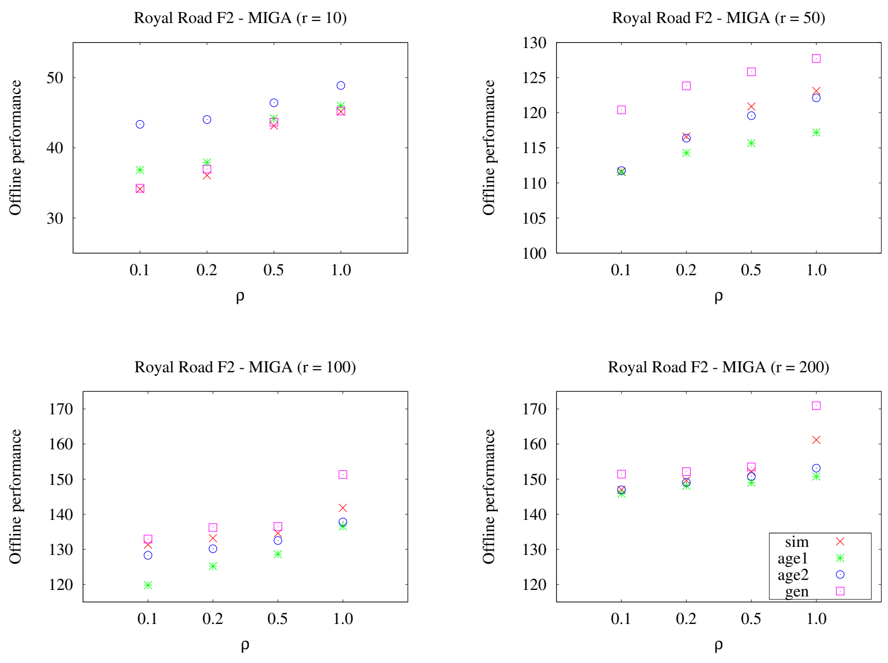


Figure 9.22: Global results obtained in the dynamic **Royal Road F2** problem using **MIGA** with different replacing strategies

		Knapsack				Onemax				Royal Road F1				Royal Road F2			
									ρ								
T-test results	r	0.1	0.2	0.5	1.0	0.1	0.2	0.5	1.0	0.1	0.2	0.5	1.0	0.1	0.2	0.5	1.0
gen – age1		+	+	+	+	+	−	−	−	−	+	−	−	−−	−	−	-
gen – age2		−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−
gen – sim	10	−	−	+	+	−	−	+	+	−	−	+	+	+	+	+	+
age1 – age2		−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−
age1 – sim		−	−	−	−	−	+	+	+	−	−	+	+	++	+	+	+
age2 – sim		++	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++
gen – age1		++	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++
gen – age2		++	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++
gen – sim	50	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++
age1 – age2		−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−	−−	−−	−−
age1 – sim		−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−	−−	−−	−−
age2 – sim		−	−−	−−	−−	−−	−−	−−	−−	−	−	−−	−−	−	−	−−	−−
gen – age1		++	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++
gen – age2		++	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++
gen – sim	100	++	++	++	++	++	++	++	++	++	++	+	++	++	++	++	++
age1 – age2		−−	−−	−−	−−	−−	−−	−	−−	−−	−−	−−	−−	−−	−−	−−	−−
age1 – sim		−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−
age2 – sim		−−	−−	−−	−−	−	−−	−−	−−	−	−	−−	−−	−−	−−	−−	−−
gen – age1		++	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++
gen – age2		++	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++
gen – sim	200	++	++	++	++	++	++	+	+	++	++	++	++	++	++	++	++
age1 – age2		−	−	−	−	−−	−−	−−	−−	−	−	−−	−	−	−	−−	−−
age1 – sim		−−	−−	−−	−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−
age2 – sim		−−	−−	−−	−	−−	−−	−−	−	−−	−−	−−	−−	−	−	−−	−−

Table 9.5: Statistical results of comparing **MIGA** using different replacing strategies

9.2.3 Analysis of the results for AMGA

Figures 9.23 through 9.26 show the results of the comparison between the memory replacing techniques used on the four problems studied. Table 9.6 shows the statistical validation of those comparisons.

When $r = 10$, for all the problems, **AMGA** obtained the best results using the *age2* method. As the plots and the first rows of Table 9.6 indicate, the remaining methods allowed the algorithm to obtain equivalent performances.

When the change period was larger ($r = 50$, $r = 100$ and $r = 200$), the *generational* technique obtained the highest scores and the worst results were achieved using the *age1* technique. For $r = 200$, *age1* and *age2*, methods presented equivalent performances.

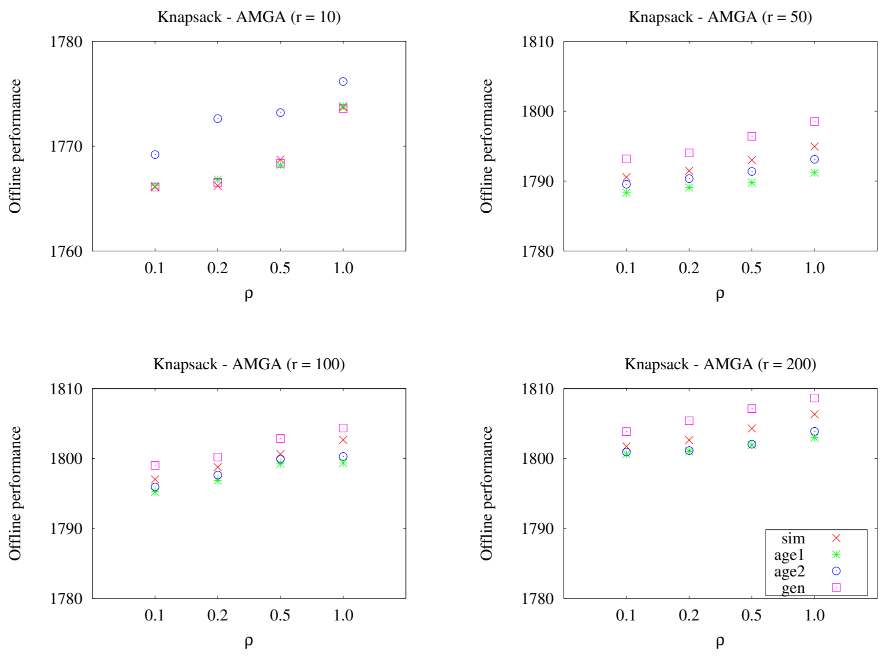


Figure 9.23: Global results obtained in the dynamic **Knapsack** problem using **AMGA** with different replacing strategies

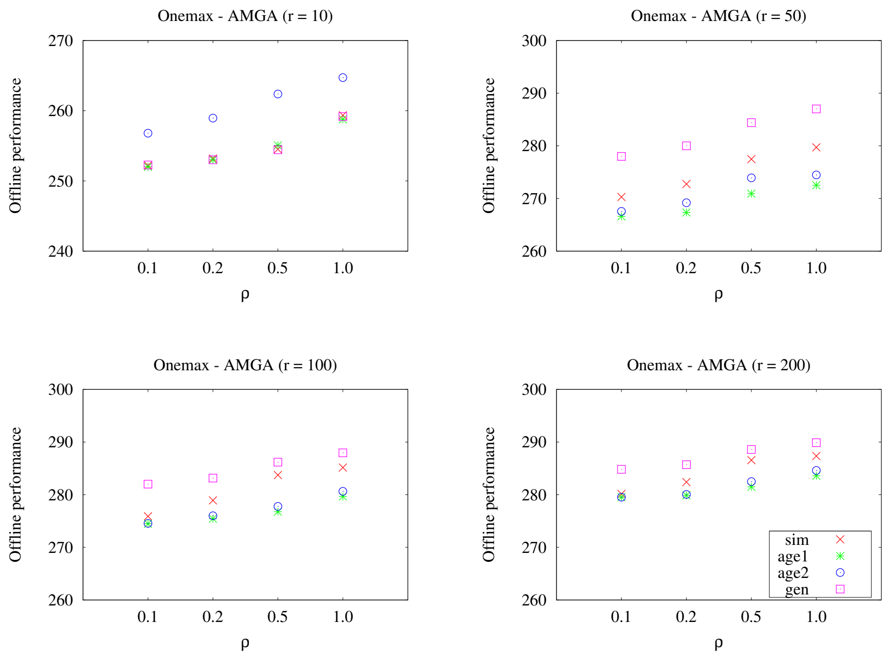


Figure 9.24: Global results obtained in the dynamic **Onemax** problem using **AMGA** with different replacing strategies

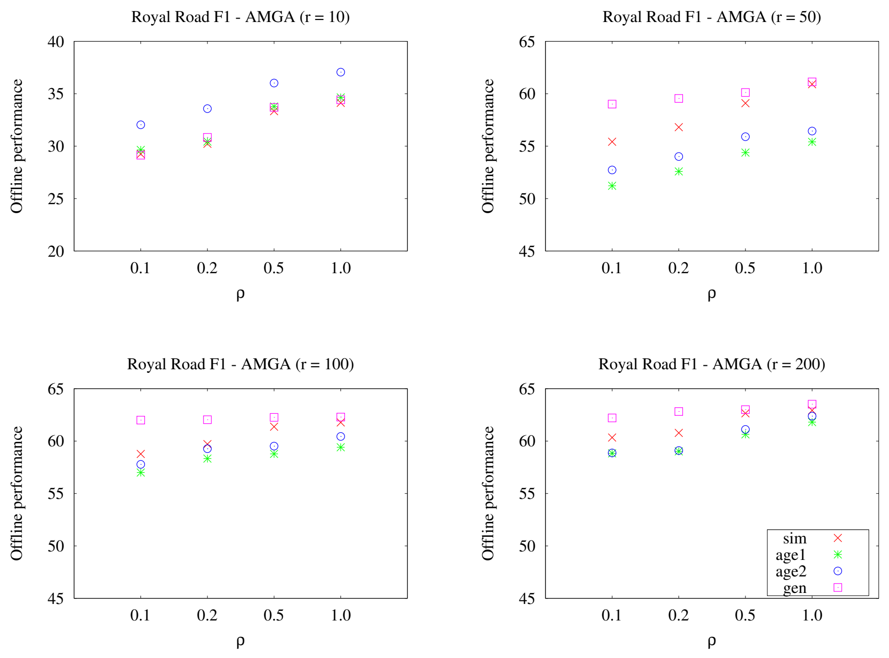


Figure 9.25: Global results obtained in the dynamic **Royal Road F1** problem using **AMGA** with different replacing strategies

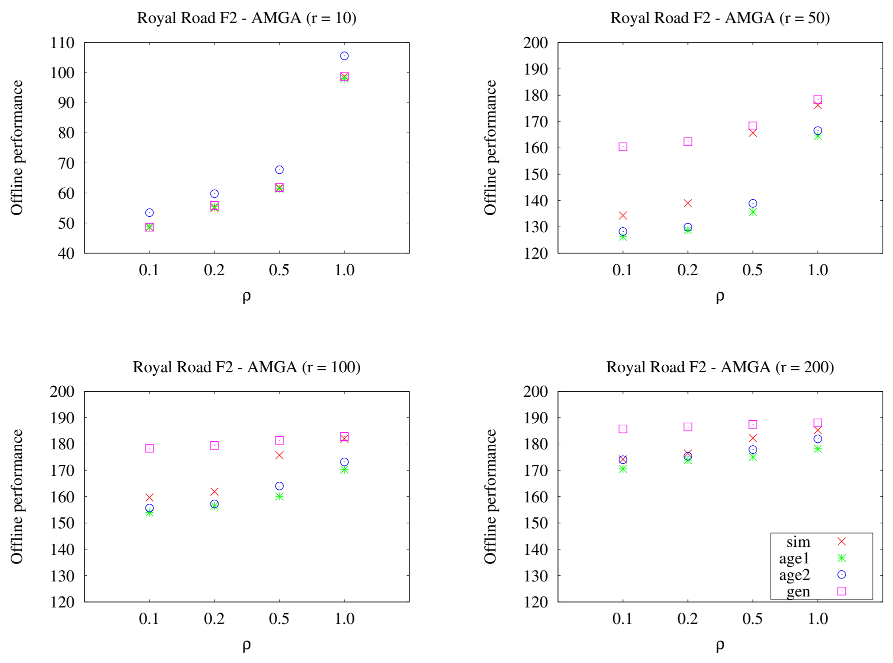


Figure 9.26: Global results obtained in the dynamic **Royal Road F2** problem using **AMGA** with different replacing strategies

		Knapsack				Onemax				Royal Road F1				Royal Road F2			
		ρ															
T-test results	r	0.1	0.2	0.5	1.0	0.1	0.2	0.5	1.0	0.1	0.2	0.5	1.0	0.1	0.2	0.5	1.0
gen – age1		−	−	+	−	+	−	−	−	−	+	+	+	−	+	+	+
gen – age2		−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−
gen – sim	10	+	+	−	−	+	−	−	−	−	+	+	+	+	+	+	−
age1 – age2		−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−
age1 – sim		+	+	−	+	−	−	+	−	+	+	+	+	+	+	−	−
age2 – sim		++	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++
gen – age1		++	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++
gen – age2		++	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++
gen – sim	50	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++
age1 – age2		−	−	−−	−−	−−	−−	−−	−−	−−	−−	−	−−	−−	−−	−−	−−
age1 – sim		−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−
age2 – sim		−−	−−	−−	−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−
gen – age1		++	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++
gen – age2		++	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++
gen – sim	100	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++
age1 – age2		−	−−	−	−	−	−−	−−	−	−	−−	−−	−−	−−	−−	−−	−−
age1 – sim		−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−
age2 – sim		−−	−	−−	−−	−−	−−	−−	−−	−−	−	−−	−−	−−	−−	−−	−−
gen – age1		++	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++
gen – age2		++	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++
gen – sim	200	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++
age1 – age2		−	−	−	−−	−	−	−	−−	−	−	−−	−−	−−	−−	−−	−−
age1 – sim		−−	−−	−−	−−	−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−
age2 – sim		−−	−−	−−	−−	−	−−	−−	−−	−−	−−	−−	−	−	−	−−	−−

Table 9.6: Statistical results of comparing **AMGA** using different replacing strategies

9.2.4 Analysis of the results for VMEA

The results obtained with the different replacing strategies in **VMEA** are shown in Figures 9.27 through 9.30. The statistical validation of the comparison between the different methods is shown in Table 9.7.

The results obtained by the proposed memory replacing techniques in **VMEA** were consistent with the previous algorithms. For rapid changing periods ($r = 10$) **VMEA** achieved the best results using the *age2* scheme. The remaining methods attained equivalent performances.

When the change period was larger ($r = 50$, $r = 100$ and $r = 200$) the best results were reached using the *generational* method. In general, the *similar* method performed better when the severity of the change was higher ($\rho = 0.5$ and $\rho = 1.0$).

The methods *age1* and *age2* achieved the worst results, with *age2* slightly better than *age1* - a similar result to the other algorithms studied.

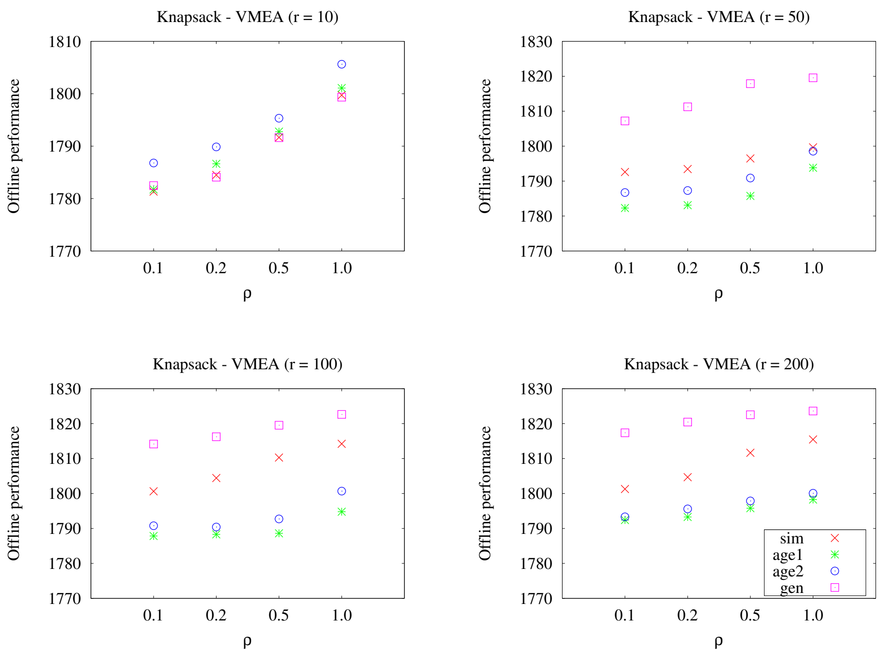


Figure 9.27: Global results obtained in the dynamic **Knapsack** problem using **VMEA** with different replacing strategies

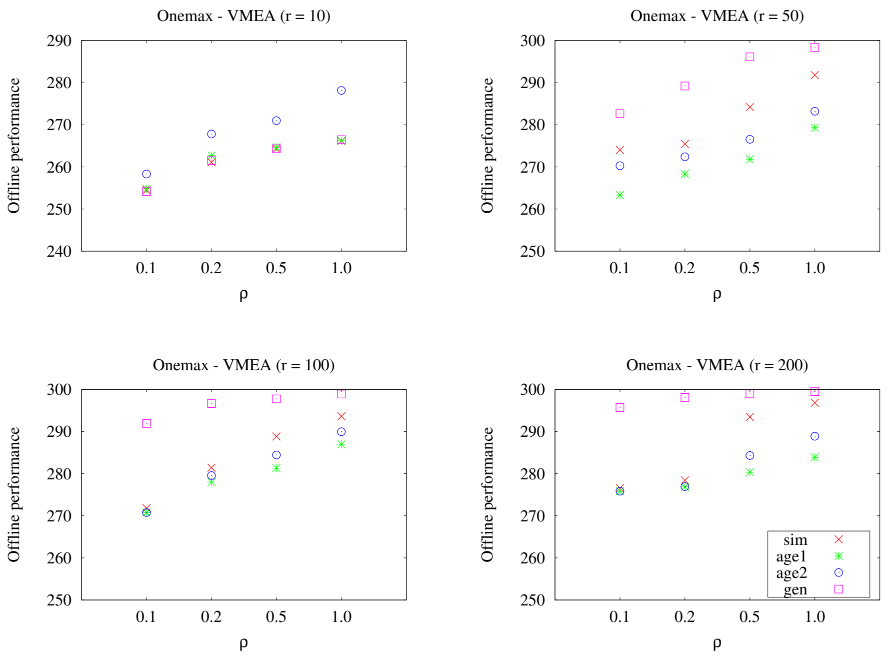


Figure 9.28: Global results obtained in the dynamic **Onemax** problem using **VMEA** with different replacing strategies

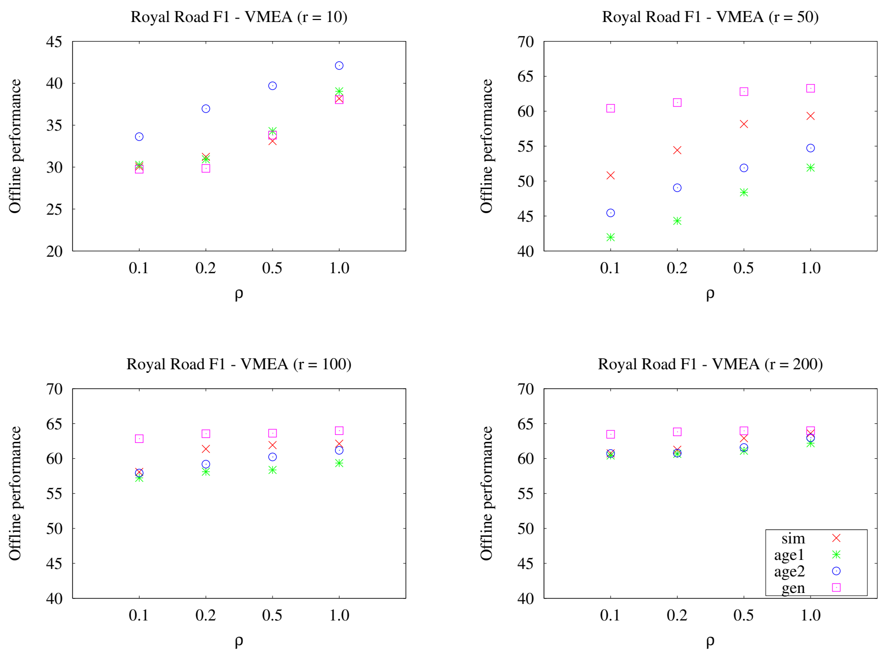


Figure 9.29: Global results obtained in the dynamic **Royal Road F1** problem using **VMEA** with different replacing strategies

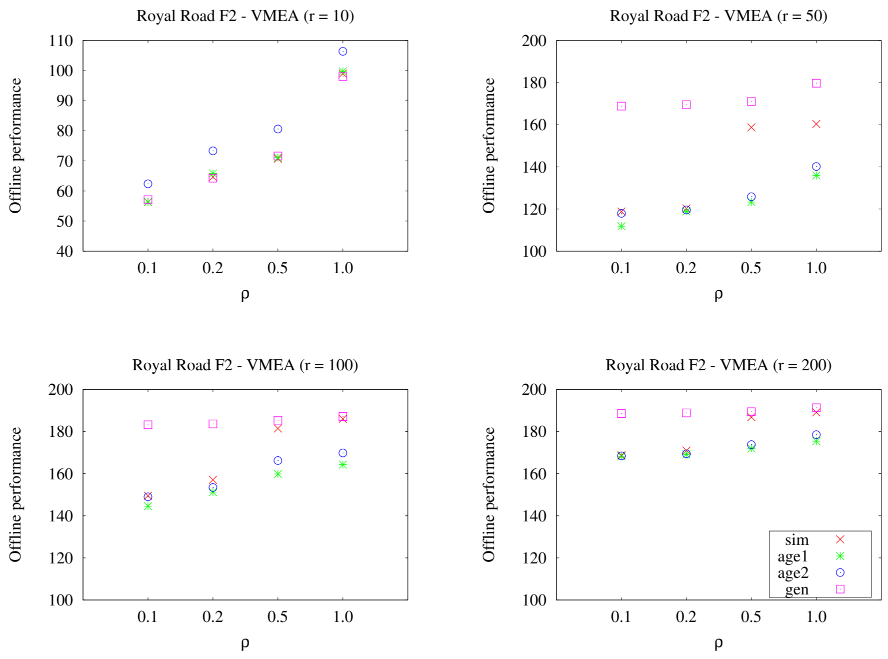


Figure 9.30: Global results obtained in the dynamic **Royal Road F2** problem using **VMEA** with different replacing strategies

		Knapsack				Onemax				Royal Road F1				Royal Road F2			
		ρ															
T-test results	r	0.1	0.2	0.5	1.0	0.1	0.2	0.5	1.0	0.1	0.2	0.5	1.0	0.1	0.2	0.5	1.0
gen – age1		+	+	−	−	−	−	−	−	−	−	−	−	+	−	+	+
gen – age2		−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−
gen – sim	10	+	−	−	−	+	+	+	+	−	−	+	−	+	−	+	+
age1 – age2		−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−
age1 – sim		+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
age2 – sim		++	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++
gen – age1		++	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++
gen – age2		++	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++
gen – sim	50	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++
age1 – age2		−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−
age1 – sim		−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−
age2 – sim		−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−	−	−−	−−
gen – age1		++	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++
gen – age2		++	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++
gen – sim	100	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++
age1 – age2		−−	−−	−−	−−	−	−−	−−	−−	−	−−	−−	−−	−−	−−	−−	−−
age1 – sim		−−	−−	−−	−−	−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−
age2 – sim		−−	−−	−−	−−	−	−−	−−	−−	−	−−	−−	−−	−	−−	−−	−−
gen – age1		++	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++
gen – age2		++	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++
gen – sim	200	++	++	++	++	++	++	++	++	++	++	++	+	++	++	++	++
age1 – age2		−−	−−	−−	−−	−	−	−−	−−	−	−	−	−	−	−	−−	−−
age1 – sim		−−	−−	−−	−−	−	−−	−−	−−	−	−	−−	−−	−	−−	−−	−−
age2 – sim		−−	−−	−−	−−	−	−−	−−	−−	−	−	−	−	−	−	−−	−−

Table 9.7: Statistical results of comparing **VMEA** using different replacing strategies

9.2.5 Discussion

When the memory has limited capacity and it is full, the replacing scheme selected to update the memory is crucial to decide which individual must be replaced. The results obtained in our study show that different types of techniques can be used and the performance of the memory-based EAs depends on the chosen method.
Branke proposed different replacing schemes and his experimentations showed that, for the problems studied, the *similar* technique was the best method. Since then, this method has been widely used in the community investigating approaches using memory.
In our work we introduced three different replacing schemes and compared the performance of different types of memory-based EAs using each one of the proposed methods. The Branke's *similar* scheme was also included in our comparisons.
The results were coherent for the four algorithms. The proposed *generational* replacing scheme always obtained the best results, except when $r = 10$. This happened because the change period was small and when a change occurred, most of the times, no individual of that period had been stored. Consequently, in such case, when memory was full, the most similar individual was replaced, conferring equivalent performances to the *similar* and the *generational* techniques. In environments that changed rapidly, the best technique was *age2*; an indication that this method - a method that computed the age of an individual based on a linear combination of its actual age and its fitness - allowed the algorithm to decide correctly which individual should be replaced. Since the individual replaced in this case was the *youngest* among the memorized solutions,

it means that its fitness was poor or its age was low.
For larger change periods, the best results were attained by the *generational* method that managed the memory capacity with more efficiency. This method kept track of the last stored individual for the current cycle and if a new individual of the same cycle had to be stored, the previous one was replaced, if worse. In this way, the memorized solutions corresponded to a wide number of different environments, minimizing redundancy efficiently. The *age1* and *age2* methods performed worse. In fact, *age1* was in general the worst method for all the situations. These results corroborated Branke's conclusions, which referred the difficulty of finding a trade-off between the fitness and the age contributions in age-based replacing schemes [14].

Chapter 10

Diversity: experimental results

This chapter shows the results obtained using the proposed genetic operators in four different types of memory-based EAs, which were tested with different benchmark problems. The results obtained using the standard uniform crossover operator with the same algorithms area also presented. The proposed genetic operators were incorporated in the EAs as substitutes for crossover. Our main goal is to see if the proposed methods are useful to preserve the population's diversity and if the performance of the memory-based EAs is related with the population's diversity. For each algorithm we present the population's diversity obtained by each operator and the consequent global performance. The statistical validation of all the comparisons is also reported.

10.1 Analysis of the results for MEGA

This section presents the results obtained by **MEGA**. Table 10.1 shows the average of the population's diversity using uniform crossover (Cx) and the two proposed operators, conjugation (Cj) and transformation (Tf). The off-line performances obtained by the different genetic operators are shown in Figures 10.1 through 10.4. The statistical validation of those comparisons are summarized on Table 10.2.
The highest diversity of the population was promoted using transformation. The lowest diversity of the population was obtained using conjugation. The diversity obtained by crossover was between those two boundaries. As the change period increased, the diversity obtained using conjugation and crossover decreased. This decrease was not so abrupt when using transformation. For the Royal Road functions F1 and F2, the diversity obtained by crossover and conjugation, for larger change periods ($r = 100$, $r = 200$), was similar.
For the Knapsack and the Onemax problems, the best performances of **MEGA** using the different genetic operators, were achieved using conjugation (lowest diversity). On the other hand, **MEGA** with transformation attained the worst results, corresponding to the highest diversity of the population. The only ex-

ception occurred in the Onemax problem, for $\rho = 1.0$, where the performances of transformation and crossover were statistically equivalent.
These observations show that, for those two problems, the retrieval of the memorized information when a change occurred, associated with some exploration (lower diversity) of the search space ensured the best performance of **MEGA**. For the Royal Road functions, when $r = 10$ and $r = 50$ - for $\rho = 0.1$ and $\rho = 0.2$ - crossover and conjugation achieved the best results (without statistical difference). For $\rho = 1.0$, transformation allowed **MEGA** to obtain the best scores. As the change period increased, conjugation and crossover continued to obtain similar performances, including when $\rho = 0.5$. Transformation was the most effective operator for harsher changes, indicating that the memorized solutions were not enough to help the EA when the changes occurred. The low diversity obtained by crossover and conjugation in larger change periods, indicate that the memorized solutions were prematurely converged and extra diversity was beneficial for those situations. For larger change periods, the diversity obtained by conjugation and crossover was equivalent and the performance of **MEGA**, for these cases, was not statistically different.
The results obtained showed that the performance of **MEGA** was affected by the population's diversity. In general, a lower diversity was better, particularly for gradual severities on the environmental changes. Depending on the problem, different performances were obtained for higher values of ρ.

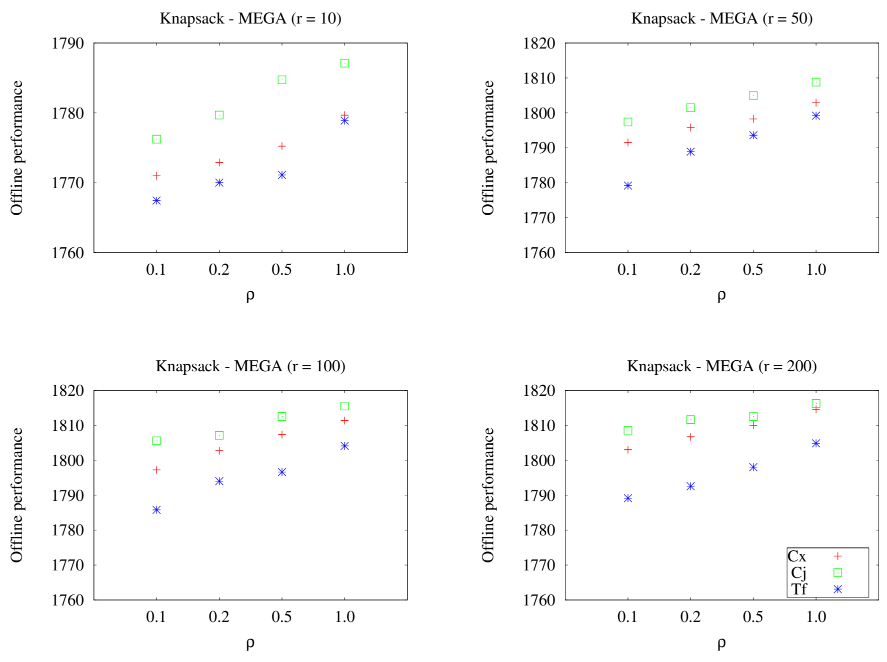


Figure 10.1: Global results obtained in the dynamic **Knapsack** problem using **MEGA** with different genetic operators

	Knapsack											
	$r = 10$			$r = 50$			$r = 100$			$r = 200$		
ρ	Cx	Cj	Tf	Cx	Cj	Tf	Cx	Cj	Tf	Cx	Cj	Tf
0.1	0.32	0.16	0.48	0.30	0.10	0.45	0.22	0.09	0.40	0.18	0.07	0.39
0.2	0.33	0.17	0.48	0.31	0.10	0.45	0.22	0.09	0.41	0.18	0.06	0.37
0.5	0.35	0.17	0.48	0.32	0.09	0.45	0.23	0.09	0.40	0.17	0.06	0.37
1.0	0.33	0.17	0.48	0.32	0.09	0.46	0.24	0.09	0.41	0.17	0.06	0.37
	Onemax											
	$r = 10$			$r = 50$			$r = 100$			$r = 200$		
ρ	Cx	Cj	Tf	Cx	Cj	Tf	Cx	Cj	Tf	Cx	Cj	Tf
0.1	0.28	0.11	0.44	0.25	0.10	0.42	0.18	0.09	0.40	0.15	0.06	0.38
0.2	0.28	0.12	0.42	0.25	0.09	0.42	0.18	0.08	0.39	0.14	0.06	0.38
0.5	0.32	0.15	0.43	0.24	0.09	0.43	0.19	0.08	0.39	0.15	0.06	0.39
1.0	0.31	0.15	0.45	0.24	0.10	0.42	0.19	0.08	0.38	0.15	0.06	0.31
	Royal Road F1											
	$r = 10$			$r = 50$			$r = 100$			$r = 200$		
ρ	Cx	Cj	Tf	Cx	Cj	Tf	Cx	Cj	Tf	Cx	Cj	Tf
0.1	0.22	0.13	0.46	0.07	0.06	0.44	0.06	0.05	0.41	0.05	0.05	0.39
0.2	0.26	0.15	0.47	0.08	0.07	0.43	0.07	0.06	0.42	0.05	0.04	0.38
0.5	0.26	0.16	0.47	0.09	0.08	0.38	0.08	0.06	0.38	0.06	0.04	0.37
1.0	0.15	0.16	0.45	0.10	0.09	0.32	0.07	0.06	0.30	0.06	0.05	0.29
	Royal Road F2											
	$r = 10$			$r = 50$			$r = 100$			$r = 200$		
ρ	Cx	Cj	Tf	Cx	Cj	Tf	Cx	Cj	Tf	Cx	Cj	Tf
0.1	0.22	0.13	0.46	0.07	0.06	0.44	0.05	0.04	0.41	0.05	0.04	0.38
0.2	0.26	0.14	0.46	0.09	0.07	0.43	0.06	0.05	0.42	0.05	0.05	0.38
0.5	0.25	0.15	0.47	0.10	0.07	0.39	0.07	0.05	0.3	0.06	0.04	0.38
1.0	0.15	0.14	0.45	0.13	0.06	0.32	0.08	0.05	0.31	0.06	0.05	0.30

Table 10.1: Population's Diversity for **MEGA**

		Knapsack				Onemax				Royal Road F1				Royal Road F2			
		ρ															
T-test results	r	0.1	0.2	0.5	1.0	0.1	0.2	0.5	1.0	0.1	0.2	0.5	1.0	0.1	0.2	0.5	1.0
Cj – Cx	10	++	++	++	++	++	++	++	++	+	+	+	+	+	+	+	+
Cj – Tf		++	++	++	++	++	++	++	++	++	++	−−	−−	++	++	−−	−−
Tf – Cx		−−	−−	−−	−	−−	−−	−−	+	−−	−−	++	++	−−	−−	++	++
Cj – Cx	50	++	++	++	++	++	++	++	++	+	+	+	+	+	+	+	+
Cj – Tf		++	++	++	++	++	++	++	++	++	++	++	−−	++	++	++	−−
Tf – Cx		−−	−−	−−	−−	−−	−−	−−	+	−−	−−	−−	++	−−	−−	−−	++
Cj – Cx	100	++	++	++	++	+	+	+	+	+	+	+	+	+	+	+	+
Cj – Tf		++	++	++	++	++	++	++	++	++	++	++	−	++	++	++	−
Tf – Cx		−−	−−	−−	−−	−−	−−	−−	−	−−	−−	−−	+	−−	−−	−−	+
Cj – Cx	200	++	++	++	++	+	+	+	+	+	+	+	+	+	+	+	+
Cj – Tf		++	++	++	++	++	++	++	+	++	++	++	−	++	++	++	−
Tf – Cx		−−	−−	−−	−−	−−	−−	−−	−	−−	−−	−−	+	−−	−−	−−	+

Table 10.2: Statistical results of comparing **MEGA** using different genetic operators

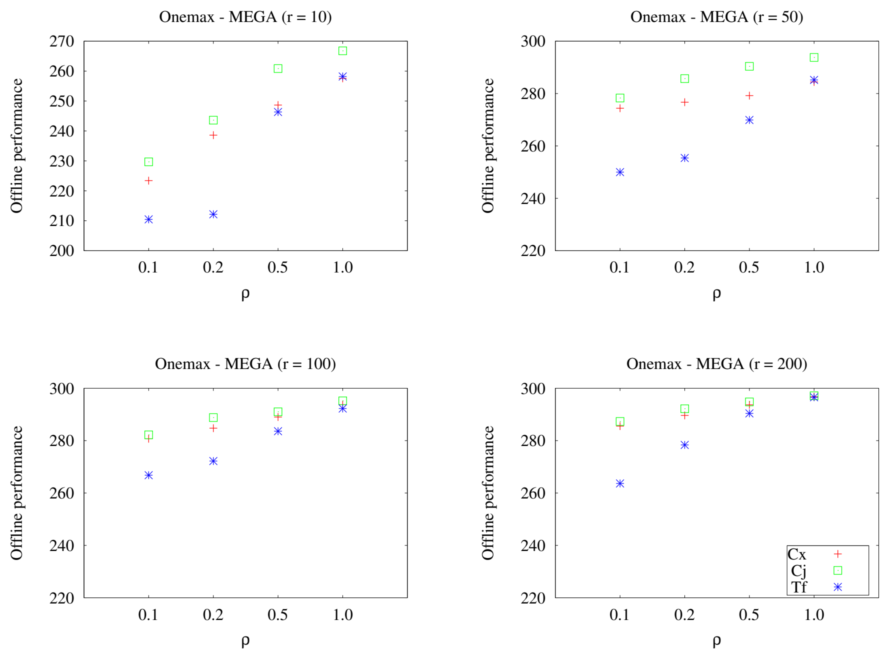


Figure 10.2: Global results obtained in the dynamic **Onemax** problem using **MEGA** with different genetic operators

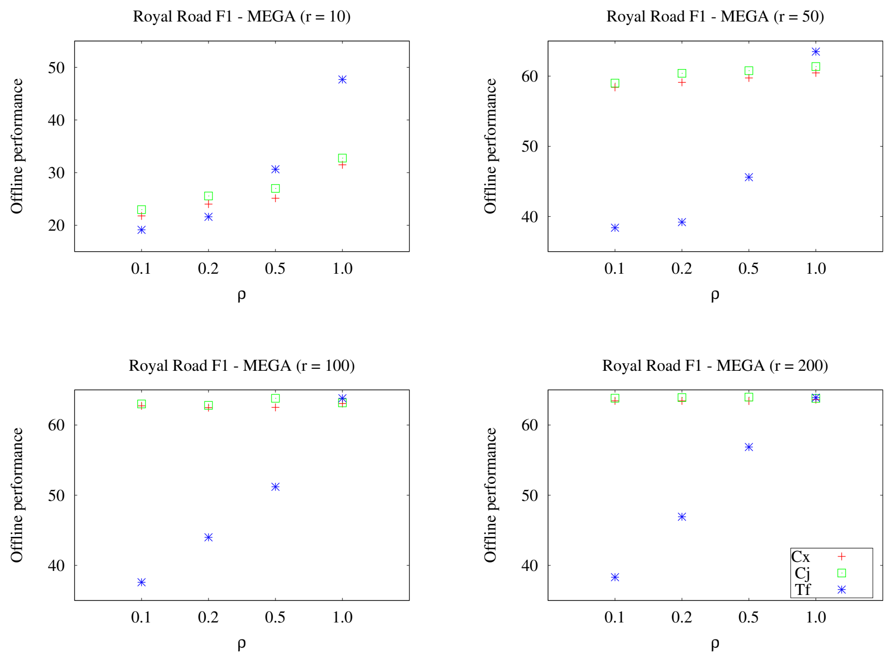


Figure 10.3: Global results obtained in the dynamic **Royal Road F1** problem using **MEGA** with different genetic operators

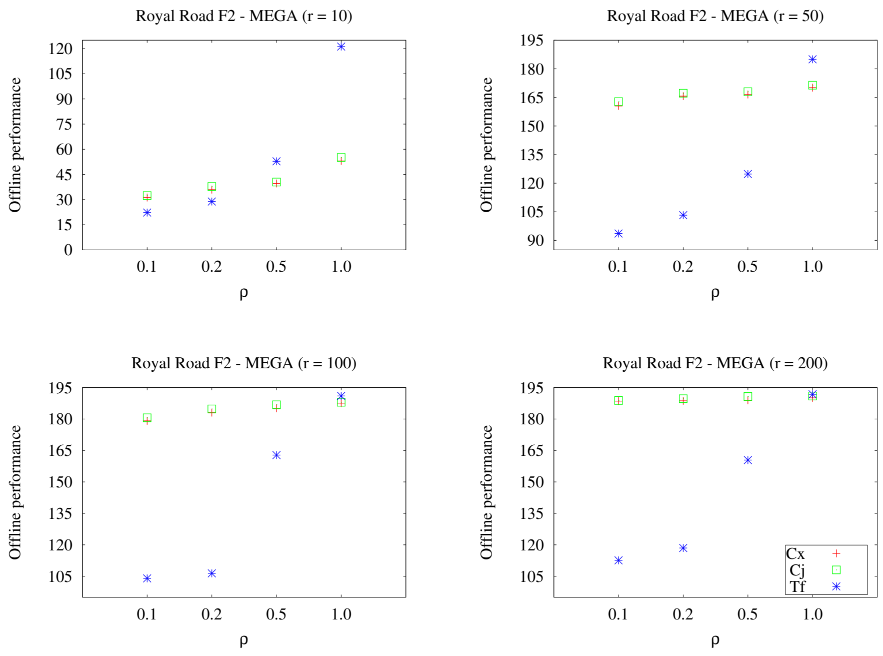


Figure 10.4: Global results obtained in the dynamic **Royal Road F2** problem using **MEGA** with different genetic operators

10.2 Analysis of the results for MIGA

The results obtained by **MIGA** are summarized in this section. The average of the population's diversity obtained by the different methods can be consulted in Table 10.3. The performance of **MIGA** using different genetic operators is plotted in Figures 10.5 through 10.8 and the corresponding statistical comparisons are in Table 10.3.
The analysis of the results attained by **MIGA** is comparable to the previously explained for **MEGA**. The operator that preserved more diversity in the population was transformation. Conjugation was the method that less contributed for the diversity. For larger change periods - $r = 100$ and $r = 200$ - the diversity maintained by crossover and conjugation was equivalent. Several results can be observed analyzing the performance of **MIGA**, as follows. For the Knapsack and Onemax problems, in general, the best results results were obtained using conjugation, corresponding to the lowest diversity in the population. The only exception occurred for $r = 10$ and $\rho = 1.0$, where transformation outperformed conjugation. As the change period increased, the performances of **MIGA** using conjugation or crossover became statistical equivalent, corresponding to similar values of diversity. The worst results were mainly achieved using transformation, corresponding to the highest diversity of the population. This operator performed better when the severity of the change was high ($\rho = 1.0$).
For the Royal Road functions F1 and F2, the results for **MIGA** were also similar to **MEGA**'s results. Conjugation was always better than crossover but, for larger change periods, those differences were not statistically evident. As before, the statistical equivalence for crossover/conjugation comparisons, corresponded to the situations where the diversity maintained by these two operators was close. Transformation was better than conjugation for severer changes in smaller change periods. As the change period increased, the performances obtained using transformation or conjugation were statistical equivalent. Once more, transformation performed better when $\rho = 1.0$. These results indicate that, when the changes were harsher, a higher level of diversity was important to improve the EA's performance. This can be justified by the fact that the memorized information was not enough to guide the immigrants towards the new optimum.

	Knapsack											
	$r = 10$			$r = 50$			$r = 100$			$r = 200$		
ρ	Cx	Cj	Tf	Cx	Cj	Tf	Cx	Cj	Tf	Cx	Cj	Tf
0.1	0.22	0.10	0.41	0.15	0.09	0.37	0.10	0.09	0.38	0.09	0.08	0.34
0.2	0.21	0.11	0.41	0.16	0.09	0.37	0.10	0.08	0.38	0.09	0.08	0.34
0.5	0.19	0.10	0.41	0.15	0.10	0.37	0.09	0.09	0.38	0.09	0.07	0.33
1.0	0.19	0.10	0.41	0.16	0.10	0.35	0.09	0.09	0.38	0.09	0.08	0.31
	Onemax											
	$r = 10$			$r = 50$			$r = 100$			$r = 200$		
ρ	Cx	Cj	Tf	Cx	Cj	Tf	Cx	Cj	Tf	Cx	Cj	Tf
0.1	0.14	0.10	0.30	0.10	0.09	0.27	0.07	0.05	0.25	0.06	0.06	0.25
0.2	0.15	0.10	0.31	0.10	0.08	0.27	0.07	0.06	0.24	0.06	0.06	0.24
0.5	0.16	0.10	0.30	0.11	0.09	0.28	0.07	0.06	0.24	0.06	0.05	0.20
1.0	0.17	0.19	0.31	0.10	0.09	0.15	0.07	0.07	0.13	0.07	0.06	0.11
	Royal Road F1											
	$r = 10$			$r = 50$			$r = 100$			$r = 200$		
ρ	Cx	Cj	Tf	Cx	Cj	Tf	Cx	Cj	Tf	Cx	Cj	Tf
0.1	0.13	0.11	0.33	0.11	0.09	0.26	0.04	0.05	0.25	0.04	0.05	0.22
0.2	0.13	0.11	0.33	0.11	0.09	0.26	0.05	0.05	0.24	0.04	0.05	0.22
0.5	0.13	0.11	0.28	0.11	0.09	0.26	0.06	0.05	0.25	0.04	0.04	0.22
1.0	0.11	0.10	0.25	0.11	0.09	0.20	0.06	0.05	0.16	0.05	0.04	0.14
	Royal Road F2											
	$r = 10$			$r = 50$			$r = 100$			$r = 200$		
ρ	Cx	Cj	Tf	Cx	Cj	Tf	Cx	Cj	Tf	Cx	Cj	Tf
0.1	0.13	0.11	0.32	0.06	0.07	0.26	0.04	0.05	0.26	0.04	0.05	0.25
0.2	0.13	0.11	0.33	0.07	0.06	0.26	0.05	0.05	0.25	0.04	0.05	0.25
0.5	0.13	0.10	0.30	0.07	0.06	0.21	0.06	0.05	0.20	0.04	0.05	0.19
1.0	0.12	0.11	0.28	0.07	0.07	0.15	0.05	0.05	0.11	0.05	0.05	0.10

Table 10.3: Population's Diversity for **MIGA**

		Knapsack				Onemax				Royal Road F1				Royal Road F2			
		ρ															
T-test results	r	0.1	0.2	0.5	1.0	0.1	0.2	0.5	1.0	0.1	0.2	0.5	1.0	0.1	0.2	0.5	1.0
Cj – Cx	10	++	++	++	++	++	++	++	++	++	++	++	++	+	+	+	+
Cj – Tf		++	++	++	−−	++	++	++	−−	++	++	−−	−−	+	+	−−	−−
Tf – Cx		−−	−−	−−	++	−−	−−	−−	++	−−	−−	++	++	−	−	++	++
Cj – Cx	50	++	++	++	+	+	+	+	+	++	++	+	+	+	+	+	++
Cj – Tf		++	++	++	++	++	++	++	+	++	++	−−	−−	++	++	−	−−
Tf – Cx		−−	−−	−−	−−	−−	−−	−−	−	−−	−−	++	++	−−	−−	+	++
Cj – Cx	100	+	+	+	+	+	+	+	+	+	++	++	++	+	+	+	+
Cj – Tf		++	++	++	++	++	++	++	+	++	++	++	−−	++	++	++	+
Tf – Cx		−−	−−	−−	−−	−−	−−	−−	−	−−	−−	−	++	−−	−−	−−	+
Cj – Cx	200	+	+	+	+	+	+	+	+	+	+	+	++	+	+	++	++
Cj – Tf		++	++	++	++	++	++	++	+	++	++	+	−	++	++	++	−
Tf – Cx		−−	−−	−−	−−	−−	−−	−−	−	−−	−−	−	++	−−	−−	−−	+

Table 10.4: Statistical results of comparing **MIGA** using different genetic operators

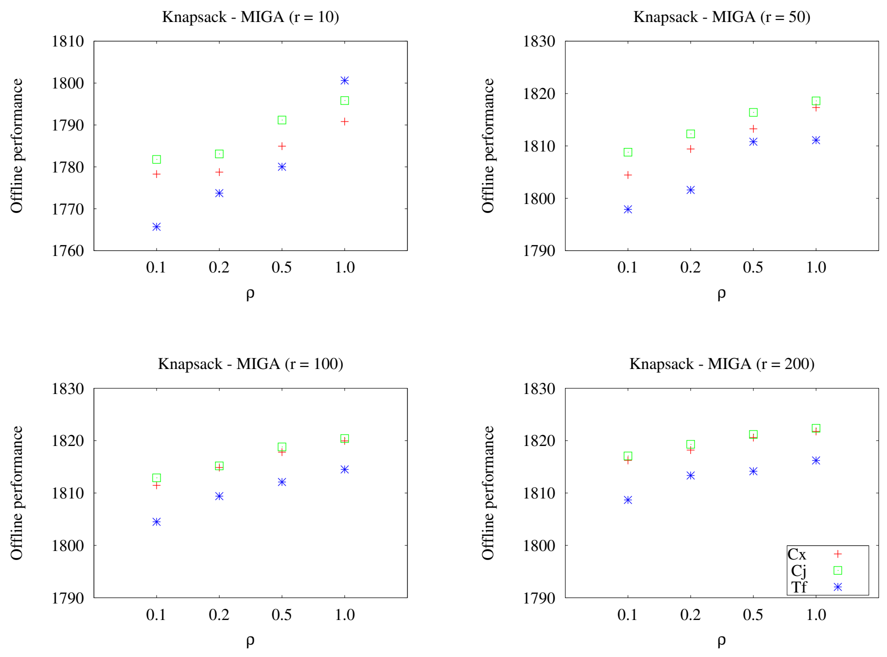


Figure 10.5: Global results obtained in the dynamic **Knapsack** problem using **MIGA** with different genetic operators

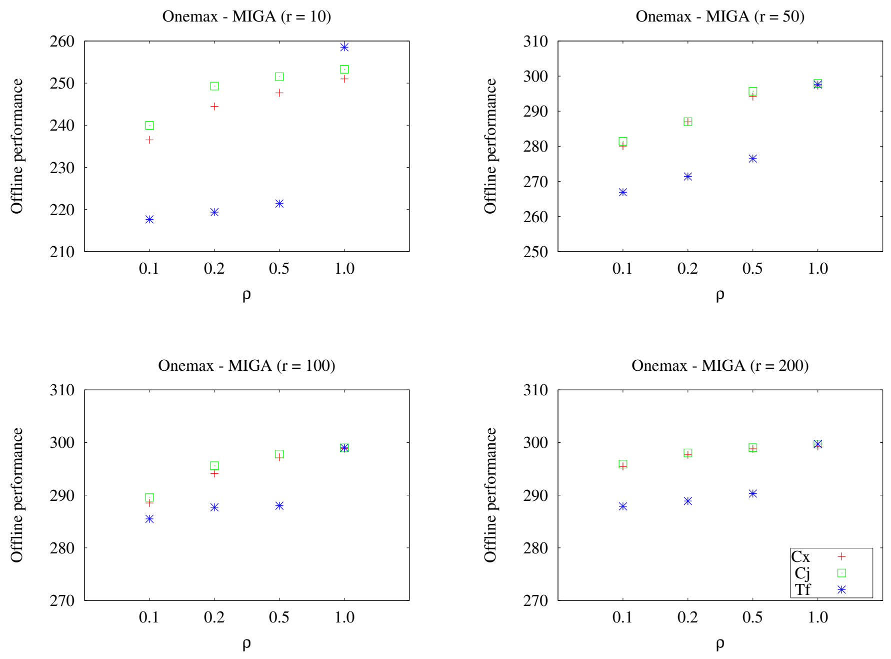


Figure 10.6: Global results obtained in the dynamic **Onemax** problem using **MIGA** with different genetic operators

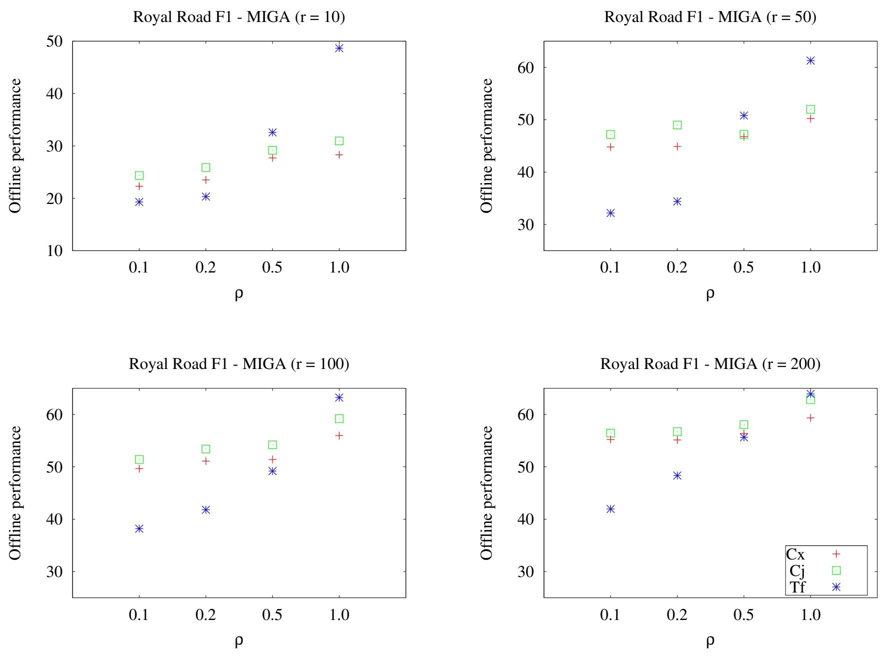


Figure 10.7: Global results obtained in the dynamic **Royal Road F1** problem using **MIGA** with different genetic operators

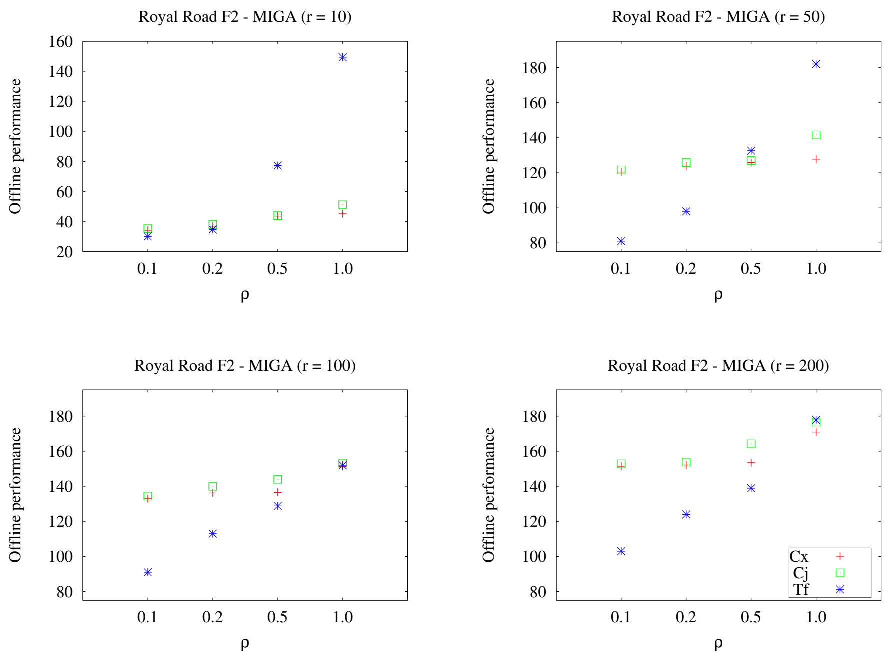


Figure 10.8: Global results obtained in the dynamic **Royal Road F2** problem using **MIGA** with different genetic operators

10.3 Analysis of the results for AMGA

Table 10.5 shows the average of the population's diversity preserved in the **AMGA** using the three genetic operators. Figures 10.9 through 10.12 show the overall performance obtained by **AMGA** and the statistical results of the comparisons using different genetic operators are reported on Table 10.6.

Concerning the diversity level, the results obtained by **AMGA** were consistent with the previous results for **MEGA** and **MIGA**: for all problems, the operator that kept the diversity of the population at a higher level was transformation; crossover came in second place and the lowest value of diversity was obtained using conjugation. Another similar conclusion was that, because of the convergence of the population, as the change period (r) increased, the diversity decreased. The diversity had similar values, independently of the severity of the change (ρ).

The results show that, a higher diversity of the population was responsible for a better performance of the algorithm only in a minority of the cases. Transformation was helpful for the Knapsack and the Onemax problems in rapid changing environments ($r = 10$) with severer changes ($\rho = 0.5$, $\rho = 1.0$). For the Royal Road functions, transformation slightly improved (not statistically significant) **AMGA**'s performance for $r = 100$ and $r = 200$ and $\rho = 1.0$. In the remaining situations, the great majority, conjugation allowed **AMGA** to obtain the best scores. Crossover and conjugation had equivalent performances for larger change periods, especially for the Royal Road functions, where the diversity levels maintained by those two methods were also very close.

These results show that, for the Onemax and Knapsack problems, when the changes occurred faster and the memorized solutions corresponded to low fitted individuals, when a severe change happened, extra diversity helped the EA in the new environment. The same didn't occur for the Royal Road functions because of the inherent characteristics of these problems. In those cases, transformation was a disruptive method with no positive contribution for the performance of the EA. Another observation was that, when the diversity maintained by crossover and conjugation became similar, the performance of this EA using those two methods was equivalent.

	Knapsack											
	$r = 10$			$r = 50$			$r = 100$			$r = 200$		
ρ	Cx	Cj	Tf	Cx	Cj	Tf	Cx	Cj	Tf	Cx	Cj	Tf
0.1	0.34	0.19	0.38	0.23	0.10	0.33	0.20	0.09	0.30	0.19	0.06	0.28
0.2	0.36	0.18	0.38	0.24	0.11	0.32	0.20	0.08	0.30	0.19	0.06	0.28
0.5	0.36	0.16	0.36	0.23	0.11	0.32	0.20	0.09	0.31	0.18	0.08	0.28
1.0	0.34	0.17	0.35	0.22	0.11	0.31	0.20	0.09	0.30	0.18	0.07	0.27
	Onemax											
	$r = 10$			$r = 50$			$r = 100$			$r = 200$		
ρ	Cx	Cj	Tf	Cx	Cj	Tf	Cx	Cj	Tf	Cx	Cj	Tf
0.1	0.23	0.15	0.36	0.14	0.10	0.33	0.11	0.09	0.30	0.11	0.07	0.31
0.2	0.27	0.14	0.36	0.15	0.09	0.34	0.11	0.09	0.30	0.11	0.06	0.29
0.5	0.25	0.15	0.35	0.15	0.10	0.31	0.11	0.08	0.31	0.11	0.08	0.30
1.0	0.27	0.15	0.37	0.15	0.10	0.30	0.11	0.09	0.29	0.11	0.08	0.24
	Royal Road F1											
	$r = 10$			$r = 50$			$r = 100$			$r = 200$		
ρ	Cx	Cj	Tf	Cx	Cj	Tf	Cx	Cj	Tf	Cx	Cj	Tf
0.1	0.23	0.15	0.38	0.08	0.07	0.37	0.06	0.06	0.35	0.05	0.05	0.33
0.2	0.23	0.17	0.38	0.09	0.08	0.37	0.07	0.06	0.35	0.05	0.05	0.33
0.5	0.25	0.19	0.39	0.14	0.11	0.35	0.09	0.08	0.34	0.06	0.05	0.30
1.0	0.18	0.15	0.36	0.14	0.12	0.31	0.11	0.10	0.30	0.09	0.08	0.27
	Royal Road F2											
	$r = 10$			$r = 50$			$r = 100$			$r = 200$		
ρ	Cx	Cj	Tf	Cx	Cj	Tf	Cx	Cj	Tf	Cx	Cj	Tf
0.1	0.24	0.15	0.37	0.08	0.07	0.35	0.06	0.06	0.32	0.05	0.05	0.30
0.2	0.25	0.19	0.38	0.08	0.08	0.35	0.06	0.06	0.31	0.05	0.05	0.30
0.5	0.29	0.20	0.37	0.12	0.09	0.32	0.08	0.07	0.31	0.05	0.05	0.29
1.0	0.19	0.14	0.36	0.13	0.11	0.31	0.12	0.10	0.28	0.08	0.07	0.24

Table 10.5: Population's Diversity for **AMGA**

		Knapsack				Onemax				Royal Road F1				Royal Road F2			
		ρ															
T-test results	r	0.1	0.2	0.5	1.0	0.1	0.2	0.5	1.0	0.1	0.2	0.5	1.0	0.1	0.2	0.5	1.0
Cj – Cx		++	++	+	+	+	+	++	++	++	++	++	++	+	+	+	++
Cj – Tf	10	++	−	−−	−−	++	++	++	−−	++	++	++	++	++	++	++	++
Tf – Cx		−−	++	++	++	−−	−−	−−	++	−−	−−	−−	−−	−−	−−	−−	−−
Cj – Cx		++	++	++	++	+	+	+	++	+	+	++	++	+	+	+	+
Cj – Tf	50	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++	++
Tf – Cx		−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−
Cj – Cx		++	++	++	++	+	+	+	+	+	+	+	+	+	+	+	+
Cj – Tf	100	++	++	++	++	++	++	++	++	++	++	++	+	++	++	++	−
Tf – Cx		−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	+	−−	−−	−−	+
Cj – Cx		++	++	++	++	++	++	++	++	+	+	+	+	+	+	+	+
Cj – Tf	200	++	++	++	++	++	++	++	++	++	++	++	+	++	++	++	−
Tf – Cx		−−	−−	−−	−−	−−	−−	−−	−−	−−	−−	−	−−	−−	−−	−−	+

Table 10.6: Statistical results of comparing **AMGA** using different genetic operators

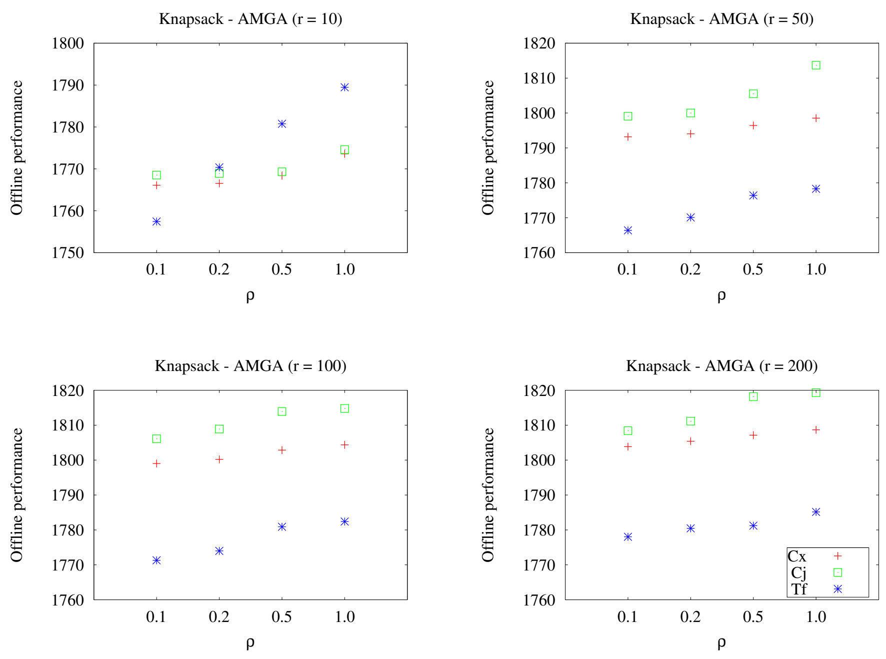


Figure 10.9: Global results obtained in the dynamic **Knapsack** problem using **AMGA** with different genetic operators

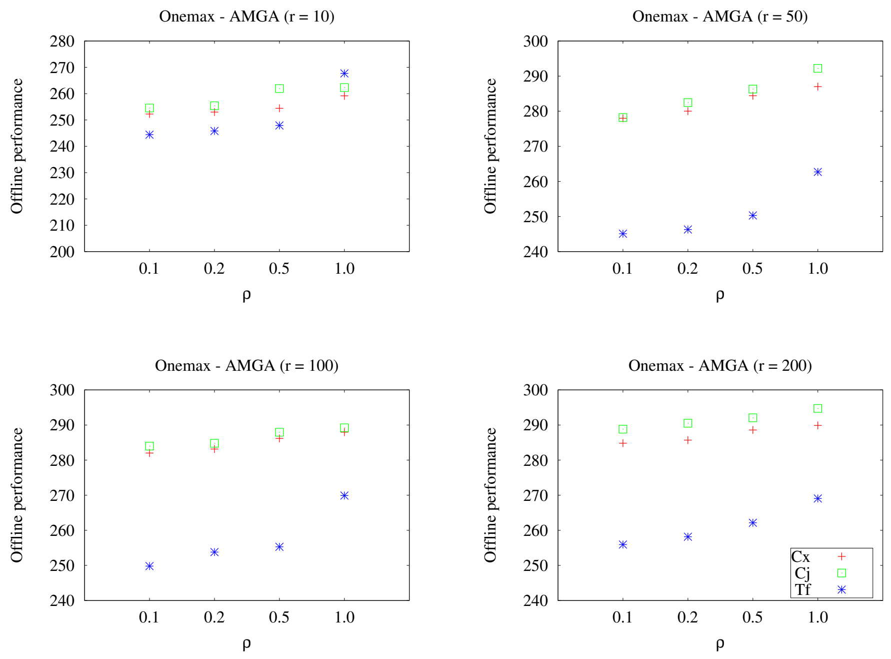


Figure 10.10: Global results obtained in the dynamic **Onemax** problem using **AMGA** with different genetic operators

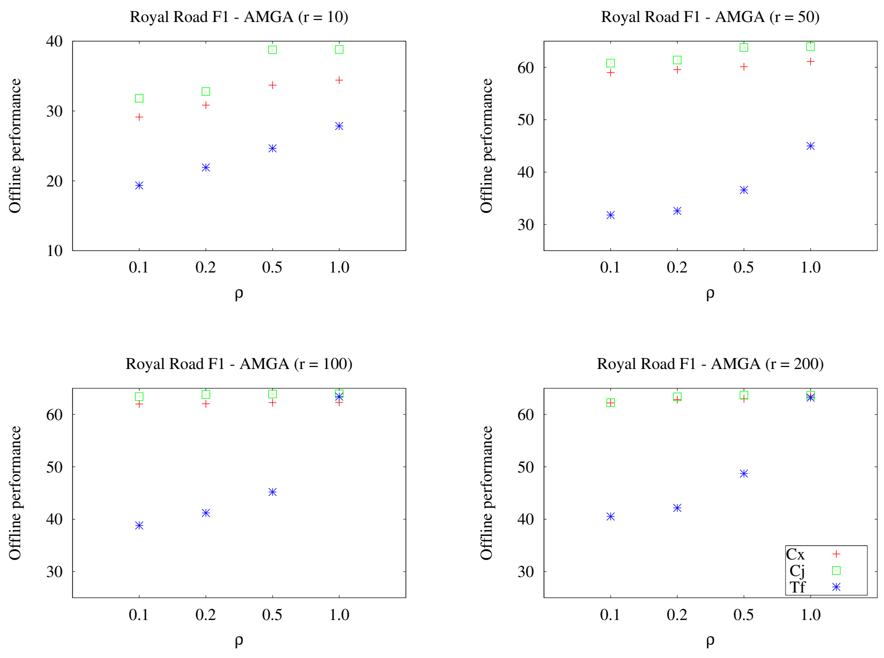


Figure 10.11: Global results obtained in the dynamic **Royal Road F1** problem using **AMGA** with different genetic operators

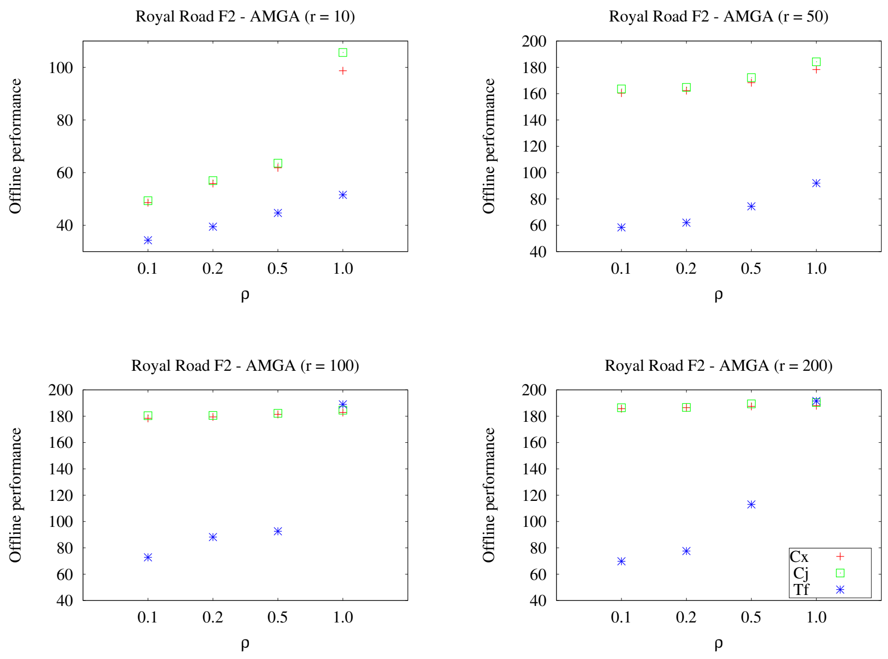


Figure 10.12: Global results obtained in the dynamic **Royal Road F2** problem using **AMGA** with different genetic operators

10.4 Analysis of the results for VMEA

Table 10.7 shows the average of the population's diversity obtained by **VMEA** with different genetic operators. The global performances of the algorithm for the studied problems are shown in Figures 10.13 through 10.16. The statistical results of comparing among the different operators are summarized in Table 10.8.

The results obtained by **VMEA** were similar to **MEGA**'s results, previously analyzed. This was because both algorithms used the same mechanisms for retrieving and storing information.

The highest diversity of the population was maintained using transformation and the lowest diversity was obtained using conjugation. The best performance of **VMEA** was generally obtained using conjugation, while the worst scores were mainly achieved using transformation. As for the previously described algorithms, in the situations were the diversity promoted by conjugation and crossover had similar values, the corresponding performance of **VMEA** was also equivalent. Transformation was beneficial in very few cases cases: for the Royal Road functions when $r = 10$ and the severity of change was higher ($\rho = 0.5$ and $\rho = 1.0$). Once more, in general, a higher diversity of the population lead to worst performances of the EA. The memorized information was responsible for improving the performance of the algorithms rather than the promotion of diversity by means of the genetic operators.

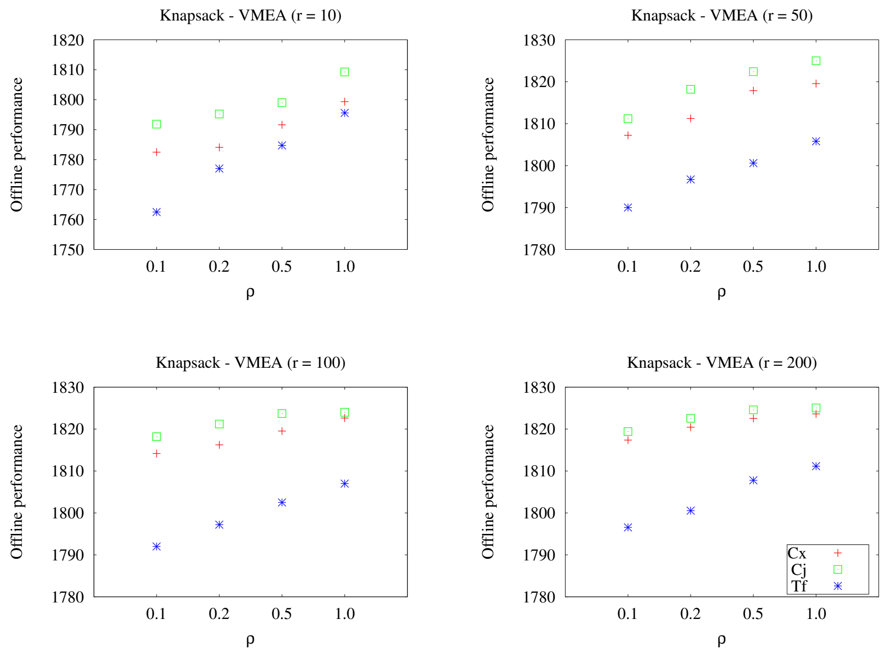


Figure 10.13: Global results obtained in the dynamic **Knapsack** problem using **VMEA** with different genetic operators

	Knapsack											
	$r = 10$			$r = 50$			$r = 100$			$r = 200$		
ρ	Cx	Cj	Tf	Cx	Cj	Tf	Cx	Cj	Tf	Cx	Cj	Tf
0.1	0.22	0.17	0.39	0.21	0.09	0.39	0.19	0.08	0.32	0.15	0.06	0.30
0.2	0.27	0.17	0.41	0.20	0.10	0.39	0.20	0.07	0.32	0.15	0.06	0.30
0.5	0.30	0.19	0.41	0.22	0.10	0.38	0.19	0.08	0.35	0.17	0.06	0.31
1.0	0.28	0.19	0.42	0.22	0.11	0.38	0.21	0.08	0.35	0.19	0.05	0.32
	Onemax											
	$r = 10$			$r = 50$			$r = 100$			$r = 200$		
ρ	Cx	Cj	Tf	Cx	Cj	Tf	Cx	Cj	Tf	Cx	Cj	Tf
0.1	0.20	0.14	0.46	0.11	0.09	0.41	0.10	0.08	0.36	0.10	0.08	0.30
0.2	0.23	0.14	0.45	0.12	0.09	0.41	0.10	0.08	0.35	0.10	0.08	0.31
0.5	0.25	0.18	0.45	0.14	0.11	0.41	0.11	0.09	0.37	0.10	0.09	0.30
1.0	0.26	0.18	0.46	0.17	0.10	0.38	0.13	0.10	0.32	0.11	0.09	0.27
	Royal Road F1											
	$r = 10$			$r = 50$			$r = 100$			$r = 200$		
ρ	Cx	Cj	Tf	Cx	Cj	Tf	Cx	Cj	Tf	Cx	Cj	Tf
0.1	0.22	0.13	0.42	0.11	0.09	0.40	0.09	0.07	0.38	0.06	0.06	0.32
0.2	0.25	0.13	0.46	0.10	0.09	0.40	0.10	0.07	0.37	0.07	0.06	0.32
0.5	0.27	0.14	0.46	0.15	0.12	0.37	0.10	0.09	0.35	0.08	0.07	0.34
1.0	0.14	0.10	0.46	0.22	0.11	0.35	0.12	0.09	0.34	0.10	0.09	0.31
	Royal Road F2											
	$r = 10$			$r = 50$			$r = 100$			$r = 200$		
ρ	Cx	Cj	Tf	Cx	Cj	Tf	Cx	Cj	Tf	Cx	Cj	Tf
0.1	0.21	0.13	0.45	0.09	0.07	0.42	0.07	0.07	0.37	0.06	0.06	0.31
0.2	0.26	0.14	0.45	0.11	0.10	0.42	0.07	0.07	0.37	0.07	0.06	0.32
0.5	0.25	0.14	0.46	0.15	0.10	0.40	0.08	0.08	0.35	0.08	0.06	0.31
1.0	0.15	0.11	0.46	0.17	0.12	0.38	0.11	0.10	0.32	0.09	0.07	0.26

Table 10.7: Population's Diversity for **VMEA**

		Knapsack				Onemax				Royal Road F1				Royal Road F2			
		ρ															
T-test results	r	0.1	0.2	0.5	1.0	0.1	0.2	0.5	1.0	0.1	0.2	0.5	1.0	0.1	0.2	0.5	1.0
Cj – Cx		++	++	++	++	++	++	++	++	++	++	+	+	+	+	+	++
Cj – Tf	10	++	++	++	++	++	++	++	++	++	++	−−	−−	++	++	−−	−−
Cx – Tf		++	++	++	++	++	++	++	++	++	++	−−	−−	++	++	−−	−−
Cj – Cx		++	++	++	++	++	++	+	+	+	+	+	+	+	+	+	+
Cj – Tf	50	++	++	++	++	++	++	++	++	++	++	++	+	++	++	++	−−
Cx – Tf		++	++	++	++	++	++	++	++	++	++	++	−	++	++	++	−
Cj – Cx		++	++	++	+	+	+	+	+	+	+	+	+	+	+	+	+
Cj – Tf	100	++	++	++	++	++	++	++	++	++	++	++	+	++	++	++	−
Cx – Tf		++	++	++	++	++	++	++	+	++	++	++	+	++	++	++	−
Cj – Cx		++	++	++	+	+	+	+	+	+	+	+	+	+	+	+	+
Cj – Tf	200	++	++	++	++	++	++	++	+	++	++	++	+	++	++	++	+
Cx – Tf		++	++	++	++	++	++	+	+	++	++	++	+	++	++	++	−

Table 10.8: Statistical results of comparing **VMEA** using different genetic operators

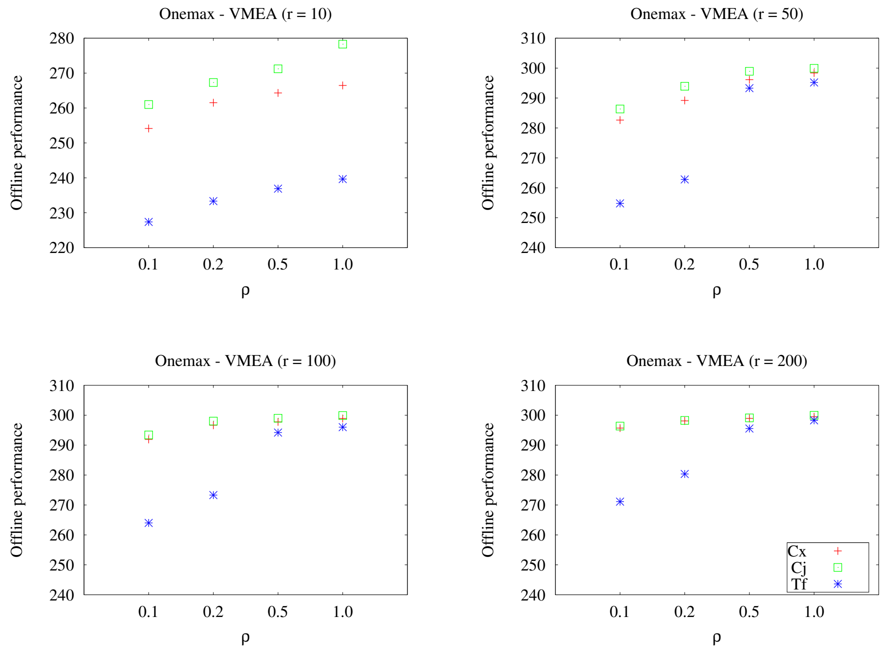


Figure 10.14: Global results obtained in the dynamic **Onemax** problem using **VMEA** with different genetic operators

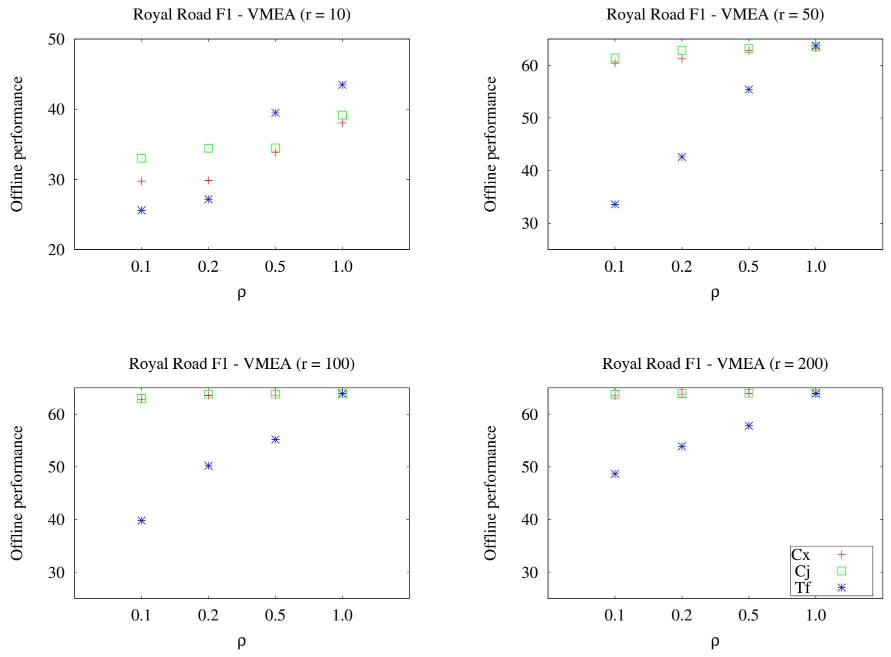


Figure 10.15: Global results obtained in the dynamic **Royal Road F1** problem using **VMEA** with different genetic operators

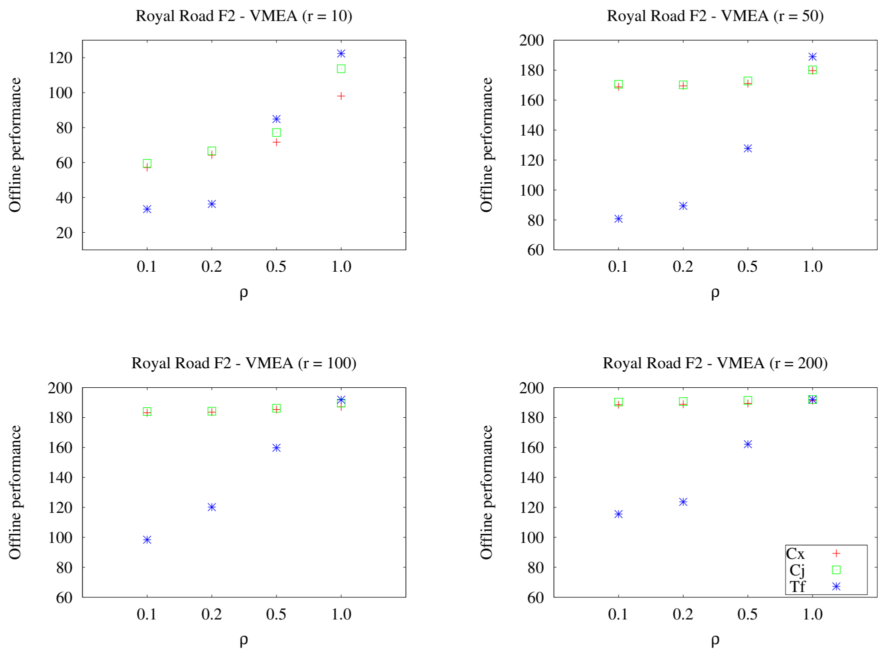


Figure 10.16: Global results obtained in the dynamic **Royal Road F2** problem using **VMEA** with different genetic operators

10.5 Discussion

The use of mechanisms that generate and maintain diversity in the population was always seen as fundamental to help EAs to achieve better performances when dealing with dynamic environments. In the last years, several studies showed that this was not always true and, in some situations, *too much* diversity could hinder the performance of the EA for dynamic environments [89], [123].
In order to have more insight about this important issue, we proposed and implemented two alternative genetic operators to use in different memory-based EAs. All the algorithms were tested with four different benchmark problems using the same parameter setting, except for the standard recombination operator. We measured the population's diversity and the performances obtained by the algorithms and obtained some important conclusions. Those conclusions can be summarized in the following points:

1. Transformation obtained the highest diversity level;
2. Conjugation obtained the lowest diversity level;
3. As the change period increased, the diversity generated by crossover and conjugation was similar;
4. In general, the best performances of the algorithms corresponded to the lowest diversity (conjugation and crossover);
5. Transformation and a higher diversity were advantageous, mainly in the Royal Road functions, when the environment changed slower with higher severity.

The stated conclusions show that, the use of memory in EAs for cyclic environments was advantageous. The use of mechanisms that promoted *too much* diversity in the population, combined with memory conferred the worst performance of the algorithms. High diversity can be synonymous of disruption and it is important to find a tradeoff between the use of the memory and the degree of diversity. This tradeoff can allow the algorithms to maintain different individuals for exploring different areas of the search space, without making this search random or disruptive.
When the environment changed slower and the changes were harsher, conjugation and crossover maintained a very low degree of diversity. Therefore, the memorized solutions were prematurely converged and, when a change occurred, those individuals weren't enough to the readaptation of the EA. For those situations, transformation (and a higher diversity) was advantageous since allowed the exploration of other regions of the search space.
Note that the studied algorithms used memory to help its readaptation when a change happened. The genetic operators were useful to continue the search after the retrieval of the memory individuals. So, if the memory had the appropriate information to introduce into the population when a change was detected, and those individuals were not prematurely converged, mechanisms that promoted a higher diversity could disrupt them, slowing the search process for the optimum.

Chapter 11

Prediction: experimental results

This chapter presents and discusses the results obtained using the proposed predictors. First, the accuracy of the predictors is analyzed under different types of environments. Second, the EA using prediction is compared with the same EA without prediction.

11.1 Prediction Accuracy

In this section, the accuracy of the proposed predictors is analyzed. By accuracy we mean the frequency of correct outcomes reached by the proposed predictors. For the Markov model predictor, the predicted value was considered correct when the module provided the correct value for the next environmental transition.
Table 11.1 shows the prediction accuracy of the Markov model based on 500 environmental changes. As the number of different environments increased, the prediction accuracy expectedly decreased. Moreover, for probabilistic dynamics the prediction accuracy was worst. This happened because the Markov model used data collected from previous transitions to estimate the future. As the number of environments increased or the transitions were not deterministic, the Markov model needed more time to learn the entire behavior of the environment.
Figure 11.1 shows how the EA behaved through time using prediction and without prediction. We can see that the EA using prediction went through a *learning* phase - where the Markov model acquired the history of possible environmental transitions - and an *equilibrium* phase - where the Markov model provided the correct predictions. During the equilibrium phase no fitness decrease was observed. On the other hand, the EA without prediction experienced a decrease on its performance every time a change happened. In this case, the recuperation was achieved only after the change, when the information from memory was introduced into the population. The example shows a typical result for the first 50 environmental changes in the dynamic bit matching problem.
The proposed model based on a Markov model provided excellent predictions,

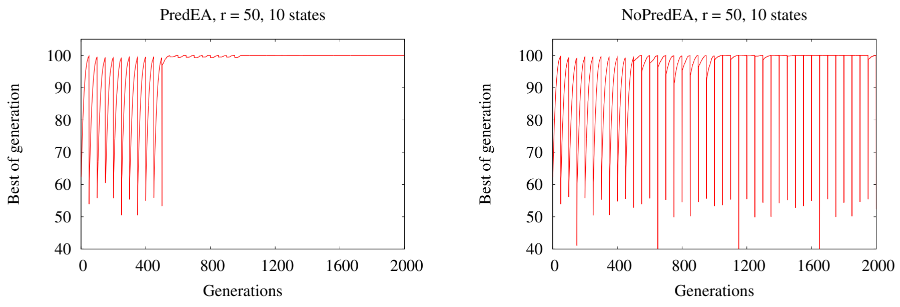


Figure 11.1: Best of generation for **PredEA** and **noPredEA**, bit matching problem

Number of different environments	Type of dynamics	% of correct predictions
3	Cyclic	99.40%
3	Probabilistic	98.80%
5	Cyclic	99.00%
5	Probabilistic	98.00%
10	Cyclic	98.00%
10	Probabilistic	96.20%
20	Cyclic	96.00%
20	Probabilistic	93.60%
50	Cyclic	90.00%
50	Probabilistic	87.00%

Table 11.1: Accuracy of the Markov model predictions

almost always above 90%.
For the linear and nonlinear regression predictors, the accuracy of the predictions provided depended on the type of adjustment used for Δ. The value provided by those predictors was considered accurate if, using the value of Δ, the anticipation was made before the real change happened. If a previous change occurred at generation g_{before} and the predicted value for next change was g_{next}, this value was considered accurate if $g_{next} - \Delta < g_{real}$ and $g_{next} - \Delta \geq g_{before}$ (where g_{real} corresponded to the generation when the change effectively happened and the accuracy of the predicted value was measured). Figure 11.2 shows different cases of good and bad predictions.

The results of Table 11.2, show that the value of Δ influenced the prediction efficacy. For instance, for the 5-10-5 change period, the linear regression predictor using methods Δ_2 and Δ_3 provided the worst prediction accuracy (around 67%) when compared with other methods (around 99%). This happened because the value of Δ achieved by methods Δ_2 and Δ_3 was smaller than the remaining cases and, depending on the prediction error, some situations were not correctly estimated. With the same change period, the nonlinear regression predictor was able to improve the prediction accuracy using different methods for adjusting Δ. The prediction error decreased and smaller values for Δ were obtained. Consequently, the anticipation of the change in the correct time step

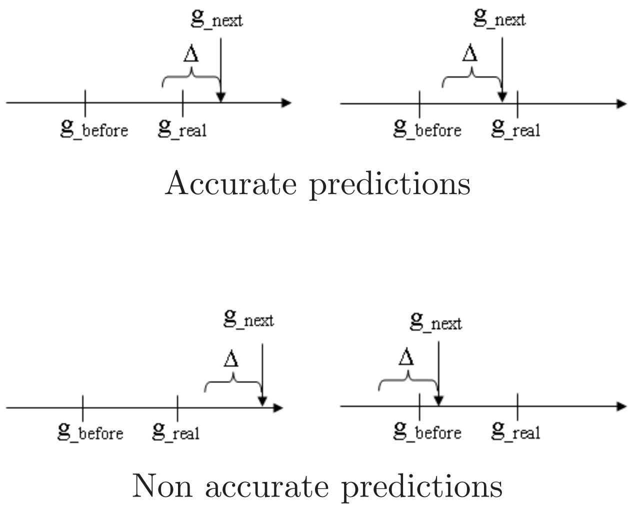


Figure 11.2: Examples of good and bad predictions for the Linear and Nonlinear predictors.

was improved. Using a constant value for Δ the prediction accuracy was slightly decreased when compared with the linear predictor. The reason for this was related to the prediction error obtained by the nonlinear predictor. Since in the 5-10-5 change period the environment changed very quickly, using $\Delta = 5$ with smaller negative prediction errors, increased the number of situations where the condition $g_{next} - \Delta \geq g_{before}$, was not reached.

Change period	Adjustment of Δ	Prediction Accuracy		Average of err		Average of $\lvert err \rvert$		Average of Δ	
		linear	nlinear	linear	nlinear	linear	nlinear	linear	nlinear
	$\Delta = 5$	100.0%	100.0%					5	5
r = 10	Δ_1	100.0%	100.0%					2	2
r = 50	Δ_2	100.0%	100.0%	0.0	0.0	0.0	0.0	2	2
r = 100	Δ_3	100.0%	100.0%					2	2
r = 200	Δ_4	100.0%	100.0%					2	2
	$\Delta = 5$	99.87%	97.86%					5	5
	Δ_1	99.87%	100.0%					5	2
5-10-5	Δ_2	67.38%	100.0%	-0.32	-0.36	2.01	1.01	2	2
	Δ_3	67.65%	100.0%					2	2
	Δ_4	99.87%	100.0%					3	2
	$\Delta = 5$	99.46%	99.73%					5	5
	Δ_1	99.73%	99.73%					10	9
10-20-10	Δ_2	68.10%	66.76%	-0.29	-0.30	2.37	2.36	3	2
	Δ_3	68.63%	99.46%					3	4
	Δ_4	99.73%	66.76%					6	3
	$\Delta = 5$	98.32%	98.32%					5	5
	$\Delta = 10$	99.66%	100.0%					10	10
	Δ_1	99.66%	99.66%					13	8
50-60-70	Δ_2	41.95%	96.64%	-0.29	-0.28	4.43	4.38	3	4
	Δ_3	99.66%	99.66%					4	5
	Δ_4	99.66%	99.66%					8	5
	$\Delta = 5$	65.77%	65.77%					5	5
	$\Delta = 25$	99.33%	100.0%					25	25
	Δ_1	99.66%	66.78%					50	16
100-150-100	Δ_2	67.79%	66.78%	-0.02	-0.02	11.53	10.77	15	12
	Δ_3	67.79%	66.78%					16	13
	Δ_4	99.66%	100.0%					32	28
	$\Delta = 5$	0.00%	97.58%					5	5
	Δ_1	0.00%	97.58%					5	5
Nlinear 1	Δ_2	0.00%	97.58%	-1885.52	-0.85	1885.52	0.85	5	5
	Δ_3	1.21%	98.79%					632	2
	Δ_4	0.00%	97.58%					5	5
	$\Delta = 5$	1.01%	100.0%					5	5
	Δ_1	0.25%	100.0%					775	2
Nlinear 2	Δ_2	0.25%	100.0%	778.56	-0.39	778.56	0.62	410	2
	Δ_3	0.25%	100.0%					410	2
	Δ_4	0.25%	100.0%					593	2
	$\Delta = 5$	4.04%	98.48%					5	5
	Δ_1	7.07%	100.0%					269	2
Nlinear 3	Δ_2	7.07%	100.0%	-485.08	-1.56	1045.57	1.56	147	2
	Δ_3	6.06%	100.0%					532	2
	Δ_4	7.07%	100.0%					208	2
	$\Delta = 5$	5.56%	74.24%					5	5
	Δ_1	10.61%	100.0%					1251	2
Nlinear 4	Δ_2	5.56%	100.0%	1240.23	-0.72	1246.50	0.72	883	2
	Δ_3	6.57%	100.0%					797	2
	Δ_4	5.56%	100.0%					1067	2

Table 11.2: Accuracy of the Linear and Nonlinear Regression Predictors

In general, for the 10-20-10 and 50-60-70 change periods, the nonlinear regression predictor was able to improve the prediction accuracy when compared with the linear predictor. The different methods for adjusting Δ also provided better scores than the use of a constant value. Besides, smaller values of Δ were attained, indicating that the anticipation was made closer to the moment of change, reducing computational costs. The accuracy of the linear and nonlinear regression predictors was not consistent for the 100-150-100 change period. In

some situations the nonlinear predictor obtained better results than the linear predictor, but using the methods Δ_1, Δ_2 and Δ_3, the nonlinear predictor was not so effective. In fact, the values obtained for Δ were not enough to cover the prediction errors. The use of $\Delta = 25$ or the method Δ_4, which used a large value for Δ, obtained better performances.
The use of a constant value for Δ should be used only as last resort, because this method requires some preliminary experimentation before an appropriate Δ value can be determined. This preliminary experimentation is not necessary when using auto-adjusting methods for Δ values.
Despite the different prediction accuracies obtained by the different methods, we can say that both, linear and nonlinear predictors, are good approaches to the problem of estimating when next change will occur, either in periodic or patterned change periods. In the following subsection we will see that both methods effectively increased the performance of the EA.

For the situations where the change period followed a nonlinear trend, the linear predictor failed. The prediction accuracies for the $Nlinear1$ and $Nlinear2$ change periods were close to zero, and for the $Nlinear3$ and $Nlinear4$ change periods they were very weak. Since the predictor was based on linear regression, these results were expected.
In $Nlinear1$ and $Nlinear3$, the predicted values corresponded to generations before the real change (negative prediction errors), but the associated error was enormous and to a great extent the predicted value was not even close to the next change. In the remaining cases ($Nlinear2$ and $Nlinear4$), all the predicted values were provided after the change occurred (positive prediction errors) and the predictor errors were also very high. Even using large values of Δ, the efficacy of the anticipation was very poor.
For nonlinear situations, the nonlinear regression predictor was able to significantly improve the prediction accuracy for the nonlinear situations. For all the situations analyzed the predictor achieved near 100% efficacy, with a small prediction error and small values for Δ. In the linear and patterned change periods, the nonlinear regression predictor was also very effective, proving, for different types of change periods, to be an appropriate method for estimating when the next change would occur.

11.2 Algorithms performance

This section sets forth the results obtained by the EA using the proposed predictors. We will show the results obtained using the first approach, consisting of the Markov model combined with linear regression (**PredEA-LR**) and the second approach, which used the Markov model combined with the nonlinear predictor (**PredEA-NLR**). The results reported correspond to the best results obtained using the different methods for adjusting Δ. Both results are compared with the performance of the EA without any predictor (**noPredEA**). Table 11.3 shows the results for the dynamic bit matching problem, and Table 11.4 contains the scores concerning the dynamic knapsack problem. The statistical results, obtained using the statistical tests, are on Tables 11.5 and 11.6 for the dynamic bit matching problem and the dynamic knapsack problem, respectively. In all tables, adjacent to the number of states is the type of environment: C for cyclic and P for probabilistic.

Bit matching		Number of states									
		3		5		10		20		50	
Period	Algorithm	C	P	C	P	C	P	C	P	C	P
r=10	noPredEA	89.68	90.63	85.99	86.53	80.74	81.97	75.27	76.53	69.74	70.89
	PredEA-LR	98.14	97.32	96.86	96.24	93.72	93.39	90.17	90.14	84.86	85.62
	PredEA-NLR	98.24	97.37	96.89	96.27	93.76	93.49	90.18	90.18	84.89	85.64
r=50	noPredEA	99.01	99.01	98.92	98.92	98.68	98.72	97.26	97.65	89.11	90.74
	PredEA-LR	99.86	99.85	99.79	99.80	99.64	99.58	99.27	99.22	98.20	98.03
	PredEA-NLR	99.91	99.89	99.83	99.81	99.66	99.63	99.32	99.25	98.26	98.05
r=100	noPredEA	99.50	99.50	99.44	99.45	99.31	99.33	99.12	98.82	94.44	95.42
	PredEA-LR	99.93	99.92	99.88	99.87	99.77	99.75	99.54	99.51	98.84	98.71
	PredEA-NLR	99.94	99.93	99.89	99.88	99.78	99.76	99.55	99.52	98.86	98.72
r=200	noPredEA	99.73	99.73	99.70	99.70	99.60	99.61	99.46	99.31	97.17	97.49
	PredEA-LR	99.94	99.94	99.91	99.90	99.82	99.81	99.65	99.63	99.13	99.06
	PredEA-NLR	99.95	99.95	99.92	99.91	99.83	99.82	99.66	99.64	99.14	99.07
5-10-5	noPredEA	85.56	89.94	85.59	86.53	80.97	81.48	74.05	75.73	67.99	69.04
	PredEA-LR	96.57	96.45	94.74	94.02	90.52	90.34	86.88	87.10	94.23	81.53
	PredEA-NLR	96.89	96.76	94.90	94.04	90.58	90.45	86.89	87.26	94.59	81.85
10-20-10	noPredEA	91.43	94.68	92.42	92.47	87.69	88.42	81.00	82.87	73.40	75.00
	PredEA-LR	97.79	98.31	97.67	96.40	95.14	94.22	92.80	92.32	87.22	86.52
	PredEA-NLR	98.10	98.62	97.92	97.20	95.90	95.07	93.12	92.83	87.38	87.02
50-60-70	noPredEA	99.15	99.15	99.06	99.07	98.83	98.86	98.15	97.88	90.78	92.34
	PredEA-LR	99.89	99.86	99.80	99.79	99.62	99.60	99.23	99.18	98.07	97.86
	PredEA-NLR	99.89	99.87	99.82	99.80	99.63	99.61	99.25	99.19	98.09	97.89
100-150-100	noPredEA	99.57	99.57	99.52	99.52	99.41	99.42	99.25	98.98	95.22	96.17
	PredEA-LR	99.92	99.92	99.91	99.90	99.81	99.80	99.62	99.58	99.02	98.91
	PredEA-NLR	99.94	99.94	99.92	99.91	99.83	99.82	99.64	99.59	99.05	98.92
Nlinear 1	noPredEA	98.95	98.94	98.73	98.77	98.38	98.49	97.02	97.41	91.60	92.83
	PredEA-LR	99.19	98.67	99.00	98.67	98.96	98.80	99.12	98.75	99.55	98.46
	PredEA-NLR	99.74	99.68	99.64	99.61	99.48	99.48	99.25	99.22	99.58	98.53
Nlinear 2	noPredEA	98.11	98.14	97.96	97.97	97.58	97.35	97.01	94.40	84.08	85.36
	PredEA-LR	98.12	98.15	97.96	97.97	97.57	97.35	97.01	94.38	84.10	85.43
	PredEA-NLR	99.80	99.79	99.67	99.65	99.33	99.26	98.66	98.53	96.55	96.19
Nlinear 3	noPredEA	99.06	99.07	98.84	98.88	98.48	98.55	97.12	97.47	93.42	94.05
	PredEA-LR	99.34	99.03	99.29	99.17	99.11	99.01	98.51	98.42	96.51	96.65
	PredEA-NLR	99.98	99.92	99.85	99.83	99.60	99.57	99.13	99.07	97.71	97.58
Nlinear 4	noPredEA	99.02	99.06	98.87	98.88	98.52	98.52	97.90	97.44	92.92	93.48
	PredEA-LR	99.07	99.09	98.90	98.91	98.53	98.53	97.91	97.41	92.94	93.57
	PredEA-NLR	99.96	99.94	99.83	99.80	99.49	99.45	98.83	98.73	96.73	96.72

Table 11.3: **PredEA** and **noPredEA** results - dynamic bit matching

The evolution of the algorithms with and without prediction during the entire run is shown, for the dynamic bit matching problem, on figures 11.3 and 11.4 and, for the knapsack problem, on figures 11.5 and 11.6. These results were obtained in the patterned change period (5-10-5) and in the nonlinear change period (Nlinear 2). The figures represent typical results for 5 and 20 different environments using cyclic and probabilistic changes.

Knapsack		Number of states									
		3		5		10		20		50	
Period	Algorithm	C	P	C	P	C	P	C	P	C	P
r=10	noPredEA	1849.57	1853.31	1849.11	1854.60	1840.03	1839.48	1793.66	1799.04	1669.13	1683.74
	PredEA-LR	1859.57	1863.93	1861.35	1867.86	1854.73	1855.57	1808.89	1814.83	1683.73	1699.14
	PredEA-NLR	1859.94	1864.75	1861.52	1868.70	1855.24	1856.36	1809.11	1815.12	1683.92	1699.28
r=50	noPredEA	1860.59	1865.40	1860.78	1867.73	1851.61	1851.68	1804.76	1817.04	1680.30	1687.44
	PredEA-LR	1864.37	1870.19	1865.64	1873.56	1858.86	1860.07	1813.40	1825.64	1688.24	1695.47
	PredEA-NLR	1864.77	1870.24	1866.22	1873.78	1858.98	1860.35	1813.61	1825.76	1688.29	1695.52
r=100	noPredEA	1863.16	1868.04	1864.16	1871.73	1855.27	1853.87	1808.21	1819.53	1683.73	1692.90
	PredEA-LR	1865.51	1870.31	1866.95	1875.17	1859.85	1859.26	1814.54	1825.57	1689.64	1698.64
	PredEA-NLR	1865.82	1870.71	1867.20	1875.57	1860.34	1859.33	1814.83	1825.58	1689.65	1698.78
r=200	noPredEA	1864.06	1867.03	1866.00	1873.43	1858.00	1859.45	1811.18	1825.43	1686.79	1700.56
	PredEA-LR	1865.57	1868.47	1867.84	1875.88	1860.81	1862.25	1815.27	1828.88	1690.74	1704.27
	PredEA-NLR	1865.58	1868.54	1867.95	1875.91	1861.08	1862.26	1815.48	1829.28	1690.80	1704.55
5-10-5	noPredEA	1833.57	1846.72	1846.18	1850.01	1836.80	1838.11	1790.34	1796.52	1665.79	1681.70
	PredEA-LR	1844.16	1859.65	1859.60	1864.39	1852.41	1853.08	1806.60	1812.45	1681.49	1697.05
	PredEA-NLR	1844.55	1860.40	1860.87	1865.03	1852.55	1853.74	1806.86	1813.29	1681.72	1697.84
10-20-10	noPredEA	1837.92	1853.56	1850.43	1855.41	1840.53	1841.94	1794.42	1808.12	1664.74	1679.17
	PredEA-LR	1844.89	1862.31	1859.08	1866.10	1852.67	1853.58	1806.49	1819.42	1674.35	1689.53
	PredEA-NLR	1845.57	1862.50	1859.81	1867.18	1853.32	1853.84	1806.98	1819.49	1674.73	1689.63
50-60-70	noPredEA	1871.61	1865.02	1861.30	1870.12	1851.74	1850.70	1805.38	1815.17	1681.20	1688.86
	PredEA-LR	1874.44	1868.35	1865.55	1875.33	1858.45	1858.15	1812.47	1822.49	1687.39	1695.53
	PredEA-NLR	1874.54	1868.74	1865.65	1875.55	1858.94	1858.21	1812.74	1822.91	1687.65	1695.86
100-150-100	noPredEA	1856.57	1867.50	1863.97	1871.71	1856.16	1855.20	1809.02	1819.75	1684.55	1693.54
	PredEA-LR	1858.19	1870.10	1867.29	1875.43	1860.85	1860.09	1815.30	1826.18	1690.59	1699.55
	PredEA-NLR	1858.72	1870.99	1867.42	1875.66	1861.61	1860.16	1815.97	1826.94	1690.90	1699.73
Nlinear 1	noPredEA	1861.10	1862.12	1860.54	1865.47	1851.44	1851.85	1804.82	1808.31	1678.06	1697.48
	PredEA-LR	1861.74	1865.58	1864.04	1873.50	1858.61	1861.20	1812.95	1817.62	1684.88	1704.58
	PredEA-NLR	1866.05	1867.93	1867.68	1874.38	1861.57	1862.84	1816.24	1819.65	1687.50	1707.01
Nlinear 2	noPredEA	1857.70	1861.36	1858.66	1863.41	1850.50	1850.46	1801.34	1815.55	1670.50	1696.64
	PredEA-LR	1857.87	1860.98	1858.50	1864.31	1849.87	1850.46	1801.46	1815.90	1670.05	1696.36
	PredEA-NLR	1860.73	1864.76	1862.58	1867.70	1855.77	1856.32	1808.55	1822.80	1677.07	1702.87
Nlinear 3	noPredEA	1859.73	1863.16	1858.37	1867.64	1851.47	1855.39	1803.53	1826.09	1680.72	1702.13
	PredEA-LR	1857.44	1863.93	1858.61	1871.32	1854.26	1860.05	1806.57	1830.53	1682.81	1704.85
	PredEA-NLR	1863.31	1867.82	1863.68	1873.22	1857.80	1861.93	1809.82	1832.89	1684.44	1706.71
Nlinear 4	noPredEA	1861.56	1861.67	1858.09	1864.67	1856.50	1850.37	1799.61	1816.59	1650.37	1708.75
	PredEA-LR	1863.19	1861.97	1857.97	1864.69	1856.52	1850.23	1799.81	1816.22	1650.39	1708.48
	PredEA-NLR	1863.91	1863.17	1859.98	1866.44	1859.02	1852.23	1802.30	1819.31	1652.66	1711.08

Table 11.4: **PredEA** and **noPredEA** results - dynamic knapsack

Bit matching		Number of states									
		3		5		10		20		50	
Period	Pair of Algorithms	C	P	C	P	C	P	C	P	C	P
	PredEA-NLR – noPredEA	++	++	++	++	++	++	++	++	++	++
r=10	PredEA-LR – noPredEA	++	++	++	++	++	++	++	++	++	++
	PredEA-NLR – PredEA-LR	++	++	++	++	++	++	+	+	+	+
	PredEA-NLR – noPredEA	++	++	++	++	++	++	++	++	++	++
r=50	PredEA-LR – noPredEA	++	++	++	++	++	++	++	++	++	++
	PredEA-NLR – PredEA-LR	++	++	+	+	++	++	++	++	++	+
	PredEA-NLR – noPredEA	++	++	++	++	++	++	++	++	++	++
r=100	PredEA-LR – noPredEA	++	++	++	++	++	++	++	++	++	++
	PredEA-NLR – PredEA-LR	+	+	+	+	+	+	+	+	+	+
	PredEA-NLR – noPredEA	++	++	++	++	++	++	++	++	++	++
r=200	PredEA-LR – noPredEA	++	++	++	++	++	++	++	++	++	++
	PredEA-NLR – PredEA-LR	+	+	+	+	+	+	+	+	+	+
	PredEA-NLR – noPredEA	++	++	++	++	++	++	++	++	++	++
5-10-5	PredEA-LR – noPredEA	++	++	++	++	++	++	++	++	++	++
	PredEA-NLR – PredEA-LR	++	++	++	+	+	++	+	+	++	++
	PredEA-NLR – noPredEA	++	++	++	++	++	++	++	++	++	++
10-20-10	PredEA-LR – noPredEA	++	++	++	++	++	++	++	++	++	++
	PredEA-NLR – PredEA-LR	++	++	++	++	++	++	++	++	++	++
	PredEA-NLR – noPredEA	++	++	++	++	++	++	++	++	++	++
50-60-70	PredEA-LR – noPredEA	++	++	++	++	++	++	++	++	++	++
	PredEA-NLR – PredEA-LR	+	+	+	+	+	+	+	+	+	++
	PredEA-NLR – noPredEA	++	++	++	++	++	++	++	++	++	++
100-150-100	PredEA-LR – noPredEA	++	++	++	++	++	++	++	++	++	++
	PredEA-NLR – PredEA-LR	+	+	+	+	+	+	+	+	+	+
	PredEA-NLR – noPredEA	++	++	++	++	++	++	++	++	++	++
Nlinear 1	PredEA-LR – noPredEA	++	++	++	++	++	++	++	++	++	++
	PredEA-NLR – PredEA-LR	++	++	++	++	++	++	++	++	++	++
	PredEA-NLR – noPredEA	++	++	++	++	++	++	++	++	++	++
Nlinear 2	PredEA-LR – noPredEA	+	+	+	+	−	+	+	−	+	++
	PredEA-NLR – PredEA-LR	++	++	++	++	++	++	++	++	++	++
	PredEA-NLR – noPredEA	++	++	++	++	++	++	++	++	++	++
Nlinear 3	PredEA-LR – noPredEA	++	+	++	++	++	++	++	++	++	++
	PredEA-NLR – PredEA-LR	++	++	++	++	++	++	++	++	++	++
	PredEA-NLR – noPredEA	++	++	++	++	++	++	++	++	++	++
Nlinear 4	PredEA-LR – noPredEA	+	+	+	++	+	+	+	−	+	++
	PredEA-NLR – PredEA-LR	++	++	++	++	++	++	++	++	++	++

Table 11.5: Statistical results - dynamic bit matching

Knapsack		Number of states									
		3		5		10		20		50	
Period	Pair of Algorithms	C	P	C	P	C	P	C	P	C	P
r=10	PredEA-NLR – noPredEA	++	++	++	++	++	++	++	++	++	++
	PredEA-LR – noPredEA	++	++	++	++	++	++	++	++	++	++
	PredEA-NLR – PredEA-LR	+	+	+	+	+	+	+	+	+	+
r=50	PredEA-NLR – noPredEA	++	++	++	++	++	++	++	++	++	++
	PredEA-LR – noPredEA	++	++	++	++	++	++	++	++	++	++
	PredEA-NLR – PredEA-LR	+	+	+	+	+	+	+	+	+	++
r=100	PredEA-NLR – noPredEA	++	++	++	++	++	++	++	++	++	++
	PredEA-LR – noPredEA	++	++	++	++	++	++	++	++	++	++
	PredEA-NLR – PredEA-LR	+	+	+	+	++	+	+	+	+	+
r=200	PredEA-NLR – noPredEA	++	++	++	++	++	++	++	++	++	++
	PredEA-LR – noPredEA	++	++	++	++	++	++	++	++	++	++
	PredEA-NLR – PredEA-LR	+	++	+	+	++	+	+	+	+	+
5-10-5	PredEA-NLR – noPredEA	++	++	++	++	++	++	++	++	++	++
	PredEA-LR – noPredEA	++	++	++	++	++	++	++	++	++	++
	PredEA-NLR – PredEA-LR	++	+	++	+	+	+	+	+	+	++
10-20-10	PredEA-NLR – noPredEA	++	++	++	++	++	++	++	++	++	++
	PredEA-LR – noPredEA	++	++	++	++	++	++	++	++	++	++
	PredEA-NLR – PredEA-LR	++	+	++	++	+	+	+	+	+	+
50-60-70	PredEA-NLR – noPredEA	++	++	++	++	++	++	++	++	++	++
	PredEA-LR – noPredEA	++	++	++	++	++	++	++	++	++	++
	PredEA-NLR – PredEA-LR	+	+	+	+	+	+	+	+	+	+
100-150-100	PredEA-NLR – noPredEA	++	++	++	++	++	++	++	++	++	++
	PredEA-LR – noPredEA	++	++	++	++	++	++	++	++	++	++
	PredEA-NLR – PredEA-LR	++	++	+	+	++	+	+	++	+	+
Nlinear 1	PredEA-NLR – noPredEA	++	++	++	++	++	++	++	++	++	++
	PredEA-LR – noPredEA	++	++	++	++	++	++	++	++	++	++
	PredEA-NLR – PredEA-LR	++	++	++	++	++	++	++	++	++	++
Nlinear 2	PredEA-NLR – noPredEA	++	++	++	++	++	++	++	++	++	++
	PredEA-LR – noPredEA	+	++	+	++	+	+	+	+	−	−
	PredEA-NLR – PredEA-LR	++	++	++	++	++	++	++	++	++	++
Nlinear 3	PredEA-NLR – noPredEA	++	++	++	++	++	++	++	++	++	++
	PredEA-LR – noPredEA	++	++	++	++	++	++	++	++	++	++
	PredEA-NLR – PredEA-LR	++	++	++	++	++	++	++	++	++	++
Nlinear 4	PredEA-NLR – noPredEA	++	++	++	++	++	++	++	++	++	++
	PredEA-LR – noPredEA	++	+	++	+	+	−	+	−	+	−
	PredEA-NLR – PredEA-LR	++	++	++	++	++	++	++	++	++	++

Table 11.6: Statistical results - dynamic knapsack

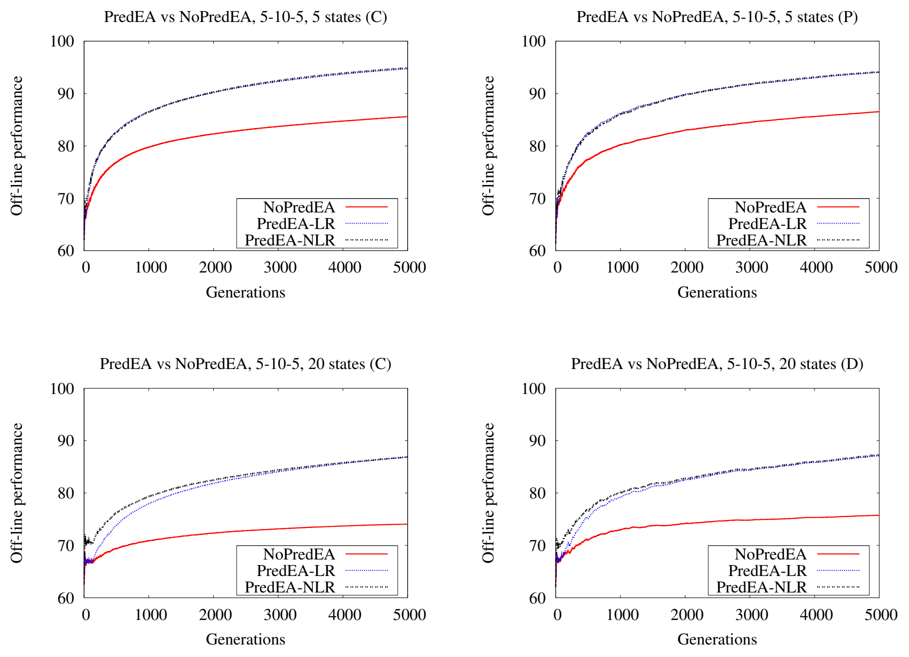


Figure 11.3: Off-line performance for the bit matching problem using PredEA-LR, PredEA-NLR and NoPredEA for 5-10-5 change period

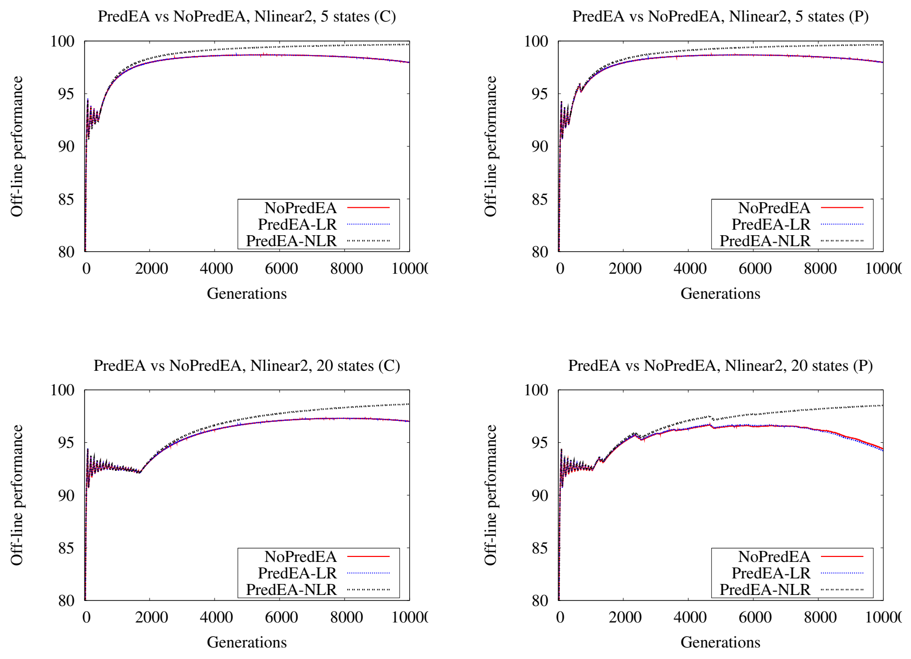


Figure 11.4: Off-line performance for the bit matching problem using PredEA-LR, PredEA-NLR and NoPredEA for Nlinear 2 change period

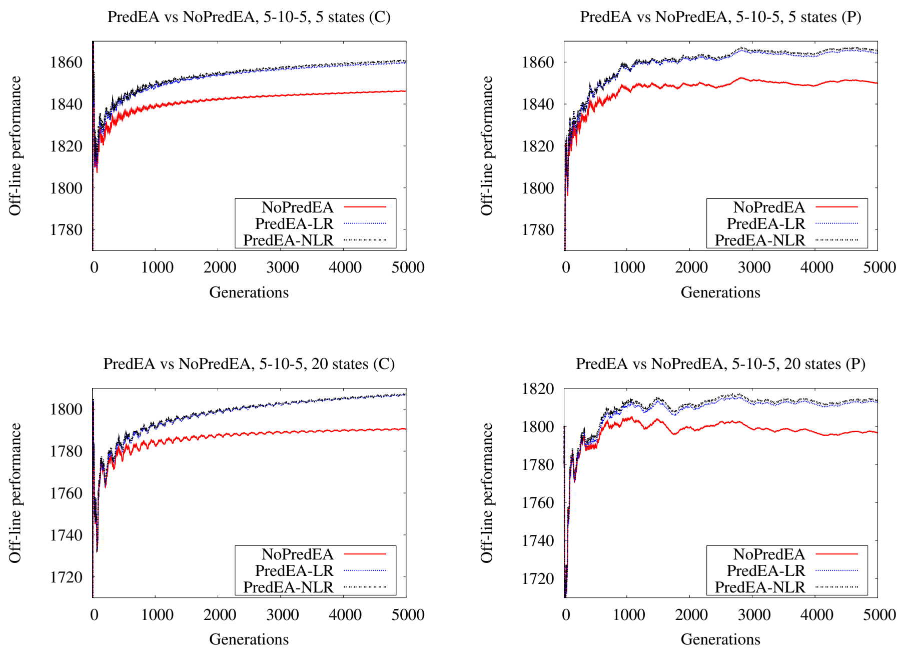


Figure 11.5: Off-line performance for the knapsack problem using PredEA-LR, PredEA-NLR and NoPredEA for 5-10-5 change period

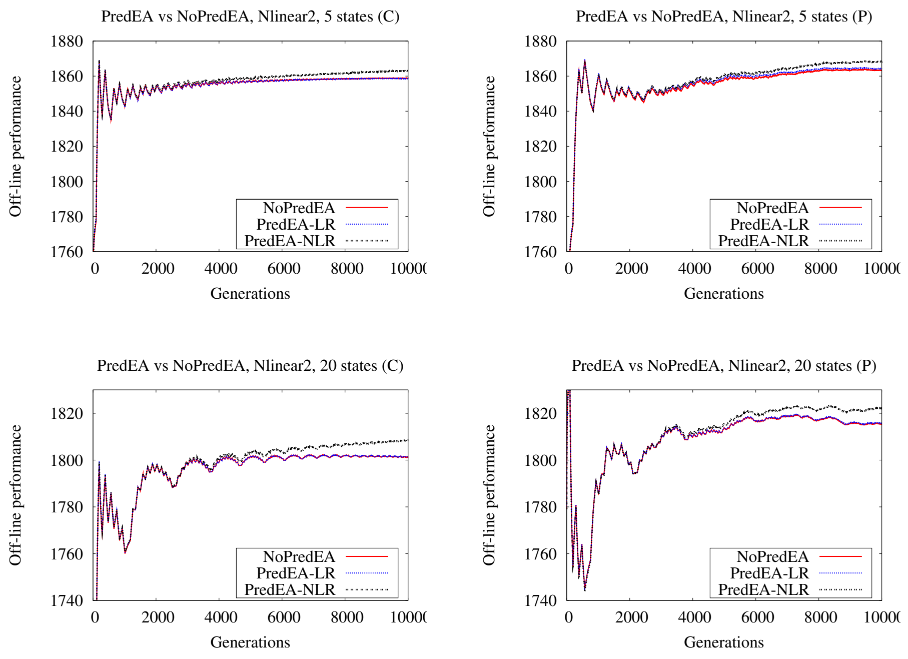


Figure 11.6: Off-line performance for the knapsack problem using PredEA-LR, PredEA-NLR and NoPredEA for Nlinear 2 change period

11.3 Discussion

When in the presence of environments that change following a repeated behavior, the use of prediction mechanisms is highly beneficial to the performance of memory-based EAs. By using past data, accurate predictions when the next change will occur can be made. Consequently, the algorithm can anticipate the change by introducing useful information into the population before such change takes place. In our investigations, we tested two different predictors to estimate when the next change would occur: one using linear regression, and another using nonlinear regression. We saw that the linear regression predictor was appropriated for cyclic or patterned change periods, but failed in the presence of nonlinear change periods. On the other hand, the nonlinear predictor provided good predictions in all types of environments analyzed. The main drawback of the nonlinear predictor was that the nonlinear function used to make the predictions should be known, and the estimation of the nonlinear parameters was time consuming. Figure 11.7 shows the computational times of a single run spent by the three algorithms.

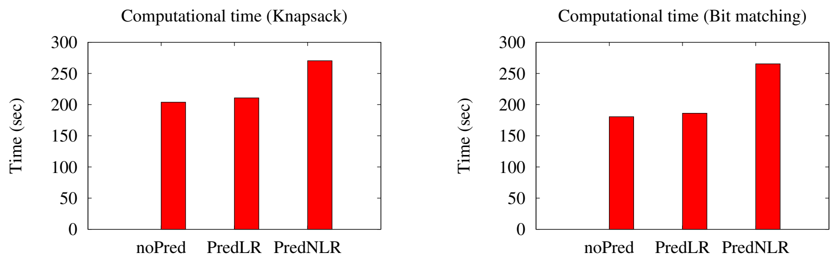


Figure 11.7: Computational times for noPredEA, PredEA-LR and PredEA-NLR - Nlinear 2 change period, 20 states, probabilistic changes

The second predictor, responsible to estimate how the environment would change, was implemented using Markov chains. The Markov chain stored information about the environments and the transitions among them. After that, this information was used to predict which possible environment(s) would appear in the next change. This predictor performed very well in the situations analyzed: it started by learning the dynamics of the environmental changes and, after that phase, the predictions provided were, for the most part, correct. When the number of different environments increased, the predictor needed more time to acquire all the necessary information in order to make valid predictions. The main limitation of this predictor was the problem dependance, that is, the information stored in each state of the Markov model corresponded to the fitness function that was being used.
Notwithstanding the foregoing limitations, the work carried out on this thesis provided an important contribution on this topic. No relevant work existed before, using this kind of methods in memory-based EAs. The results obtained showed that prediction combined with memory highly increased the performance of the EA and should be further investigated.

Chapter 12

Conclusion

This final chapter summarizes the main contributions of this thesis and the results obtained in the experiments. Future directions of this work are also discussed.

12.1 Summary

The main goal of this thesis is to improve memory-based EAs on how to deal with dynamic optimization problems. This type of algorithms is clearly beneficial when the environmental changes follow some repeated behavior. The basic principle is to keep in memory past good solutions and/or environmental information so it can later be used with benefit. Throughout this thesis, several aspects and problems of classical memory-based EAs were revised:

- The relationship between the size of the memory and that of the population. Memory size is usually chosen as a small percentage of the global number of individuals, and this choice is seldom the best option;
- The limited size of the memory itself, implying that it is necessary to decide whether and which individual should be replaced by a new one;
- The combination of memory and preservation of diversity, which were assumed to benefit EAs in solving dynamic problems;
- The retrieval of memorized individuals, which usually occurs after a change takes place and before the readaptation of the EA to the new environment, resulting in a decrease of the EA's performance.

This work has introduced approaches that focus on each one of the foregoing topics:

- An exhaustive experimental study about the influence of the memory and population sizes was taken and a new algorithm - called Variable-Memory Evolutionary Algorithm (**VMEA**) - using population and memory of variable size, was proposed;

- Three new replacing strategies, called *age1*, *age2* and *generational*, were introduced and tested on different memory-based EAs;
- Two new genetic operators for promoting diversity, called conjugation and transformation, were introduced and tested, replacing the standard crossover operator on different memory-based EAs. The experimentation taken on different types of dynamic environments allowed us to draw important conclusions about the advantage of increasing the population's diversity;
- Two prediction modules were incorporated in the memory-based EA. One module was responsible for predicting when the next change would happen. This module was tested using linear and nonlinear regression techniques. The second module, based on Markov models, gathered information about the environmental transitions and provided prediction about how the environment would change in the future. The combination of these two modules interplay with memory and the ability of make good predictions allowed for the introduction of useful memorized information into the population **before** the change took place.

The proposed methods were tested with different benchmark problems and the performance of different memory-based EAs was compared. We concluded that the population and memory sizes had a significant impact in the performance of different memory-based EAs. The results showed that the traditional choice, which uses memory of smaller proportions when compared with the population size, was rarely the best choice. Unfortunately, it was not possible to point out a general rule to decide which memory and population sizes should be used, because they depended on the problem, the characteristics of the environment, and on the algorithm. An important algorithm was introduced and studied in this thesis. This algorithm, called Variable-size Memory Evolutionary Algorithm (**VMEA**) used the global number of individuals distributed in the memory and the search population in a variable manner during the run. Our study showed that this algorithm achieved superior results when compared with the algorithms using constant population and memory sizes.

The mechanisms that manage the memorized solutions and decide which individuals should be replaced when the memory is full are very important in the performance of memory-based EAs. This is obvious, since if we store *bad* individuals, replacing *good* ones, the EAs will not be able to readapt when a change occurs. We proposed three different replacing schemes and compared them with the popular **similar** method proposed by Branke. The analysis of the results showed that the replacing mechanisms based on age were difficult to tune, since it was difficult to find appropriated values for the age of the individuals. The proposed age-based methods obtained good results in a minority of the situations analyzed. On the other hand, the *generational* mechanism introduced and studied in this thesis proved to be an efficient scheme to decide which memory individuals should be replaced. The performance of all the algorithms analyzed was significantly improved in most of the environments studied.

For the past years, different mechanisms that promote and maintain the population diversity have been proposed and used in EAs for dynamic environments. Recently, some studies showed that in some situations the use of high diversity could be detrimental to the performance of EAs for dynamic environments. In this thesis we introduced two new genetic operators and used them as alternative to the traditional crossover operator. We were concerned in studying the diversity of the population maintained by these two mechanisms, and seeing if the performance of the algorithms was affected by it. We used the two genetic operators, as well as the uniform crossover in four different memory-based EAs, and analyzed the relation between the population's diversity and the algorithm's performance. The results showed that the proposed methods generated different values of diversity and different performances were obtained. In general, the best performances were obtained by conjugation that was able to preserve the lowest diversity. Transformation (highest diversity) performed better in slower environments with severer changes. Finally, when the diversity promoted for crossover and conjugation was similar, the performance of the algorithms was also equivalent.

The last contribution of this thesis consisted of the introduction of mechanisms capable of using information from the past to correctly predict the future. The mechanisms introduced in the memory-based EA were used to predict **when** the next generation would occur and **how** the environment would change. To predict when a change would take place, we introduced two predictors: one based on linear regression and another based on nonlinear regression. In order to forecast how the environment would change, we used a Markov chain model to keep track of past environments (and the transitions among them) and to use that information to predict the future environment. The prediction modules were incorporated in a standard memory-based EA, which was tested on different situations. Results showed the effectiveness of the proposed methods, which gave accurate predictions in most of the situations analyzed. The performance of the memory-based EA was significantly improved by the mechanisms proposed. We showed that, in cyclic environments, not only was prediction attainable, but also that anticipating the changes resulted in a superior performance. The proposed predictors are problem-dependent and can fail in some types of nonlinear change periods. These shortcomings deserve further investigation in order to develop predictors that are more flexible and adaptable.

12.2 Future Work

Several topics studied in this thesis yield results that indicate that extra work should be done. For instance, it is important to determine whether the proposed **VMEA** can also achieve good performances in other types of environments besides cyclic ones. In this regard, some preliminary experimentation was performed and published in [91], but further investigation is still needed. Also, different types of dynamic environments should be used to evaluate the proposed replacing strategies. This is a work in progress, and should be continued.

The results related to diversity were very interesting and have spur our curiosity in studying the real importance of diversity as it pertains to EAs for dynamic environments. We are extending our experimentations with memory-based EAs to other types of environments. Moreover, different mechanisms for promoting and maintaining diversity are also being used and tested in EAs without memory. This work is in progress.
Finally, the prediction mechanisms must be improved. Both linear and nonlinear regression predictors must be tested using the concept of *time window*: instead of using all the available information from the beginning of the run, we intend to use only a small number of past observations, enclosed by a time window. This can reduce the computational effort without hindering the prediction performance. Moreover, we intend to evolve, through genetic programming, the function that is used by the nonlinear predictor. Concerning the Markov model, some enhancements must be introduced in order to make this module independent from the problem. All these prediction-related subjects are currently under study.

Bibliography

[1] P. Angeline. Tracking extrema in dynamic environments. In *Proceedings oth the Sixth Annual Conference on Evolutionary Programming (EP VI)*, volume 1213 of *Lecture Notes on Computer Science*, pages 335–345. Springer, 1997.

[2] J. Arabas, Z. Michalewicz, and J. Mulawka. Gavaps - a genetic algorithm with varying population size. In F. Varela and P. Bourgine, editors, *Proceedings of the First IEEE Conference on Evolutionary Computation (CEC 1994)*, pages 73–78. IEEE Press, 1994.

[3] T. Bäck. Self adaptation in genetic algorithms. In F. Varela and P. Bourgine, editors, *Toward a Practice of Autonomous Systems: Proceedings of the First Conference on Artificial Life*, pages 263–271. MIT Press, 1992.

[4] T. Bäck. Optimal mutation rates in genetic search. In S. Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms (ICGA 1993)*, pages 2–8. Morgan Kaufmann, 1993.

[5] T. Bäck, A. E. Eiben, and N. A. L. van der Vaart. An empirical study on gas 'without parameters'. In M. Schoenauer, K. Deb, G. Rudolf, X. Yao, E. Lutton, J. J. J. Merelo, and H.-P. Schwefel, editors, *Proceedings of Parallel Problem Solving from Nature (PPSN V)*, volume 1917 of *Lecture Notes in Computer Science*, pages 315–324. Springer, 2000.

[6] S. Baluja. Population-based incremental learning: a method for integrating genetic search based function optimization and competitive learning. Technical Report TR CMU-CS-94-163, Carnegie Mellon University, 1994.

[7] G. J. Barlow and S. F. Smith. A memory enhanced evolutionary algorithm for dynamic scheduling problems. In Springer, editor, *Applications of Evolutionary Computing*, volume 4974 of *Lecture Notes in Computer Science*, pages 606–615, 2008.

[8] C. N. Bendtsen and T. Krink. Dynamic memory model for non-stationary optimization. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC 2002)*, pages 145–150. IEEE Press, 2007.

[9] C. Bierwirth, K. Kopfer, D. Mattfeld, and I. Rixen. Genetic algorithm based sceduling in a dynamic manufacturing environment. In *Proceedings*

of the IEEE Congress on Evolutionary Computation (CEC 1995). IEEE Press, 1995.

[10] C. Bierwirth and D. Mattfeld. Production scheduling and rescheduling with genetic algorithms. *Evolutionary Computation*, 7(1):1–17, 1999.

[11] P. A. N. Bosman and H. L. Poutré. Computationally intelligent online dynamic vehicle routing by explicit loa prediction in evolutionary algorithm. In T. P. Runarsson, H.-G. Beyer, E. Burke, J. Merelo-Guervós, L. D. Whitley, and X. Yao, editors, *Proceedings of Parallel Problem Solving from Nature (PPSN IX)*, Lecture Notes in Computer Science 4193, pages 312–321. Springer-Verlag, 2006.

[12] P. A. N. Bosman and H. L. Poutré. Inventory management and the impact of anticipation in evolutionary stochastic online dynamic optimization. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC 2007)*, pages 268–275. IEEE Press, 2007.

[13] J. Branke. Memory enhanced evolutionary algorithms for changing optimization problems. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC 1999)*, pages 1875–1882. IEEE Press, 1999.

[14] J. Branke. *Evolutionary Optimization in Dynamic Environments.* Kluwer Academic Publishers, 2002.

[15] J. Branke, T. Kaußler, and C. Schmidt. A multi-population approach to dynamic optimization problems. In I. Parmee, editor, *Proceedings of Adaptsim03ive Computing in Design and Manufacture (ACDM 2000)*, pages 299–308. Spriger-Verlag, 2000.

[16] J. Branke and D. Mattfeld. Anticipation in dynamic optimization: The scheduling case. In M. Schoenauer, K. Deb, G. Rudolph, X. Yao, E. Lutton, J. Merelo, and H.-P. Schwefel, editors, *Parallel Problem Solving from Nature*, pages 253–262, 2000.

[17] W. Cedeno and V. R. Vemuri. On the use of niching for dynamic landscapes. In *Proceedings of the International Conference on Evolutionary Computation (ICEC 1997)*, pages 361–366. IEEE Press, 1997.

[18] H. Cheng and S. Yang. Genetic algorithms with elitism-based immigrants for dynamic shortest path problem in mobile ad hoc networks. In *Proceedings of the 2009 IEEE Congress on Evolutionary Computation (CEC 2009*, pages 3135–3140. IEEE Press, 2009.

[19] H. G. Cobb. An investigation into the use of hypermutation as an adaptive operator in genetic algorithms having continuous, time-dependent nonstationary environments. Technical Report TR AIC-90-001, Naval Research Laboratory, 1990.

[20] H. G. Cobb and J. J. Grefenstette. Genetic algorithms for tracking changing environments. In *Proceedings of the Fifth International Conference on Genetic Algorithms (ICGA 1993)*, pages 523–530. Morgan Kaufmann, 1993.

[21] G. P. M. D. S. Moore. *Introduction to the Practice of Statistics (4th edition).* Freeman and Company, 2003.

[22] L. Davis. Adapting operator probabilities in genetic algorithms. In J. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms (ICGA 1989)*, pages 61–69. Morgan Kaufmann, 1989.

[23] A. E. Eiben, R. Hinterding, and Z. Michalewicz. Parameter control in evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, 2(3):124–141, 1999.

[24] A. E. Eiben, E. Narchiori, and V. A. Valkó. Adapting operator probabilities in genetic algorithms. In X. Yao and et al., editors, *Proceedings of Parallel Problem Solving from Nature (PPSN VIII)*, volume 3242 of *Lecture Notes in Computer Science*, pages 41–50. Springer, 1989.

[25] A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing.* Springer, 2003.

[26] M. Farina, K. Deb, and P. Amato. Dynamic multiobjective optimization problem: test cases, approximation and applications. In E. Cantú-Paz and et al., editors, *Proceedings of the Fifth International Genetic and Evolutionary Computation Conference (GECCO 2003)*, volume 2723 of *Lecture Notes in Computer Science*, pages 311–326. Springer, 2003.

[27] M. Farina, K. Deb, and P. Amato. Dynamic multiobjective optimization problem: test cases, approximation and applications. *IEEE Transactions on Evolutionary Computation*, 8(5):425–442, 2004.

[28] D. Floreano and C. Mattiussi. *Bio-Inspired Artificial Intelligence: Theories, Methods and Technologies.* MIT Press, 2008.

[29] L. Fogel, A. Owens, and M. Walsh. *Artificial Intelligence Through Simulated Evolution.* John Wiley & Sons, Inc, 1966.

[30] D. E. Goldberg, K. Deb, and J. H. Clark. Genetic algorithms, noise and the sizing of populations. *Complex Systems*, (6):333–362, 1992.

[31] D. E. Goldberg and J. Richardson. Genetic algorithms with sharing for multimodal function optimization. In J. J. Grefenstette, editor, *Proceedings of the Second International Conference on Genetic Algorithms (ICGA 1987)*, pages 41–49. Lawrence Erlbaum Associates, 1987.

[32] D. E. Goldberg and R. E. Smith. Nonstationary function optimization using genetic algorithms with dominance and diploidy. In J. J. Grefenstette, editor, *Proceedings of the Second International Conference on Genetic Algorithms (ICGA 1987)*, pages 59–68. Lawrence Erlbaum Associates, 1987.

[33] J. L. Gould and W. T. Keeton. *Biological Science.* W. W. Norton & Company, 1996.

[34] J. J. Grefenstette. Optimisation of control parameters for genetic algorithms. *IEEE Transactions on Systems, Man and Cybernetics*, 1(16):122–128, 1986.

[35] J. J. Grefenstette. Genetic algorithms for changing environments. In R. Männer and B. Manderick, editors, *Parallel Problem Solving from Nature (PPSN II)*, 1992.

[36] J. J. Grefenstette and C. L. Ramsey. An approach to anytime learning. In D. Sleeman and P. Edwards, editors, *Proceedings of the Ninth International Conference on Machine Learning*, pages 189–195. Morgan Kaufmann, 1992.

[37] I. Harvey. The microbial genetic algorithm. 1996.

[38] I. Hatzakis and D. Wallace. Dynamic multi-objective optimization with evolutionary algorithms: A forward-looking approach. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2006)*, pages 1201–1208. ACM Press, 2001.

[39] J. H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology.* University of Michigan Press, 1975.

[40] Y. Jin and B. Sendhoff. Constructing dynamic optimization test problems using the multi-objective optimization concept. In G. Raidl and et al., editors, *Applications of Evolutionary Algorithms*, volume 3005 of *LNCS*, pages 525–536. Springer, 2004.

[41] K. D. Jong. *An Analysis of the Behavior of a Class of Genetic Adaptive Systems.* PhD thesis, University of Michigan, 1975.

[42] K. D. Jong. *Evolutionary Computation: a unified approach.* MIT Press, 2006.

[43] A. Karaman, S. Uyar, and G. Eryigit. The memory indexing evolutionary algorithm for dynamic environments. In *Applications of Evolutionary Computing*, volume 3449 of *Lecture Notes in Computer Science*, pages 563–573. Springer, 2005.

[44] C. L. Karr. Genetic algorithms and fuzzy logic for adaptive process control. In S. Goonatilake and S. Khebbal, editors, *Intelligent Hybrid Systems*, volume 4, pages 63–83. John Wiley, 1995.

[45] M. D. Kidwell and D. J. Cook. Genetic algorithm for dynamic task scheduling. In *Proceedings of the Thirteenth International Phoenix Conference on Computers and Communications*, pages 61–67. IEEE Press, 1994.

[46] J. R. Koza. Genetic programming: On the programming of computers by means of natural selection. *Statistics and Computing*, 4(2):87–112, 1994.

[47] J. R. Koza, J. P. Rice, and J. Roughgarden. Evolution of food foraging strategies for the caribbean anolis lizard using genetic programming. *Adaptive Behavior*, 1(2):171–199, 1992.

[48] E. H. J. Lewis and G. Ritchie. A comparison of dominance mechanisms and simple mutation on non-stationary problems. In M. Schoenauer, K. Deb, G. Rudolf, X. Yao, E. Lutton, J. J. J. Merelo, and H.-P. Schwefel, editors, *Proceedings of the Parallel Problem Solving from Nature (PPSN V)*, volume 1917 of *Lecture Notes on Computer Science*, pages 139–148. Springer, 1998.

[49] C. Li and S. Yang. A generalized approach to construct benchmark problems for dynamic optimization. In X. Li and et al., editors, *Proceedings of the 7th International Conference on Simulated Evolution and Learning (SEAL 2008)*, volume 5361 of *Lecture Notes on Computer Science*, pages 391–400. Springer, 2008.

[50] Q. Ling, G. Wu, and Q. Wang. Deterministic robust optimal design based on standard crowding genetic algorithm. In S. Yang, Y.-S. Ong, and Y. Jin, editors, *Evolutionary Computation in Dynamic and Uncertain Environments*, volume 51 of *Studies in Computational Intelligence*, pages 583–598. Springer-Verlag, 2007.

[51] L. Liu, D. Wang, and S. Yang. An immune system based genetic algorithm using permutation-based dualism for dynamic traveling salesman problems. In M. Giacobini and et al., editors, *EvoWorkshops 2009: Applications of Evolutionary Computing (EVOSTOC 2009)*, volume 5484 of *Lecture Notes on Computer Science*, pages 725–734. Springer, 2009.

[52] F. Lobo and C. F. Lima. Revisiting evolutionary algorithms with on-the-fly population adjustment. In M. Keijzer and et al., editors, *Proceedings of the Eighth International Genetic and Evolutionary Computation Conference (GECCO 2006)*, pages 1241–1248. ACM Press, 2006.

[53] S. J. Louis and Z. Xu. Genetic algorithms for open shop scheduling and rescheduling. In M. E. Cohen and D. L. Hudson, editors, *Proceedings of the Eleventh International Conference on Computers and their Applications (ISCA)*, pages 99–102, 1996.

[54] R. I. Lung and D. Dumitrescu. Evolutionary swarm cooperative optimization in dynamic environments. *Natural Computing*, 2009.

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

[56] M. Mitchell, S. Forrest, and J. Holland. The royal road for genetic algorithms: fitness landscape and ga performance. In F. J. Varela and P. Bourgine, editors, *Proceedings of the First European Conference on Arti*

cial Life, pages 245–254. MIT Press, 1992.

[57] N. Mori, H. Kita, and Y. Nishikawa. Adaptation to a changing environment by means of the thermodynamical genetic algorithm. In H.-M. Voigt, editor, *Parallel Problem Solving from Nature (PPSN IV)*, volume 1141 of *Lecture Notes in Computer Science*, pages 513–522, 1996.

[58] N. Mori, H. Kita, and Y. Nishikawa. Adaptation to changing environments by means of the memory-based thermodynamical genetic algorithm. In I. Bäck, editor, *Proceedings of the Seventh International Conference on Genetic Algorithms (ICGA 1997)*, pages 299–306. Morgan Kaufmann, 1997.

[59] N. Mori, H. Kita, and Y. Nishikawa. Adaptation to a changing environment by means of the feedback thermodynamical genetic algorithm. In *Parallel Problem Solving from Nature (PPSN V)*, volume 1498 of *Lecture Notes in Computer Science*, pages 149–158, 1998.

[60] R. W. Morrison. *Designing Evolutionary Algorithms for Dynamic Environments*. Springer, 2004.

[61] R. W. Morrisoni and K. D. Jong. A test problem generator for nonstationary environments. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC 1999)*, pages 2047–2053. IEEE Press, 1999.

[62] R. W. Morrisoni and K. D. Jong. Triggered hypermutation revisited. In *Proceedings of the 2000 Congress on Evolutionary Computation (CEC 2000)*, pages 1025–1032. IEEE Press, 2000.

[63] J. C. Nash and M. Walker-Smith. *Nonlinear Parameter Estimation: an integrated system in BASIC*. Marcel Dekker, Inc, 1987.

[64] P. Ng and K. C. Wong. A new diploid scheme and dominance change mechanism for nonstationary function optimization. In *Proceedings of the Sixth International Conference on Genetic Algorithms (ICGA 1995)*, pages 159–166. Morgan Kaufmann, 1995.

[65] F. Oppacher and M. Wineberg. The shifting balance genetic algorithm. In W. Banzhaf and et al., editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 1999)*, pages 504–510. Morgan Kaufmann, 1999.

[66] Z. Pan, Y. Chen, L. Kan, and Y. Zhang. Parameter estimation by genetic algorithms for nonlinear regression. In *Proc. of the International Conference on Optimization Techniques and Applications*, pages 946–953. World Scientific, 1995.

[67] T. Park, R. Choe, and K. R. Ryu. Dual-population genetic algorithm for nonstationary optimization. In *Proceedings of the Tenth International Genetic and Evolutionary Computation. Conference (GECCO 2008)*, pages 1025–1032. ACM Press, 2008.

[68] C. L. Ramsey and J. J. Grefenstette. Case-based initialization of genetic algorithms. In S. Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 84–91. Morgan Kaufmann, 1993.

[69] I. Rechenberg. *Evolutionsstrategie;: Optimieruntechnischer Systeme nach Prinzipien der biologischen Evolution.* Cambridge Series in Statistical and Probabilistic Mathematics. Fromman-Holzboogg (German Edition), 1973.

[70] H. Richter and S. Yang. Memory-based on abstraction for dynamic fitness functions. In M. Giacobini and et al., editors, *Applications of Evolutionary Computing*, volume 4974 of *Lecture Notes in Computer Science*, pages 597–606. Springer-Verlag, 2008.

[71] H. Richter and S. Yang. Learning behavior in abstract memory schemes for dynamic optimization problems. *Soft Computing*, 13(12):1163–1173, 2009.

[72] C. Rossi, M. Abderrahim, and J. C. Díaz. Tracking moving optima using kalman-based predictions. *Evolutionary Computation*, 16(1):1–30, 2008.

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

[74] C. Ryan. Dyploidy without dominance. In J. T. Alander, editor, *Proceedings of the Nordic Workshop on Genetic Algorithms*, pages 63–70, 1997.

[75] K. Sastry, H. A. Abbass, and D. E. Goldberg. Sub-structural niching in non-stationary environments. Technical Report 2004035, Illinois Genetic Algorithms Laboratory (IlliGAL), 2004.

[76] M. Schmidt, Z. Michalewicz, M. Michalewicz, and C. Chiriac. Prediction and optimization in a dynamic environment: a case study. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC 2005)*, volume 1, pages 781–788. IEEE Press, 2005.

[77] I. Schoeman and A. Engelbrecht. Niching for dynamic environments using particle swarm optimization. In *Simulated Evolution and Learning*, volume 4247, pages 134–141. Springer, 2006.

[78] L. Schönemann. On the influence of population sizes in evolution strategies in dynamic environments. In E. Cantú-Paz and et al., editors, *EvoDOP Workshop, Fifth International Genetic and Evolutionary Computation Conference (GECCO 2003)*, volume 2723 of *Lecture Notes in Computer Science*, pages 123–127. Springer, 2003.

[79] L. Schönemann. The impact of population sizes and diversity on the adaptability of evolution strategies in dynamic environments. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC 2004)*, volume 2, pages 1270–1277. IEEE Press, 2004.

[80] A. Simões and E. Costa. Improving prediction in evolutionary algorithms for dynamic environments. In *Proceedings of the 11th International Genetic and Evolutionary Computation Conference (GECCO 2009)*, pages 875–888. ACM Press.

[81] A. Simões and E. Costa. On biologically inspired genetic operators: Transformation in the standard genetic algorithm. In L. Spector, E. Goodman, A. Wu, W. Langdon, H.-M. Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M. Garzon, and E. Burke, editors, *Proceedings of the 3th International Genetic and Evolutionary Computation Conference (GECCO 2001)*, pages 584–591. Morgan Kaufmann.

[82] A. Simões and E. Costa. Parametric study to enhance the genetic algorithm's performance when using transformation. In *Proceedings of the 4th International Genetic and Evolutionary Computation Conference (GECCO 2002)*. Morgan Kaufmann.

[83] A. Simões and E. Costa. Prediction in evolutionary algorithms for dynamic environments using markov chains and nonlinear regression. In *Proceedings of the 11th International Genetic and Evolutionary Computation Conference (GECCO 2009)*, pages 883–890. ACM Press.

[84] A. Simões and E. Costa. Using biological inspiration to deal with dynamic environments. In *Proceedings of the Seventh International Conference on Soft Computing (MENDEL 2001)*. Brno University Press.

[85] A. Simões and E. Costa. Using genetic algorithms to deal with dynamic environments: A comparative study of several approaches based on promoting diversity. In *Proceedings of the 4th International Genetic and Evolutionary Computation Conference (GECCO 2002)*. Morgan Kaufmann.

[86] A. Simões and E. Costa. A comparative study using genetic algorithms to deal with dynamic environments. In D. W. Pearson, N. C. Steele, and R. Albrecht, editors, *Proceedings of the 6th International Conference on Artificial Neural Networks (ICANNGA 2003)*, pages 203–209. Springer-Verlag, 2003.

[87] A. Simões and E. Costa. An immune system-based genetic algorithm to deal with dynamic environments: Diversity and memory. In D. W. Pearson, N. C. Steele, and R. Albrecht, editors, *Proceedings of the 6th International Conference on Artificial Neural Networks (ICANNGA 2003)*, pages 168–174. Springer-Verlag, 2003.

[88] A. Simões and E. Costa. Improving the genetic algorithm's performance when using transformation. In D. W. Pearson, N. C. Steele, and R. Albrecht, editors, *Proceedings of the 6th International Conference on Artificial Neural Networks (ICANNGA 2003)*, pages 175–181. Springer-Verlag, 2003.

[89] A. Simões and E. Costa. Improving memory's usage in evolutionary algorithms for changing environments. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC 2007)*, pages 276–283. IEEE Press, 2007.

[90] A. Simões and E. Costa. Variable-size memory evolutionary algorithm: Studies on replacing strategies and diversity in dynamic environments.

In D. Thierens and et al., editors, *Proceedings of the 9th International Genetic and Evolutionary Computation Conference (GECCO 2007)*, page 1530. ACM Press, 2007.

[91] A. Simões and E. Costa. Variable-size memory evolutionary algorithm to deal with dynamic environments. In M. G. et al., editor, *Applications of Evolutionary Computing*, volume 4448 of *Lecture Notes in Computer Science*, pages 617–626. Springer, 2007.

[92] A. Simões and E. Costa. Evolutionary algorithms for dynamic environments: Prediction using linear regression and markov chains. In *Parallel Problem Solving from Nature (PPSN X)*, volume 5199 of *Lecture Notes on Computer Science*, pages 306–315. Springer, 2008.

[93] A. Simões and E. Costa. The influence of population and memory sizes on the evolutionary algorithm's performance for dynamic environments. In M. Giacobini and et al., editors, *Applications of Evolutionary Computing*, volume 5484 of *Lecture Notes on Computer Science*, pages 705–714. Springer-Verlag, 2009. This paper received the Best Paper Award of EVOSTOC 2009.

[94] P. Smith. Conjugation: A bacterially inspired form of genetic. In *Late Breaking Papers at the Genetic Programming 1996 Conference*, 1996.

[95] P. Smith. Finding hard satisfiability problems using bacterial conjugation. In *AISB Workshop on Evolutionary Computing*, pages 236–244, 1996.

[96] P. D. Stroud. Kalman-extended genetic algorithm for search in nonstationary environments with noisy fitness evaluations. *IEEE Transactions on Evolutionary Computation*, 5(1):66–77, 2001.

[97] R. Tinos and S. Yang. A self-organizing random immigrants genetic algorithm for dynamic optimization problems. *Genetic Programming and Evolvable Machines*, 3(8):255–286, 2007.

[98] R. Tinós and S. Yang. Continuous dynamic problem generators for evolutionary algorithms. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC 2007)*, pages 236–243. IEEE Press, 2007.

[99] K. Trojanowski and Z. Michalewicz. Evolutionary algorithms for nonstationary environments. In *Proceedings of 8th Workshop on Intelligent Information Systems*, 1999.

[100] K. Trojanowski and Z. Michalewicz. Searching for optima in nonstationary environments. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC 1999)*, pages 1843–1850. IEEE Press, 1999.

[101] R. K. Ursem. Multimodal optimization techniques in dynamic environments. In D. Whitley and et al., editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2000)*, pages 19–26. Morgan Kaufmann, 2000.

[102] A. S. Uyar and A. E. Harmanci. Investigation of new operators for a diploid genetic algorithm. In *Proceedings of SPIE's Annual Meeting*, 1999.

[103] A. S. Uyar and A. E. Harmanci. Preserving diversity in changing environments through diploidy with adaptive dominance. In W. B. Langdon and et al., editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002)*, page 679. Morgan Kaufmann, 2002.

[104] A. S. Uyar and A. E. Harmanci. A new population based adaptive dominance change mechanism for diploid genetic algorithms in dynamic environments. *Soft Computing*, 9(11):803–814, 2005.

[105] J. van Hemert, C. V. Hoyweghen, E. Lukshandl, and K. Verbeeck. A futurist approach to dynamic environments. In *GECCO EvoDOP Workshop*, pages 35–38, 2001.

[106] F. Vavak, T. C. Fogarty, and K. Jules. A genetic algorithm with variable range of local search for tracking changing environments. In H.-M. Voigt, editor, *Proceedings of the Parallel Problem Solving from Nature (PPSN IV)*, volume 1141 of *Lecture Notes in Computer Science*, pages 376–385. Springer, 1996.

[107] H. Wang and D. Wang. An improved primal-dual genetic algorithm for optimization in dynamic environments. In *Neural Information Processing*, volume 4234 of *Lecture Notes in Computer Science*, pages 836–844. Springer-Verlag, 2006.

[108] H. Wang, S. Yang, W. H. Ip, and D. Wang. Adaptive primal-dual genetic algorithms in dynamic environments. *IEEE Transactions on Systems, Man and Cybernetics*, 2009.

[109] K. Weicker. Performance measures for dynamic environments. In *Parallel Problem Solving from Nature (PPSN VII)*, volume 2439 of *Lecture Notes in Computer Science*, pages 64–73. Springer-Verlag, 2002.

[110] K. Weicker. *Evolutionary Algorithms and Dynamic Optimization Problems.* Der Andere Verlag, 2003.

[111] M. Wineberg and F. Oppacher. Enhancing the ga's ability to cope with dynamic environments. In D. Whitley, editor, *Proc. 2nd Genetic and Evolutionary Computation. Conference (GECCO 2000).* Morgan Kaufmann, 2000.

[112] S. Yang. Non-stationary problem optimization using the primal-dual genetic algorithm. In R. Sarker and et. al, editors, *Proceedings of the 2003 IEEE Congress on Evolutionary Computation (CEC 2003)*, volume 3, pages 2246–2253. IEEE Press, 2003.

[113] S. Yang. Constructing dynamic test environments for genetic algorithms based on problem difficulty. In *Proceedings of the 2004 IEEE Congress on Evolutionary Computation (CEC 2004)*, volume 2, pages 1262–1269. IEEE Press, 2004.

[114] S. Yang. Memory-based immigrants for genetic algorithms in dynamic environments. In H.-G. Beyer, editor, *Proceedings of the Seventh International Genetic and Evolutionary Computation Conference (GECCO 2005)*, volume 2, pages 1115–1122. ACM Press, 2005.

[115] S. Yang. Memory-enhanced univariate marginal distribution algorithms for dynamic optimization problems. In *Proceedings of the 2005 IEEE Congress on Evolutionary Computation (CEC 2005)*, volume 3, pages 2560–2567. IEEE Press, 2005.

[116] S. Yang. Population-based incremental learning with memory scheme for changing environments. In H.-G. Beyer, editor, *Proceedings of the Seventh International Genetic and Evolutionary Computation Conference (GECCO 2005)*, volume 1, pages 711–718. ACM Press, 2005.

[117] S. Yang. Associative memory scheme for genetic algorithms in dynamic environments. In F. Rothlauf and et al., editors, *Applications of Evolutionary Computing*, volume 3907 of *Lecture Notes in Computer Science*, pages 788–799. Springer-Verlag, 2006.

[118] S. Yang. A comparative study of immune system based genetic algorithms in dynamic environments. In M. Keijzer and et al., editors, *Proceedings of the Eighth International Genetic and Evolutionary Computation. Conference (GECCO 2006)*, pages 1377–1384. ACM Press, 2006.

[119] S. Yang. Dominance learning in diploid genetic algorithms for dynamic optimization problems. In M. Keijzer and et al., editors, *Proceedings of the Eighth International Genetic and Evolutionary Computation. Conference (GECCO 2006)*, pages 1435–1436. ACM Press, 2006.

[120] S. Yang. Adaptive business intelligence: Three case studies. In S. Yang, Y.-S. Ong, and Y. Jin, editors, *Evolutionary Computation in Dynamic and Uncertain Environments*, volume 51 of *Studies in Computational Intelligence*, pages 179–196. Springer-Verlag, 2007.

[121] S. Yang. Explicit memory schemes for evolutionary algorithms in dynamic environments. In S. Yang, Y.-S. Ong, and Y. Jin, editors, *Evolutionary Computation in Dynamic and Uncertain Environments*, volume 51 of *Studies in Computational Intelligence*, pages 3–28. Springer-Verlag, 2007.

[122] S. Yang. Genetic algorithms with elitism-based immigrants for changing optimization problems. In M. Giacobini and et al., editors, *Applications of Evolutionary Computing*, volume 4448 of *Lecture Notes in Computer Science*, pages 627–636. Springer-Verlag, 2007.

[123] S. Yang. Genetic algorithms with memory- and elitism-based immigrants in dynamic environments. *Evolutionary Computation*, 3(16):385–416, 2008.

[124] S. Yang and R. Tinós. A hybrid immigrants scheme for genetic algorithms in dynamic environments. *International Journal of Automation and Computing*, 3(4):243–254, 2007.

[125] S. Yang and X. Yao. Dual population-based incremental learning for problem optimization in dynamic environments. In M. Gen and et. al, editors, *Proceedings of the 7th Asia Pacific Symposium on Intelligent and Evolutionary Systems*, pages 49–56, 2003.

[126] S. Yang and X. Yao. Experimental study on population-based incremental learning algorithms for dynamic optimization problems. *Soft Computing*, 9(11):815–834, 2005.

[127] S. Yang and X. Yao. Population-based incremental learning with associative memory for dynamic environments. *IEEE Transactions on Evolutionary Computation*, 5(12):542–561, 2008.

[128] A. Younes, O. Basir, and P. Calamai. A hybrid evolutionary approach for combinatorial problems in dynamic environments. In *Proceedings of the Canadian Conference on Electrical and Computer Engineering (CCECE 2006)*, pages 1595–1600. IEEE Press, 2006.

[129] A. Zhou, Y. Jin, Q. Zhang, B. Sendhoff, and E. Tsang. Prediction-based population re-initialization for evolutionary dynamic multi-objective optimization. In *Evolutionary Multi-Criterion Optimization*, volume 4403 of *Lecture Notes in Computer Science*, pages 832–846. Springer, 2007.

Index