# Population-Based Incremental Learning with Memory Scheme for Changing Environments

Shengxiang Yang
Department of Computer Science, University of Leicester
University Road, Leicester LE1 7RH, United Kingdom
s.yang@mcs.le.ac.uk

## ABSTRACT

In recent years there has been a growing interest in studying evolutionary algorithms for dynamic optimization problems due to its importance in real world applications. Several approaches have been developed, such as the memory scheme. This paper investigates the application of the memory scheme for population-based incremental learning (PBIL) algorithms, a class of evolutionary algorithms, for dynamic optimization problems. A PBIL-specific memory scheme is proposed to improve its adaptability in dynamic environments. In this memory scheme the working probability vector is stored together with the best sample it creates in the memory and is used to reactivate old environments when change occurs. Experimental study based on a series of dynamic environments shows the efficiency of the memory scheme for PBILs in dynamic environments. In this paper, the relationship between the memory scheme and the multi-population scheme for PBILs in dynamic environments is also investigated. The experimental results indicate a negative interaction of the multi-population scheme on the memory scheme for PBILs in the dynamic test environments.

## Categories and Subject Descriptors

G.3 [**Mathematics of Computing**]: Probability and Statistics—*Probabilistic algorithms (including Monte Carlo)*

## General Terms

Algorithms, Experimentation, Performance

## Keywords

Population-based incremental learning, dynamic optimization problem, memory scheme, multi-population scheme

## 1. INTRODUCTION

Evolutionary algorithms (EAs), as a class of meta-heuristic algorithms inspired from principles of natural selection and population genetics, have been widely applied for solving stationary optimization problems. However, the environments of real world optimization problems are often dynamic, where the fitness function, design variables, and/or environmental conditions may change over time. This seriously challenges traditional EAs since they cannot adapt well to the changing environment once converged. In recent years, there is a growing interest in studying EAs for dynamic problems due to its importance in EA's real world applications since many real world problems are known to be dynamic [1]. Several approaches have been developed into EAs to address dynamic problems. They are categorized in [6] into four types: increasing diversity after a change, maintaining diversity throughout the run, memory-based schemes, and multi-population approaches.

In this paper, the memory scheme is investigated for the Population-Based Incremental Learning (PBIL) algorithm, which was first proposed by Baluja [2] as a class of EAs, for dynamic optimization problems. A PBIL-specific explicit memory scheme is proposed to improve its adaptability in dynamic environments. Within this memory scheme, the best sample created by the working probability vector together with the probability vector are stored in the memory in certain time and space pattern. When the environmental change is detected, the probability vector corresponding to the memory point that is re-evaluated as the best according to the new environment is retrieved to compete with the current working probability vector in PBIL for further iterations. Using the dynamic problem generator proposed in [17, 18], a series of dynamic test environments are constructed from two stationary functions and experimental study is carried out to compare the performance of investigated PBILs. The experimental study validates the efficiency of the memory scheme for PBILs for dynamic optimization problems. This paper also investigates the relationship between memory and multi-population schemes for PBILs in dynamic environments. The experimental results indicate interestingly that the multi-population scheme has a negative interaction on the memory scheme for PBILs in the dynamic test environments. The reason to this result is analysed.

The rest of this paper is organized as follows. The next section briefly reviews memory scheme for EAs in dynamic environments. Section 3 details the memory scheme proposed for PBIL and several PBIL algorithms investigated in this paper. Section 4 describes the dynamic test environment for this study. The experimental results and relevant analysis are presented in Section 5. Section 6 concludes this paper with discussions on relevant future work.


*GECCO'05*, June 25–29, 2005, Washington, DC, USA.

## 2. MEMORY SCHEMES FOR EAS IN DYNAMIC ENVIRONMENTS

The application of memory schemes has proved to be able to enhance EA's performance in dynamic environments, especially when the environment changes periodically. The basic principle of memory schemes is to store information, such as good solutions, from the current environment and reuse it later in new environments. As reviewed in [5], the information may be stored in two mechanisms: by implicit memory or by explicit memory.

For implicit memory schemes, EAs use genotype representations that contain redundant information to store good (partial) solutions to be reused later. Here, redundant representation acts as memory, which is implicit for the EA to use appropriately. Typical examples of implicit memory schemes are genetic algorithms based on diploidy or multiploidy representations. Goldberg and Smith [8] first extended the simple haploid GA to diploid GA with a triallelic dominance scheme. Thereafter, Ng and Wong [14] proposed a dominance scheme with four alleles for diploidy based GA. Lewis *et. al.* [10] further investigated an additive diploidy scheme where a gene becomes 1 if the addition of all alleles exceeds certain threshold and 0 otherwise. Similar to diploid GAs, Yang and Yao [18] proposed a dual PBIL for dynamic problems inspired by dualism in nature. In the dual PBIL, a dual probability vector is associated and compete with the main probability vector to generate samples. The dual PBIL has proved successful in dynamic environments with significant changes in genotypic space. In addition to multiploidy and dualism, a quite different implicit memory scheme was proposed in [7], which is haploid based but has a multi-levelled structure. In this representation, high level genes can regulate the activation of a set of low level genes. The set of low level genes can memorize good (partial) solutions in old environments that can be re-activated by high level genes in new environments.

While implicit memory schemes depend on redundant representations to store useful information for EAs to exploit in dynamic environments during the run, explicit memory schemes use precise representations but split an extra storage space where useful information from current generation can be explicitly stored and reused in later generations or environments. Explicit memory schemes mainly involve three concerns: what to store in the memory, how to organize and update the memory, and how to retrieve the memory.

For the first concern, a natural choice is to store good solutions and reuse them when the environment change is detected. We call this *direct memory scheme*. For example, Louis and Xu [11] studied the open shop re-scheduling problem. Whenever a change (in a known pattern) occurs, the GA is restarted from a population with partial (5-10%) individuals inherited from the old run while the rest randomly initialized. The authors reported significant improvements of their GA over the GA with totally random restart scheme. Instead of storing good solutions only, information that associates good solutions with their environments can also be stored with good solutions. This information can be used for similarity measure to associate a new environment with certain stored good solutions and then reuse these associated solutions more efficiently. We call this *associating memory scheme*. For example, Ramsey and Greffenstette [15] studied a GA for robot control problem, where good candidate solutions are stored in a permanent memory together with information about the robot current environment. When the robot incurs a new environment that is similar to a stored environment instance, the associated stored controller solution is re-activated. This scheme was reported to yield significant improvements.

The memory space or size is usually limited (and fixed) for computational and searching efficiency. This leads to the concern of memory organization and updating mechanisms. As to the memory organization, there exist two mechanisms: *local mechanism* where the memory is individual oriented and *global mechanism* where the memory is population oriented. Trojanowski and Michalewicz [16] introduced a local memory approach, where for each individual the memory stores a number of its ancestors. When the environment changes, the current individual and its ancestors are re-evaluated and compete together with the best becoming the active individual while the others stored in the memory. The global memory mechanism seems more natural and popular, see [5, 12]. In these global memory mechanisms best individual of the population is stored in the memory every certain generations while deleting one individual from the memory according to certain measure. As to the memory updating mechanism, a general principle is to select one memory individual to be removed for or updated by the best individual from the population in order to make the stored individuals to be of above average fitness, not too old, and distributed across several promissing areas of the search space [5]. Branke has discussed several memory replacement strategies, of which the most practical one is to replace the most similar memory individual if the new individual is better [5]. Bendtsen and Krink [4] proposed a dynamic memory updating scheme where the memory individual closest to the best population individual is, instead of being removed from the memory, moved toward the best population individual.

As to how to retrieve the memory, a natural idea is to retrieve the best memory individual(s) to replace the least fit individual(s) in the population. This can be done every generation or only when the environment changes. The memory retrieval is sort of coupled with the above two concerns. For example, for direct memory scheme the whole memory individuals may enter the new population as in [11] or compete with the population individuals for the new population as in [5], while for associated memory scheme only associated memory individual(s) may enter the new population [15].

In the following section, we will introduce an explicit memory scheme into PBIL for dynamic problems, which can be classified as an associating scheme.

## 3. MEMORY AND MULTI-POPULATION BASED PBILS

### 3.1 The Standard PBIL

The PBIL algorithm is a combination of evolutionary optimization and competitive learning. In fact, PBIL is an abstraction of the GA that explicitly maintains the statistics contained in GA's population [2, 3]. PBIL has proved to be very successful on numerous benchmark and real-world problems [9]. PBIL aims to generate a real-valued probability vector $\vec{P} = \{P[1], \ldots, P[l]\}$ ($l$ is the encoding length), which creates high quality solutions with high probability

**begin**
initialize probability vector $\vec{P}^0 := \vec{0.5}$
**repeat**
generate a set $S^t$ of $n$ samples by probability vector $\vec{P}^t$
evaluate samples in $S^t$ and denote the best sample $\vec{B}^t$
learn $\vec{P}^t$ toward $\vec{B}^t$ by Eq. (1)
**until** *terminated* = **true**
**end**

**Figure 1: Pseudocode for the standard PBIL with one probability vector (PBIL1).**

when sampled[1]. The pseudocode for standard PBIL, denoted *PBIL1*, is shown in Figure 1.

The standard PBIL starts from the *central probability vector* with each element set to 0.5. Sampling this initial probability vector creates random solutions since the probability of generating a 1 or 0 on each locus is equal. At iteration $t$, a set $S^t$ of $n$ solutions are sampled from the probability vector $\vec{P}^t$. The samples are evaluated using the problem-specific fitness function. Then the probability vector is learnt towards the best solution $\vec{B}^t$ of the set $S^t$ as follows.

$$P^{t+1}[i] := (1-\alpha) * P^t[i] + \alpha * B^t[i], \quad i = \{1, \ldots, l\} \quad (1)$$

where $\alpha$ is the learning rate, which determines the distance the probability vector is pushed for each iteration. After the probability vector is updated, a new set of samples is generated by the new probability vector and this cycle is repeated. As the search progresses, the elements in the probability vector move away from their initial settings of 0.5 towards either 0.0 or 1.0, representing high evaluation solutions. The search progress stops when some termination condition is satisfied, e.g., the maximum allowable number of iterations is reached or the probability vector is converged to either 0.0 or 1.0 for each bit position.

## 3.2 Memory-based PBIL

In this paper we propose a memory-based PBIL for dynamic optimization. The pseudocode for the memory-based PBIL with one probability vector, denoted *MPBIL1*, is shown in Figure 2, where $n$ is the number of evaluations per iteration including the memory points and $f(X)$ denotes the fitness of individual $X$.

Within MPBIL1, a memory of size $m = 0.1 * n$ is used to store samples and probability vectors. Each memory point consists of a pair: a smaple and an associated probability vector. The most similar measure, as discussed in [5], is used as the memory replacement strategy but in a varying time interval[2]. That is, when the memory is due to update, we first find the memory point with its sample closest to the best population sample. If the best population sample has higher fitness than this memory sample, it is replaced

[1]When sampling the probability vector for a solution, for each locus $i$ if a randomly created number $r = rand(0.0, 1.0) < P[i]$, it is set to 1; otherwise, it is set to 0.

[2]After each memory updating, a randomly created integer $R \in [5, 10]$ decides the next memory updating time. For example, suppose a memory updating happens at generation $t$, then the next memory updating time is at generation $t+R$.

**begin**
initialize prob. vector $\vec{P}^0 := \vec{0.5}$ and empty memory
**repeat**
generate a set $S^t$ of $n-m$ samples by $\vec{P}^t$
evaluate samples in $S^t$ and denote the best one $\vec{B}^t$
evaluate memory and denote the best memory sample $\vec{B}^t_M$ and its associated prob. vector $\vec{P}^t_M$
**if** *environmental change detected* **then**
**if** $f(\vec{B}^t_M) > f(\vec{B}^t)$ **then** replace $\vec{P}^t$ with $\vec{P}^t_M$
**else** learn $\vec{P}^t$ toward $\vec{B}^t$ by Eq. (1)
**if** *time to update memory* **then**
**if** *memory not full* **then**
store $\vec{B}^t$ and $\vec{P}^t$ into memory
**else**
find the memory sample $\vec{M}^t_C$ closest to $\vec{B}^t$ and its associated prob. vector $\vec{P}^t_C$
**if** $f(\vec{B}^t) > f(\vec{M}^t_C)$ **then**
replace $\vec{M}^t_C$ and $\vec{P}^t_C$ with $\vec{B}^t$ and $\vec{P}^t$ resp.
**until** *terminated* = **true**
**end**

**Figure 2: Pseudocode for the memory-based PBIL with one probability vector (MPBIL1).**

by the best population sample; otherwise, the memory stays unchanged. When a best population sample is stored in the memory, the working probability vector that generates the sample is also stored in the memory and is associated with the sample. Similarly, when replacing a memory point, both the sample and the associated probability vector within the memory point are replaced by the best population sample and the working probability vector respectively.

The memory is re-evaluated every iteration. If any memory sample has its fitness changed, the environment is detected to be changed. Then the memory probability vector associated with the best re-evaluated memory sample will replace the current working probability vector if its associated memory sample outperforms the best sample created by the working probability vector. If no environmental change is detected, MPBIL1 progresses as the standard PBIL.

The key idea behind MPBIL1 is to store good solutions as well as the associated environmental information in the memory. Here, the stored probability vector is taken as the representation of the environment when it is stored. MPBIL1 differs from Rasmey and Greffenstette's memory-based GA [15] in that for MPBIL1 the stored environment information, the probability vector, is used to directly re-activate an old environment it represents for MPBIL1, which may be similar to the newly changed problem environment. And the stored solutions, besides their role as environmental change detectors and memory replacement locators, are used to indicate which associated environment should be re-activated.

## 3.3 Multi-Population Based PBILs

In order to study the effect of multi-population on the memory scheme for PBIL in dynamic environments, PBILs

**begin**
initialize prob. vectors: $\vec{P}_1^0 := \vec{0.5}$, $\vec{P}_2^0 := \vec{rand}(0.0, 1.0)$
**if** *memory used* **then** initialize memory to be empty
**repeat**
generate a set $S_1^t$ of $n_1^t$ samples by $\vec{P}_1^t$
generate a set $S_2^t$ of $n_2^t$ samples by $\vec{P}_2^t$
evaluate samples in $S_1^t$ and denote the best one $\vec{B}_1^t$
evaluate samples in $S_2^t$ and denote the best one $\vec{B}_2^t$
adjust sample sizes $n_1^t$ and $n_2^t$ for $\vec{P}_1^t$ and $\vec{P}_2^t$
**if** *no memory used* **then** // for PBIL2
learn $\vec{P}_1^t$ and $\vec{P}_2^t$ toward $\vec{B}_1^t$ and $\vec{B}_2^t$ resp. by Eq. (1)
**else** // for MPBIL2
evaluate memory and denote the best memory sample $\vec{B}_M^t$ and its associated prob. vector $\vec{P}_M^t$
**if** *environmental change detected* **then**
denote $\vec{B}_w^t$ to be the worse of $\vec{B}_1^t$ and $\vec{B}_2^t$ and its associated prob. vector $\vec{P}_w^t$
**if** $f(\vec{B}_M^t) > f(\vec{B}_w^t)$ **then** replace $\vec{P}_w^t$ with $\vec{P}_M^t$
**else**
learn $\vec{P}_1^t$ and $\vec{P}_2^t$ toward $\vec{B}_1^t$ and $\vec{B}_2^t$ resp. by Eq. (1)
**if** *time to update memory* **then**
denote $\vec{B}_b^t$ to be the better of $\vec{B}_1^t$ and $\vec{B}_2^t$ and its associated prob. vector $\vec{P}_b^t$
**if** *memory not full* **then**
store $\vec{B}_b^t$ and $\vec{P}_b^t$ into memory
**else**
find the memory sample $\vec{M}_C^t$ closest to $\vec{B}_b^t$ and its associated prob. vector $\vec{P}_C^t$
**if** $f(\vec{B}_b^t) > f(\vec{M}_C^t)$ **then**
replace $\vec{M}_C^t$ and $\vec{P}_C^t$ with $\vec{B}_b^t$ and $\vec{P}_b^t$ resp.
**until** *terminated* = **true**
**end**

**Figure 3: Pseudocode for PBIL with two probability vectors and no memory (PBIL2) and memory-based PBIL with two probability vectors (MPBIL2).**

with two probability vectors are also investigated in this paper. The pseudocode for PBIL with two parallel probability vectors and no memory (denoted *PBIL2*) and memory-based PBIL with two parallel probability vectors (denoted *MPBIL2*) is shown in Figure 3.

In PBIL2 and MPBIL2, the two probability vectors work in parallel. Each one is sampled independently and is learnt toward the best solution generated by itself. The probability vector $\vec{P}_1$ is initialized to the central probability vector while $\vec{P}_2$ is randomly initialized. The sample sizes for $\vec{P}_1$ and $\vec{P}_2$ are equally initialized to $0.5 * n$ for PBIL2 and $0.45 * n$ for MPBIL2 and are slightly adjusted within the range of $[n_{min}, n_{max}] = [0.2*n, 0.8*n]$ for PBIL2 and $[n_{min}, n_{max}] = [0.2 * n, 0.7 * n]$ for MPBIL2 according to their performance. The winner probability vector gets $\Delta = 0.05 * n$ for its sample size from the loser; if the two probability vectors tie, their sample sizes do not change. For MPBIL2, the memory size is fixed to $0.1 * n$. When an environmental change is detected, the best memory probability vector will compete with the loser working probability vector. The memory updating mechanism for MPBIL2 is similar as for MPBIL1 except that the winner working probability vector and its best sample will be stored if suitable.

# 4. DYNAMIC TEST ENVIRONMENTS

The dynamic problem generator proposed in [17, 18] can construct dynamic environments from any binary-encoded stationary function $f(\vec{x})$ ($\vec{x} \in \{0,1\}^l$) by a bitwise exclusive-or (XOR) operator. Suppose the environment is changed every $\tau$ generations. For each environmental period $k$, an XORing mask $\vec{M}(k)$ is incrementally generated as follows:

$$\vec{M}(k) = \vec{M}(k-1) \oplus \vec{T}(k) \tag{2}$$

where "$\oplus$" is the XOR operator (i.e., $1 \oplus 1 = 0$, $1 \oplus 0 = 1$, $0 \oplus 0 = 0$) and $\vec{T}(k)$ is an intermediate binary template randomly created with $\rho \times l$ ones for environmental period $k$. For the first period $k = 1$, $\vec{M}(1)$ is set to a zero vector. Then, the population at generation $t$ is evaluated as below:

$$f(\vec{x}, t) = f(\vec{x} \oplus \vec{M}(k)) \tag{3}$$

where $k = \lceil t/\tau \rceil$ is the environmental period index.

With this generator, the environmental dynamics can be easily tuned by two parameters. The parameter $\tau$ controls the change speed while $\rho \in (0.0, 1.0)$ controls the severity each time the environment changes. Bigger value of $\rho$ means severer environmental change and greater challenge to EAs.

In this paper, in order to compare PBILs in dynamic environments, two stationary functions are selected. The first function is a 120-bit $OneMax$ function, which aims to maximize the number of ones in a binary chromosome. The second function consists of 30 contiguous order-4 building blocks, called $NK(30, 4)$. Each building block contributes a value of 4 to the total fitness if all its four bits equal to one, otherwise it contributes 0. The fitness of a bit string is the sum of contributions from all building blocks. The optimum fitness of $NK(30, 4)$ is 120.

Dynamic test environments are constructed from the two stationary functions using above dynamic problem generator. The landscape is periodically changed every $\tau$ generations during the run of PBILs and $\tau$ is set to 10, 50 and 100 respectively. The environmental change severity parameter $\rho$ is set to 0.1, 0.2, 0.4, 0.6, and 0.9 respectively. And in order to study the behaviour of PBILs in randomly changing environment, $\rho$ is also set to a random number uniformly distributed in $[0.01, 0.99]$ (i.e., $\rho = rand(0.01, 0.99)$) each time the environment changes during the run.

Totally, a series of 18 dynamic problems, 3 values of $\tau$ combined with 6 values of $\rho$, are constructed from each stationary function.

# 5. EXPERIMENTAL STUDY

## 5.1 Experimental Design

Experiments were carried out to compare different PBILs on the test environments constructed above. For all PBILs, the total sample size $n$ (including memory samples if used) is set to 120, the memory size $m = 0.1 * n = 10$ if used, and the learning rate for all probability vectors is set to 0.05.

For each experiment of an algorithm on a dynamic test problem, 20 independent runs were executed with the same

**Table 1: Experimental results with respect to overall performance of PBILs.**

| Performance | | $OneMax$ | | | | $NK(30,4)$ | | | |
|---|---|---|---|---|---|---|---|---|---|
| $\tau$ | $\rho$ | *PBIL*1 | *PBIL*2 | *MPBIL*1 | *MPBIL*2 | *PBIL*1 | *PBIL*2 | *MPBIL*1 | *MPBIL*2 |
| 10 | 0.1 | 72.8 | 74.2 | 83.1 | 82.2 | 22.5 | 23.9 | 33.0 | 32.1 |
| 10 | 0.2 | 71.8 | 72.8 | 78.7 | 78.1 | 21.0 | 22.2 | 28.3 | 27.9 |
| 10 | 0.4 | 71.2 | 71.7 | 75.7 | 75.5 | 20.4 | 21.4 | 25.3 | 25.0 |
| 10 | 0.6 | 70.6 | 71.4 | 74.5 | 74.3 | 20.3 | 21.1 | 24.4 | 24.3 |
| 10 | 0.9 | 70.1 | 70.8 | 77.2 | 75.7 | 20.2 | 21.4 | 25.8 | 25.6 |
| 10 | *rand* | 70.8 | 71.7 | 76.1 | 75.9 | 20.3 | 21.3 | 25.7 | 25.3 |
| 50 | 0.1 | 66.2 | 66.8 | 85.3 | 84.5 | 14.9 | 17.7 | 44.7 | 40.7 |
| 50 | 0.2 | 65.2 | 65.5 | 84.6 | 83.6 | 12.9 | 15.4 | 38.3 | 36.3 |
| 50 | 0.4 | 64.5 | 65.7 | 81.6 | 80.1 | 12.2 | 14.4 | 34.1 | 30.4 |
| 50 | 0.6 | 64.9 | 66.5 | 82.2 | 79.8 | 11.7 | 14.2 | 31.9 | 29.1 |
| 50 | 0.9 | 65.7 | 68.5 | 86.0 | 85.2 | 12.9 | 15.8 | 36.5 | 30.7 |
| 50 | *rand* | 64.2 | 66.5 | 84.6 | 82.7 | 12.0 | 14.4 | 35.5 | 32.0 |
| 100 | 0.1 | 64.4 | 64.9 | 85.7 | 85.5 | 12.9 | 14.9 | 44.6 | 40.1 |
| 100 | 0.2 | 63.0 | 62.9 | 84.9 | 84.4 | 11.4 | 13.0 | 41.2 | 36.9 |
| 100 | 0.4 | 61.6 | 62.4 | 83.5 | 80.9 | 9.7 | 12.5 | 37.3 | 33.6 |
| 100 | 0.6 | 61.6 | 62.7 | 85.3 | 81.1 | 9.6 | 11.9 | 36.1 | 32.2 |
| 100 | 0.9 | 62.7 | 65.0 | 86.6 | 85.3 | 10.6 | 13.6 | 42.7 | 35.2 |
| 100 | *rand* | 61.9 | 62.9 | 86.2 | 84.1 | 10.2 | 12.7 | 40.4 | 34.8 |

**Table 2: Statistical results of comparing PBIL algorithms on dynamic problems.**

| $t$-test Result | $OneMax$ | | | | | | $NK(30,4)$ | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\tau = 10$, $\rho \Rightarrow$ | 0.1 | 0.2 | 0.4 | 0.6 | 0.9 | *rand* | 0.1 | 0.2 | 0.4 | 0.6 | 0.9 | *rand* |
| *PBIL*2 − *PBIL*1 | + | + | + | + | + | + | + | + | + | + | + | + |
| *MPBIL*1 − *PBIL*1 | + | + | + | + | + | + | + | + | + | + | + | + |
| *MPBIL*1 − *PBIL*2 | + | + | + | + | + | + | + | + | + | + | + | + |
| *MPBIL*2 − *PBIL*2 | + | + | + | + | + | + | + | + | + | + | + | + |
| *MPBIL*2 − *MPBIL*1 | − | − | ∼ | − | − | ∼ | − | − | − | ∼ | ∼ | − |
| $\tau = 50$, $\rho \Rightarrow$ | 0.1 | 0.2 | 0.4 | 0.6 | 0.9 | *rand* | 0.1 | 0.2 | 0.4 | 0.6 | 0.9 | *rand* |
| *PBIL*2 − *PBIL*1 | ∼ | ∼ | + | + | + | + | + | + | + | + | + | + |
| *MPBIL*1 − *PBIL*1 | + | + | + | + | + | + | + | + | + | + | + | + |
| *MPBIL*1 − *PBIL*2 | + | + | + | + | + | + | + | + | + | + | + | + |
| *MPBIL*2 − *PBIL*2 | + | + | + | + | + | + | + | + | + | + | + | + |
| *MPBIL*2 − *MPBIL*1 | ∼ | − | − | − | − | − | − | − | − | − | − | − |
| $\tau = 100$, $\rho \Rightarrow$ | 0.1 | 0.2 | 0.4 | 0.6 | 0.9 | *rand* | 0.1 | 0.2 | 0.4 | 0.6 | 0.9 | *rand* |
| *PBIL*2 − *PBIL*1 | ∼ | ∼ | + | + | + | + | + | + | + | + | + | + |
| *MPBIL*1 − *PBIL*1 | + | + | + | + | + | + | + | + | + | + | + | + |
| *MPBIL*1 − *PBIL*2 | + | + | + | + | + | + | + | + | + | + | + | + |
| *MPBIL*2 − *PBIL*2 | + | + | + | + | + | + | + | + | + | + | + | + |
| *MPBIL*2 − *MPBIL*1 | ∼ | ∼ | − | − | − | − | − | − | − | − | − | − |

set of random seeds. And for each run 100 environmental changes were allowed and the best-of-generation fitness was recorded every generation. The overall performance of an algorithm on a problem is formulated as below:

$$\overline{F}_{BOG} = \frac{1}{G}\sum_{i=1}^{G}(\frac{1}{N}\sum_{j=1}^{N}F_{BOG_{ij}}) \quad (4)$$

where $G$ is the total number of generations for a run (i.e., $G = 100 * \tau$), $N = 20$ is the total number of runs, $F_{BOG_{ij}}$ is the best-of-generation fitness of generation $i$ of run $j$, and $\overline{F}_{BOG}$ is the offline performance, i.e., the best-of-generation fitness averaged across the 20 runs and then averaged over the data gathering period.

## 5.2 Experimental Results and Analysis

The experimental results of algorithms on dynamic problems are presented in Table 1. The statistical results of comparing algorithms by one-tailed $t$-test with 38 degrees of freedom at a 0.05 level of significance are given in Table 2. In Table 2, the $t$-test result regarding Alg. 1 − Alg. 2 is shown as "+", "−", or "∼" when Alg. 1 is significantly better than, significantly worse than, or statistically equivalent to Alg. 2 respectively. The results are also plotted in Figure 4 and Figure 5 for dynamic $OneMax$ and $NK(30,4)$ functions respectively. From the tables and figures several results can be observed.

First, a prominent result is that the memory-based PBILs are significantly better than PBILs without memory on all dynamic test problems. This result validated the efficiency

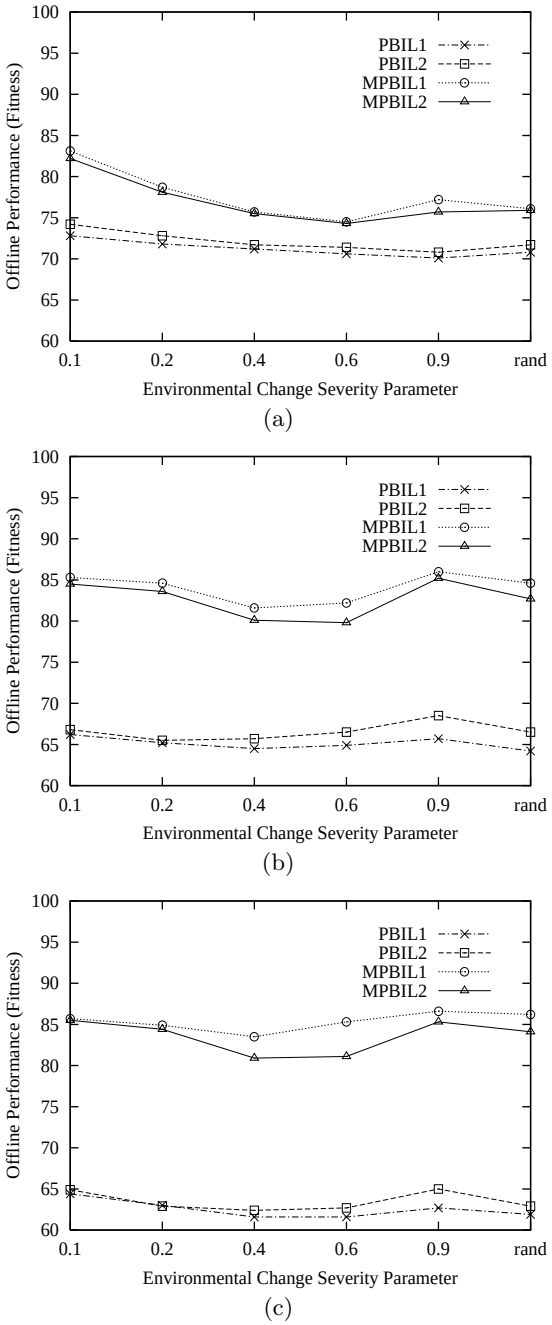


**Figure 4: Experimental results of PBILs on dynamic $OneMax$ functions with different $\rho$ and (a) $\tau = 10$, (b) $\tau = 50$, and (c) $\tau = 100$.**

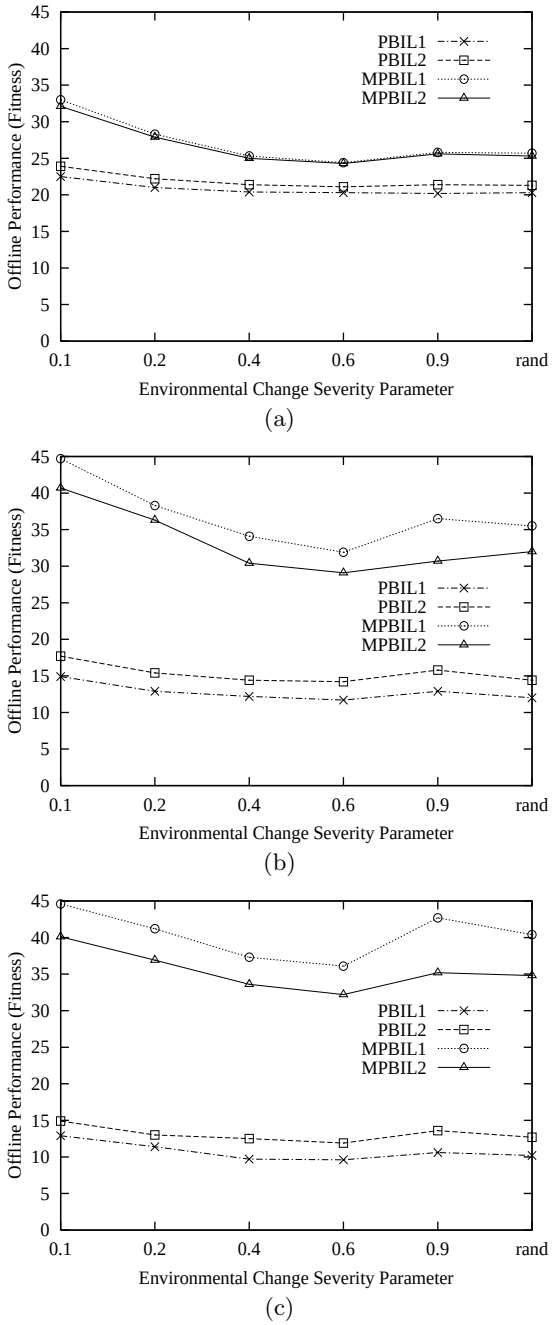


**Figure 5: Experimental results of PBILs on dynamic $NK(30, 4)$ functions with different $\rho$ and (a) $\tau = 10$, (b) $\tau = 50$, and (c) $\tau = 100$.**

of introducing the memory scheme into PBILs. In order to better understand the effect of the memory scheme on PBIL's performance in dynamic environments, the dynamic behaviour of algorithms with respect to best-of-generation fitness against generations on the dynamic problems with $\tau = 50$ and $\rho = 0.1$ is plotted in Figure 6, where the data were averaged over 20 runs. From Figure 6, it can be seen that the performance of PBIL1 and PBIL2, after rising for a few first environmental changes, drops with the generation consistently till a stable low fit state. On the contrast, for MPBIL1 and MPBIL2, their performance rises for a few first environmental changes and then drops for subsequent several environmental changes. However, when the memory is sort of stable, it starts to take effect and adapt the PBILs to the environmental changes. And hence, their performance stays at certain level instead of dropping consistently.

Second, an interesting observation is that for each fixed value of $\rho$, when the value of $\tau$ increases, the performance of

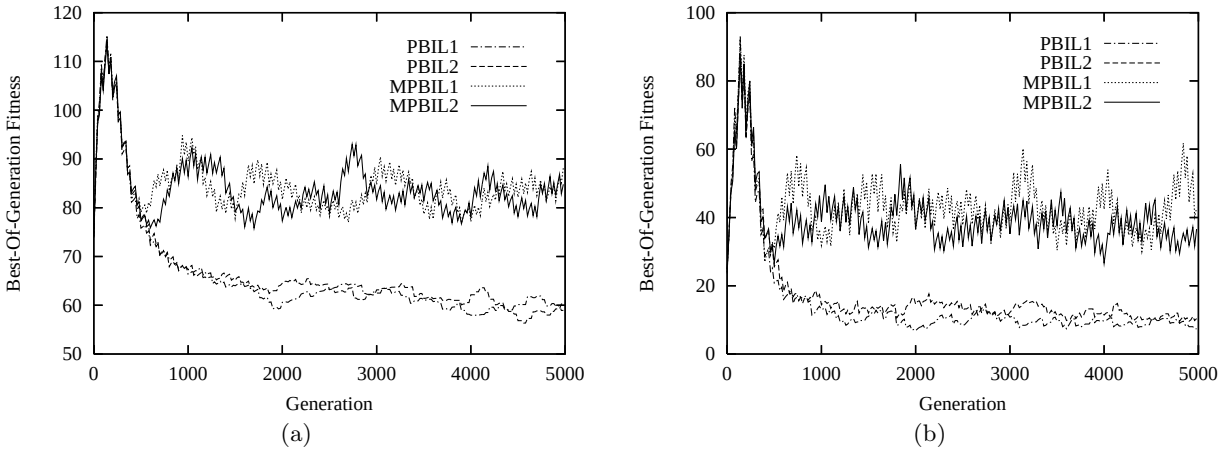


**Figure 6: Dynamic behaviour of PBILs on dynamic problems: (a) $OneMax$ and (b) $NK(30,4)$ with $\tau = 50$ and $\rho = 0.1$. The data were averaged over 20 runs.**

**Table 3: Experimental results relevant to MPBIL2a on dynamic problems.**

| Performance | $OneMax$ | | | | | | $NK(30,4)$ | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\rho \Rightarrow$ | 0.1 | 0.2 | 0.4 | 0.6 | 0.9 | *rand* | 0.1 | 0.2 | 0.4 | 0.6 | 0.9 | *rand* |
| $\tau = 10$ | 82.3 | 78.1 | 75.7 | 74.3 | 76.3 | 75.9 | 32.3 | 27.9 | 25.1 | 24.3 | 25.8 | 25.4 |
| $\tau = 50$ | 84.9 | 84.1 | 81.2 | 81.0 | 90.3 | 85.1 | 41.2 | 36.2 | 31.4 | 30.1 | 32.0 | 32.6 |
| $\tau = 100$ | 85.4 | 84.2 | 83.5 | 83.1 | 94.7 | 86.2 | 40.8 | 38.4 | 34.0 | 32.2 | 38.9 | 36.4 |
| $t$-test Result | $OneMax$ | | | | | | $NK(30,4)$ | | | | | |
| $\tau = 10$, $\rho \Rightarrow$ | 0.1 | 0.2 | 0.4 | 0.6 | 0.9 | *rand* | 0.1 | 0.2 | 0.4 | 0.6 | 0.9 | *rand* |
| *MPBIL2a* − *MPBIL1* | − | − | ∼ | − | − | ∼ | − | − | ∼ | ∼ | ∼ | ∼ |
| *MPBIL2a* − *MPBIL2* | ∼ | ∼ | ∼ | ∼ | + | ∼ | ∼ | ∼ | ∼ | ∼ | ∼ | ∼ |
| $\tau = 50$, $\rho \Rightarrow$ | 0.1 | 0.2 | 0.4 | 0.6 | 0.9 | *rand* | 0.1 | 0.2 | 0.4 | 0.6 | 0.9 | *rand* |
| *MPBIL2a* − *MPBIL1* | ∼ | ∼ | ∼ | − | + | ∼ | − | − | − | − | − | − |
| *MPBIL2a* − *MPBIL2* | ∼ | ∼ | + | + | + | + | ∼ | ∼ | + | + | + | ∼ |
| $\tau = 100$, $\rho \Rightarrow$ | 0.1 | 0.2 | 0.4 | 0.6 | 0.9 | *rand* | 0.1 | 0.2 | 0.4 | 0.6 | 0.9 | *rand* |
| *MPBIL2a* − *MPBIL1* | ∼ | ∼ | ∼ | − | + | ∼ | − | − | − | − | − | − |
| *MPBIL2a* − *MPBIL2* | ∼ | ∼ | + | + | + | + | ∼ | ∼ | ∼ | ∼ | + | ∼ |

PBILs without memory (PBIL1 and PBIL2) decreases while the performance of PBILs with memory (MPBIL1 and MPBIL2) increases. This is because when $\tau$ is bigger, the environment changes slower and PBILs get more time to search and hence to converge before next change. And this leads to less adaptability for PBILs without memory when the environment changes and hence their worse performance. However, for PBILs with memory, long search time, i.e., bigger $\tau$, makes the stored memory probability vectors more precisely represent the old environments when they are stored. This leads to better efficiency of the memory scheme and hence better performance of PBILs with memory.

Third, for a fixed $\tau$, when the value of $\rho$ increases (i.e., the severity of environment change increases) the performance of PBILs decreases. But when $\rho = 0.9$ PBILs perform better than when $\rho = 0.4$ or $\rho = 0.6$. The reason is similar as explained in [18]. When the environment randomly changes with respect to the severity, i.e., $\rho = rand(0.01, 0.99)$, the performance of PBILs is similar as when $\rho$ is set to medium values, e.g., 0.4 or 0.6. This fits well with the fact that the expected value of $rand(0.01, 0.99)$ is about 0.5.

Finally, another interesting result is that PBIL2 outperforms PBIL1 while MPBIL2 is beaten by MPBIL1 on most dynamic problems. In other words, multi-population is beneficial for PBIL without memory while negative for PBIL with memory. For PBIL without memory introducing an extra probability vector increases the diversity and hence helps the adaptability in dynamic environments. However, for PBIL with memory, when the environment changes, the existence of an extra probability vector, which is the better working probability vector when change occurs, affects the performance of the newly re-activated memory probability vector. This negative effect outweighs the possible positive effect due to the diversity introduced by the extra probability vector.

In order to further validate the negative effect of multi-population on memory scheme for PBIL in dynamic environments, we modify MPBIL2 in the memory retrieval strategy, denoted *MPBIL2a*, where the best two memory probability vectors compete with both the two working probability vectors. The experimental results of MPBIL2a and relevant statistical test results are presented in Table 3.

From Table 3, it can be seen that though the performance of MPBIL2a is significantly improved over MPBIL2 on several dynamic problems, it is still beaten by MPBIL1 on most dynamic problems.

## 6. CONCLUSIONS AND FUTURE WORK

In this paper, the memory scheme is introduced into the PBIL algorithm to enhance its performance in dynamic environments. Within this memory scheme, the best sample and the working probability vector that creates it are stored in the memory by replacing the most similar memory point. When the environmental change is detected, the probability vector corresponding to the memory point, which is re-evaluated as the best according to the new environment, is retrieved to compete with the current working probability vector for further iterations. The experimental results based on a series of dynamic environments validate the efficiency of the memory scheme for PBILs in dynamic environments.

This paper also investigates the relationship between memory and multi-population schemes for PBILs in dynamic environments. An interesting result indicated by the experimental results is that the multi-population scheme improves the performance of PBIL when no memory is used. However, when memory is used for PBIL, multi-population has a negative effect on PBIL's performance in the tested dynamic environments. This happens because through occupying the resource of the sample size, the existence of an extra probability vector slows down the seaching progress (and hence affects the efficiency) of the memory probability vector that is just re-activated when the environment changes.

The work studied in this paper can be extended in several ways. In this paper, the dynamic environments tested are not cyclical. It would be interesting to investigate the effect of the memory scheme for PBILs under cyclically changing environments, where an old environment will return exactly after certain fixed number of changes. Under such dynamic environments the memory scheme should be expected to be more beneficial. Developing other memory management and retrieval mechanisms would be another interesting future work for memory-based PBILs and other estimation of distribution algorithms [9, 13] in dynamic environments. And it is also an interesting work to further investigate the interactions between the memory scheme and other apporaches, such as random immigrants and mutation, for PBILs in dynamic environments.

## 7. REFERENCES


[1] T. Bäck. On the behaviour of evolutionary algorithms in dynamic fitness landscape. In *Proc. of the 1998 IEEE Int. Conf. on Evolutionary Computation*, pages 446–451, 1998.

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

[3] S. Baluja and R. Caruana. Removing the genetics from the standard genetic algorithm. In *Proc. of the 12th Int. Conf. on Machine Learning*, pages 38–46, 1995.

[4] C. N. Bendtsen and T. Krink. Dynamic memory model for non-stationary optimization. In *Proc. of the 2002 Congress on Evolutionary Computation*, pages 145–150, 2002.

[5] J. Branke. Memory enhanced evolutionary algorithms for changing optimization problems. In *Proc. of the 1999 Congress on Evolutionary Computation*, volume 3, pages 1875–1882, 1999.

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

[7] D. Dasgupta and D. McGregor. Nonstationary function optimization using the structured genetic algorithm. In *Proc. of the 2nd Int. Conf. on Parallel Problem Solving from Nature*, pages 145–154, 1992.

[8] D. E. Goldberg and R. E. Smith. Nonstationary function optimization using genetic algorithms with dominance and diploidy. In *Proc. of the 2nd Int. Conf. on Genetic Algorithms*, pages 59–68. Lawrence Erlbaum Associates, 1987.

[9] P. Larrañaga and J. A. Lozano. *Estimation of Distribution Algorithms: A New Tool for Evolutionary Computation*. Kluwer Academic Publishers, 2002.

[10] E. H. J. Lewis and G. Ritchie. A comparison of dominance mechanisms and simple mutation on non-stationary problems. In *Proc. of the 5th Int. Conf. on Parallel Problem Solving from Nature*, pages 139–148, 1998.

[11] S. J. Louis and Z. Xu. Genetic algorithms for open shop scheduling and re-scheduling. In *Proc. of the 11th ISCA Int. Conf. on Computers and their Applications*, pages 99–102, 1996.

[12] N. Mori, H. Kita and Y. Nishikawa. Adaptation to changing environments by means of the memory based thermodynamical genetic algorithm. In *Proc. of the 7th Int. Conf. on Genetic Algorithms*, pages 299–306. Morgan Kaufmann Publishers, 1997.

[13] H. Mühlenbein and G. Paaß. From recombination of genes to the estimation of distributions I. Binary parameters. In H.-M. Voigt, W. Ebeling, I. Rechenberg, and H.-P. Schwefel, editors, *Proc. of the 4th Int. Conf. on Parallel Problem Solving from Nature*, pages 178–187, 1996.

[14] K. P. Ng and K. C. Wong. A new diploid scheme and dominance change mechanism for non-stationary function optimisation. In *Proc. of the 6th Int. Conf. on Genetic Algorithms*. Morgan Kaufmann Publishers, 1997.

[15] C. L. Ramsey and J. J. Greffenstette. Case-based initializtion of genetic algorithms. In *Proc. of the 5th Int. Conf. on Genetic Algorithms*. Morgan Kaufmann Publishers, 1993.

[16] K. Trojanowski and Z. Michalewicz. Searching for optima in non-stationary environments. In *Proc. of the 1999 Congress on Evolutionary Computation*, pages 1843–1850, 1999.

[17] S. Yang. Non-stationary problem optimization using the primal-dual genetic algorithm. In *Proc. of the 2003 Congress on Evolutionary Computation*, volume 3, pages 2246–2253, 2003.

[18] S. Yang and X. Yao. Experimental study on population-based incremental learning algorithms for dynamic optimization problems. *Soft Computing*, 2005, to be published.