



# A novel multi-swarm algorithm for optimization in dynamic environments based on particle swarm optimization

Q1 Danial Yazdani <sup>a,\*</sup>, Babak Nasiri <sup>b</sup>, Alireza Sepas-Moghaddam <sup>b</sup>, Mohammad Reza Meybodi <sup>c,d</sup>

<sup>a</sup> Young Researchers Club, Mashhad Branch, Islamic Azad University, Mashhad, Iran

<sup>b</sup> Department of Computer Engineering and Information Technology, Islamic Azad University, Qazvin Branch, Qazvin, Iran

<sup>c</sup> Department of Computer Engineering and Information Technology, Amirkabir University of Technology, Tehran, Iran

<sup>d</sup> Institute for Studies in Theoretical Physics and Mathematics (IPM), School of Computer Science, Tehran, Iran

## ARTICLE INFO

### Article history:

Received 17 May 2012

Received in revised form 5 September 2012

Accepted 20 December 2012

Available online xxx

### Keywords:

Particle swarm optimization

Dynamic environments

Swarm intelligence

Moving Peak Benchmark

Multi-swarm

## ABSTRACT

Optimization in dynamic environment is considered among prominent optimization problems. There are particular challenges for optimization in dynamic environments, so that the designed algorithms must conquer the challenges in order to perform an efficient optimization. In this paper, a novel optimization algorithm in dynamic environments was proposed based on particle swarm optimization approach, in which several mechanisms were employed to face the challenges in this domain. In this algorithm, an improved multi-swarm approach has been used for finding peaks in the problem space and tracking them after an environment change in an appropriate time. Moreover, a novel method based on change in velocity vector and particle positions was proposed to increase the diversity of swarms. For improving the efficiency of the algorithm, a local search based on adaptive exploiter particle around the best found position as well as a novel *awakening–sleeping* mechanism were utilized. The experiments were conducted on Moving Peak Benchmark which is the most well-known benchmark in this domain and results have been compared with those of the state-of-the art methods. The results show the superiority of the proposed method.

© 2013 Published by Elsevier B.V.

## 1. Introduction

Optimization is considered among the most important problems in mathematics and sciences. The importance of optimization and its numerous applications has inspired the scientists to investigate on different aspects of it. Optimization problems could be seen in real-world applications, e.g. itinerary selection. The goal in all optimization problems is to maximize or minimize one or more cost functions in a problem considering its limitations. While there are a limited number of limitations in a problem space, it can be solved easily. However, increasing limitations leads to an NP-hard problem which needs a high computational cost to be solved. Therefore, researchers are continually seeking the efficient ways for solving such NP-hard problems. Meta-heuristic methods are among these techniques.

Meta-heuristic methods present a computing method for solving optimization problems in which an iterative process for enhancing the obtained solution is utilized until a terminating state is reached. Until now, most existing meta-heuristic methods

have focused on static problems. In such problems, the problem space remains unchanged during the optimization process. However, most optimization problems in real world are dynamic and non-deterministic, i.e. the problem search space changes during the optimization process. For example, scheduling tasks is a problem usually solved as a static optimization problem. However, by arriving of a new task during the scheduling procedure, or occurrence of some other problems such as failures in resources, the search environment is changed from a static problem into a dynamic one. As a result, the previous static solutions may no longer be applicable on the new environment. Such problems are called dynamic state optimization problems.

In static optimization problems, finding a global optimum is considered as the main goal. On the other hand, finding a global optimum is not the only goal in dynamic environments and tracking the optimum in the problem space is extremely important in this domain. In fact, the proposed methods for optimization in static environments fail to appropriately follow the optimum. Thus, such methods are not suitable to be used in dynamic environments and the necessity of finding different techniques involving different goal functions and different evaluation criteria for optimization in dynamic environments is obvious.

In this paper, a new optimization method based on PSO has been proposed, by presenting a set of consistency techniques with the problem space for optimization in dynamic environments. To

\* Corresponding author. Tel.: +98 935 1185556.

E-mail addresses: [danial.yazdani@yahoo.com](mailto:danial.yazdani@yahoo.com), [d.yazdani@mshdiau.ac.ir](mailto:d.yazdani@mshdiau.ac.ir) (D. Yazdani), [nasiri.babak@qiau.ac.ir](mailto:nasiri.babak@qiau.ac.ir) (B. Nasiri), [\(A. Sepas-Moghaddam\)](mailto:sepasmoghaddam@qiau.ac.ir), [\(M.R. Meybodi\)](mailto:mmeybodi@aut.ac.ir).

evaluate the proposed method, Moving Peak Benchmark (MPB) has been used, which is the best-known benchmark for evaluating optimization methods in dynamic environments.

The rest of the paper is organized as follows. Section 2 reviews the previous literature on the subject. Section 3 explains the proposed method for solving optimization problems in dynamic environments. Section 4 is dedicated to the experiments and the obtained results. The last section concludes the paper and presents the scope of the future works.

## 2. Related work

Using meta-heuristic methods for optimization in dynamic environments has its own challenges which do not exist in static environments. The most important challenges encountered by meta-heuristic methods in dynamic environments are *outdated memory* and *diversity loss*. The outdated memory challenge exists in optimization in dynamic environments, because when the environment changes, the fitness value of the obtained solutions will change and will no longer correspond to the stored value in the memory used by these methods.

Diversity loss occurs because of the intrinsic nature of meta-heuristic methods for convergence. The reason for this challenge is the inherent characteristic of these algorithms for converging to the previous optimum position and the exorbitant proximity of the solutions to each other. The easiest way to solve these two issues is re-initialization [1]. In re-initialization, we look at the changed environment as a new problem and re-start the optimization method using the changed environment. However, due to this fact that the efficiency of the optimization process in the changed environment could be improved using the knowledge acquired from the previous environment, the re-initialization methods imply the loss of all the knowledge obtained so far from the problem space.

In the following, the presented solutions for resolving these two main challenges are studied in details. The first challenge, i.e. outdated memory, is of less importance compared to the other one, and two solutions have been proposed for it, which are forgetting memory and re-evaluating memory [2]. These two solutions are also used for optimization methods which use the memory for storing information obtained from the problem space. In forgetting memory method, the position stored for each solution will be replaced by its position in the new environment. In re-evaluating memory method, the stored positions in the memory are re-evaluated.

Several solutions have been proposed for the second challenge, i.e. diversity loss. The solutions have been classified into two main categories: presenting diversity methods and diversity maintenance methods.

### 2.1. Presenting diversity methods

In this category, the algorithms allow the diversity loss to occur, and afterward they try to solve it. They are divided into two sub-categories as well:

#### 2.1.1. Mutation and self-adaptation

In this subcategory, the algorithms try to generate the lost diversity in the environment by performing mutation and self-adaptation. In [3], an adaptive mutation operator called Triggered Hyper-Mutation was proposed as a coefficient which was multiplied by the normal mutation. In [4], a chaotic mutation is used adaptively to create diversity in the environment. Another method is proposed in [5], introducing a variable local search which solves the problem of constancy in mutation step size in [3] by making it adaptive. Replacing the random solutions by the previous suitable solutions after the environment change is another strategy suggested in [6] for producing diversity. In [7] a variable relocation

method has been proposed which relocates the solutions according to the amount of change in the fitness function value at the time of the environment change, performing such relocations for each solution with different radiiuses. Also, in [55] a hyper-reflection scheme is presented for optimization in dynamic environments.

#### 2.1.2. Other approaches

There are other methods proposed for creating diversity in the environment after discovering a change in the environment. In [8], a method called RPSO has been proposed for randomizing some or all of the solutions after detecting changes in the environment. In [9], another algorithm called PBIL (Population-Based Incremental Learning) is presented which uses a flexible probability vector for generating solutions. In [56] a hybrid approach for dynamic continuous optimization problems is presented and in [57] a cooperative meta-heuristic approach configured via fuzzy logic and SVMs.

### 2.2. Diversity maintenance methods

In this method, it is tried to keep diversity in the environment at all times (before and after the change). The proposed algorithms in this category are divided into three subcategories:

#### 2.2.1. Dynamic topology

Algorithms in this subcategory try to decrease the rate of algorithm convergence into a global optimum by limiting the existing communications among solutions, and thus keep the diversity in the environment. In [10], a method called FGPSO with a proximity structure similar to a grid is proposed for maintaining diversity. In [11], HPSO has been proposed for preserving diversity, with its hierarchical and pseudo-tree structure. Another method, called Cellular PSO [12] has been suggested for optimization in dynamic environments, and this method utilizes the local information exchange and cellular automata distribution features. In [13], the presented model in [12] has been improved by changing the roles of some particles to quantum particles just after the environment change.

#### 2.2.2. Memory-based methods

When the environment changes are periodical or recurrent, it will be very useful to store previous optimal solutions for using in the future. Memory-based methods try to store such information. These methods have been usually suggested for evolutionary methods such as GA and EDA which are genetic-based. Memory-based methods themselves can be divided into two main subclasses of implicit memory and explicit memory.

**2.2.2.1. Implicit memory.** In this subcategory of memory-based methods, memory is integrated with meta-heuristic methods as a redundant representation. Redundant representation using diploid genomes is the most widely used method in this subclass [14,15]. In diploid genome, each solution consists of two alleles in every locus. In [14,15], two methods based on diploid genetic algorithms have been proposed for optimization in dynamic environments using implicit memory methods.

**2.2.2.2. Explicit memory.** In this subclass, memory is created explicitly. This subclass has its own two subclasses as follows:

(a) **Direct memory:** most of the times, direct memories include the previous suitable solutions [16], but in some cases solutions with the most diversity are also kept in memory.

(b) **Associative memory:** a wide variety of information is stored in associative memories, such as the maintained information about the environment in certain times, a list of environment variables and the probability of their state change [17], the probability of occurrence of a suitable solution in a certain

region [18], the probability vector for generating the best solution [19], the probability of feasible regions [20], and so forth.

**2.2.2.3. Other approaches.** Other methods have also been presented for maintaining the diversity in the environment after detecting change in the environment. In [21], a Sentinel Placement method is used for diversity maintenance. In this method, a series of sentinels distributed in the search space are utilized in order to generate a new population. In [22], using Random Immigrant method has been proposed in which in every generation a number of random solutions are added to the population to preserve diversity. In [23] a multi-objective method is proposed in which two goal functions are implemented. The first one is the main goal function, and the second one is a goal function for keeping diversity in the environment. Furthermore, in [24], a method based on fitness sharing is suggested in order to preserve diversity.

### 2.3. Hybrid methods

This category is a hybrid of both presenting diversity and diversity maintenance methods. This means that diversity is maintained during execution, and also diversity creation is considered in the environment, in case of change in the environment. The algorithms presented in this category can be categorized as follows:

#### 2.3.1. Multi-population

In this subcategory of algorithms, a number of subpopulations are used for covering different regions of the search space. All subpopulations usually have the similar tasks, but they may also have different tasks. In [25], a method called Shifting Balance Genetic Algorithm (SBGA) has been proposed in which a number of small subpopulations are responsible for global search in the problem space, and a large subpopulation is responsible for tracking the changing peaks. Another approach is suggested in [26] and it is called Self Organizing Scouts (SOS), which contrarily uses a big subpopulation for global search and a number of small subpopulations for tracking changes. This strategy has also been proposed with other meta-heuristic methods such as Genetic Algorithm in [27] and Differential Evolution in [28]. In [29] and [30], two methods similar to SOS are proposed, called FMSO and MPSO, in which a parent type is responsible for exploring the search space for discovering existing promising areas in the environment, and a series of child types has the task of local search. Another approach is to use a population for both local search and global search simultaneously. In [31], a population is first used to global search, and when an optimum is discovered, the population is divided into two subpopulations, one responsible for tracking optimum changes and another responsible for global search. In [32–34] speciation based PSO approaches are proposed for optimization in dynamic environments. Also in [35], a regression-based approach named rSPSO is suggested for enhancing the convergence rate using speciation based methods. In [36], a technique named SPSO is proposed in which every cluster is divided into two. The first cluster is responsible for exploitation and the second one is in charge of exploration. In [37], a method based on clustering is proposed for creating subpopulations, and in [38], this method has been improved and has been named PSO-CP. It uses simplifications such as removing the learning procedure, and reducing the number of phases for clustering from two phases to only one phase. In [39] two multi-population methods, namely mQSO and mCPSO, are proposed. In the former one, quantum particles and in the latter one, charged particles are used to generate diversity. In [40], an approach for enhancing this method is proposed by making the number of subpopulations adaptive, and it is named AmQSO. It has significantly improved the performance of the algorithm. In [41] a method for optimization in dynamic environments is proposed and it is based on composite

```

for each Particle  $i \in [1..N]$ 
    initialize  $x_i, v_i$ 
     $P_i = x_i$ 
endfor
 $G = \arg \min_{P_i} f(P_i)$ 
repeat:
    for each Particle  $i \in [1..N]$ 
        update  $v_i$  using Eq. (1)
        Check the velocity boundaries.
        update  $x_i$  using Eq. (2)
        perform absorbing walls [44,60] for particle position ;
        if  $f(x_i) \leq f(P_i)$  then
             $P_i = x_i$ 
        endif
        if  $f(P_i) < f(G)$  then
             $G = P_i$ 
        endif
    endfor
until stopping criterion is met

```

Fig. 1. PSO algorithm pseudo-code [44,60].

particles. This method has an acceptable efficiency. Finally, in [58] a new approach based artificial fish swam algorithm represented.

### 3. Finder-tracker multi-swarm PSO (FTPSO)

In this section, a novel algorithm based on PSO is proposed for optimization in dynamic environments. PSO is one of the swarm intelligence methods proposed by Kennedy and Eberhart in 1995 [42]. In the following, we briefly describe PSO algorithm as the base approach of the proposed method.

Let  $N$  be the population size. For the  $i$ th particle ( $1 \leq i \leq N$ ) in a  $D$ -dimensional space, the current position is  $x_i = (x_{i1}, x_{i2}, \dots, x_{iD})$ , and the velocity is  $v_i = (v_{i1}, v_{i2}, \dots, v_{iD})$ . During the optimization process, the velocity and the position of every particle in every step is updated using Eqs. (1) and (2):

$$v_{i,j}(t+1) = \chi(v_{i,j}(t) + c_1 R_{i,j}^1 (P_{i,j}(t) - x_{i,j}(t)) + c_2 R_{i,j}^2 (G_j(t) - x_{i,j}(t))) \quad (1)$$

$$x_i(t+1) = x_i(t) + v_i(t+1) \quad (2)$$

In which  $v_{ij}$  is equal to the  $j$ th dimension of the  $i$ th particle velocity.  $c_1$  and  $c_2$  are acceleration coefficients.  $R^1$  and  $R^2$  are two random numbers with a uniform distribution in the range of [0,1].  $\chi$  is the constriction factor which reduces the particles velocity during the optimization process [43].  $P_i(t)$  is the best position found so far by particle  $i$  by the time  $t$  (particle  $i$ 's best individual experience) and  $G(t)$  is the best position found so far by all members by the time  $t$  (the best group experience). Particle  $i$ 's best individual experience can be computed using Eq. (3):

$$P_i(t+1) = \begin{cases} P_i(t), & \text{iff } f(x_i(t+1)) \geq f(P_i(t)) \\ x_i(t+1), & \text{iff } f(x_i(t+1)) < f(P_i(t)) \end{cases} \quad (3)$$

in which  $f(x)$  is the fitness value of vector  $x$ . The best group experience is calculated by Eq. (4):

$$G(t+1) = \arg \min_{P_i} f(P_i(t+1)), \quad 1 \leq i \leq N \quad (4)$$

Pseudo-code of PSO algorithm is demonstrated in Fig. 1.

The proposed algorithm which is extended version of PSO algorithm has been designed based on particular challenges of dynamic environments. The proposed algorithm uses various mechanisms in order to remove the dynamic environment challenges, and to

```

Finder_move()
inttracker_size = number of particles in each tracker_swarm
for each particle  $i$  infinder_swarm
    update  $V_{F,i}$  based on Eq. (1);
    check Velocity boundaries;
    update  $X_{F,i}$  based on Eq. (2);
    perform absorbing walls [44,60] for particle position ;
    update PbestF,i based on Eq. (3);
    update GbestF based on Eq. (4);
endfor

```

**Fig. 2.** Pseudo-code for *tracker* swarm movement.

improve efficiency. In the following, the various mechanisms used in the proposed algorithm will be discussed.

### 3.1. Eliminating the potential optimums' coverage challenge

As previously mentioned, there are many peaks in most of dynamic environments, every one of which can change into a global optimum after environment change. This implies that each peak can be a potential optimum. Therefore the algorithm which is designed for optimization in a dynamic environment should observe all the peaks so as to quickly detect the optimum peak after environment change.

In the proposed algorithm, a multi-swarm mechanism is used to cover the peaks. There are several particle swarms in the proposed algorithm which are responsible to observe all the existing peaks in the problem space. The mechanism used in the proposed algorithm to control swarms in a *finder-tracker* type. Hence, there is one *finder* swarm and several similar *tracker* swarms. The *finder* swarm is the one whose number of particles exceeds that of the *tracker* swarms, and is responsible for finding the peaks. At the beginning of algorithm execution, only the *finder* exists in the search space, and other swarms are inactive, i.e. their particles make no move. At first, the particles in the *finder* swarm are randomly initialized in the problem space, and they start the search. When the *finder* converges into a peak, it activates a *tracker* swarm to be replaced on the *finder* position placed at the peak. After activation and placement at the peak found by the *finder*, the *tracker* swarm covers the peak and will be responsible for following that peak after environment change. Furthermore, finding the top of the peak by a local search is another responsibility of the *tracker* swarm placed at that peak. The pseudo-code for the movement of the *finder* is shown in Fig. 2.

As already mentioned, the number of particles in the *finder* swarm exceeds the number of particles in each *tracker* swarm. Thus, during the replacement of a *tracker* swarm on the *finder* swarm, the better particles' parameters in the *finder* swarm are copied into those of the *tracker* swarm. For instance, if there are 10 particles in the *finder* swarm and 6 in the *tracker* swarm, then the *velocity*, *position*, and *Pbest* components of the 6 particles in the *finder* swarm with a better *Pbest fitness value* are copied into those 6 particles in the *tracker* swarm. After *tracker* swarm activation, the *finder* particles are once more randomly initialized in the problem space. Therefore, performing local search and covering the peaks are the functions of the *tracker* swarm particles, and the *finder* swarm starts its search for discovering other peaks not already found and not already covered by active *tracker* swarms. The aforementioned process will continue until all the peaks are covered by *tracker* swarms, and the *finder* replaces a *tracker* swarm after finding each uncovered peak and activates it. The *finder* has found a peak and has converged to it, when the Euclidean distance of *Gbest* position in its  $m$ th iteration and *Gbest* position in its  $(m + n)$ th iteration is less than a threshold value called *Conv\_limit*. The pseudo-code for activation and replacing a *tracker* swarm in a newly found peak by the *finder* is shown in Fig. 3.

```

Activation()
if finder_swarm is converged then
    list = sort particles in finder_swarm based on their Pbest_fitness_value in ascending order
    activate an inactive_tracker_swarm y
    for counter=1 to tracker_size
        Xy,counter=XF,list counter;
        Vy,counter=VF,list counter;
        Pbesty,counter=PbestF,list counter;
    endfor
    Gbesty = GbestF;
    Reinitializefinder_swarm;
endif

```

**Fig. 3.** Pseudo-code for activating a *tracker* swarm.

If the *finder* finds a peak which is already found, then it is reinitialized without activating a *tracker* swarm. In fact, the peaks previously found by the *finder* are covered by one active *tracker* swarm. As a result, when the Euclidean distance of *Gbest* position for the *finder* swarm and *Gbest* position of an active *tracker* swarm is less than a certain number  $r_{excl}$ , it means that the *finder* has converged into a peak already found. The value of  $r_{excl}$  in the proposed algorithm is determined according to [39].

In [39] it was illustrated that two swarms whose *Gbests* Euclidean distance is less than  $r_{excl} = 0.5 \times (X/p^{1/d})$  have converged into one peak. In the mentioned equation,  $p$  is the number of peaks,  $d$  is the number of dimensions in the problem space, and  $X$  indicates the length of the search space in every dimension. So in every iteration of algorithm execution, the Euclidean distance of *Gbest* position for the *finder* swarm and *Gbest* position of all active *tracker* swarms in the problem space is calculated, and if its amount is less than  $r_{excl}$  for any of the *tracker* swarms, then the *finder* is reinitialized. The pseudo-code for establishing exclusion between the *finder* swarm and the *tracker* swarms is illustrated in Fig. 4.

Sometimes, it is possible that the *finder* is converged before reaching a peak. Therefore, it replaces a *tracker* swarm in its place and activates it. This *tracker* swarm may move toward a peak which is previously covered by another active *tracker* swarm. As a result, there will be two active *tracker* swarms in a peak. Moreover, sometimes two covered peaks may get very close to each other and the *tracker* swarm in one peak may leave its covered peak and move into the other peak. This also means that one peak will be covered by two active *tracker* swarms. Such situations not only do not improve results, but also lead to performing inefficient fitness evaluations. To solve this problem, the Euclidean distance of *Gbest* position in all involving active *tracker* swarms should be calculated in every iteration. Therefore, two swarms with a *Gbest* distance of less than  $r_{excl}$  are placed at one peak. In this case, the swarm with worse fitness value will be deactivated, and the other swarm will continue its function. This situation is a local relation between activated *tracker* swarms whose *Gbests* distance is less than  $r_{excl}$ . The pseudo-code for establishing exclusion among active *tracker* swarms is illustrated in Fig. 5.

### 3.2. Environment change

One of the major challenges in dynamic environments is detecting environment change and to adapt the algorithm against the outcomes of the problems. As mentioned in Section 2, after an environment change, optimization algorithms encounter two prominent problems of invalid memory and remarkable diversity reduction in swarms. Consequently, the algorithms designed for

```

finder_exclusion()
for each active_tracker_swarmi
    if Euclidian distance between GbestF and Gbesti is less than rexclthen
        Reinitializefinder_swarm;
    endif
endfor

```

**Fig. 4.** Pseudo-code for exclusion between the *finder* swarm and *tracker* swarms.

```

trackers_exclusion()
for each active_tracker_swarm i
    for each active_tracker_swarm j
        if Euclidian distance between Gbesti and Gbestj is less than rexcl then
            if F(Gbesti) < F(Gbestj) then
                deactivate tracker_swarm j;
            else
                deactivate tracker_swarm i;
            endif
        endif
    endfor
endfor

```

**Fig. 5.** Pseudo-code for exclusion among tracker swarms.

dynamic environments should be able to detect the environment changes quickly so that they can use their utilized mechanisms to eliminate these two problems.

To detect environment change in the proposed algorithm, a point with a random position called *test\_point* is selected in the search space at the beginning of algorithm execution, and its fitness value is stored. In every iteration of the proposed algorithm execution, the fitness value of *test\_point* is evaluated and compared to its previous result. If the two results in two subsequent executions are the same, it means the environment has not changed. On the other hand, their difference illustrates a change in the environment.

It must be noted that using *test\_point* is only adequate for those dynamic environments in which change occurs globally. If the change is local, then it may not occur in the vicinity of *test\_point* position, and therefore using *test\_point* can no longer be useful to detect environment change. If changes happen in the neighborhood of an optimum, *Gbest* position of the best swarm can be used as *test\_point*. To do so, *Gbest* value of the best swarm is evaluated at the end of every iteration of algorithm execution, and this value is compared with the value previously stored. If the two values are different, it means the environment has changed. But if the change is local, and the change neighborhood in the environment is also undetermined, then *Gbest* fitness value must be evaluated for all swarms at the end of each iteration. If the obtained results in any of the swarms differ from the stored value, it means the environment has changed.

After detecting change in the environment, the mechanisms for eliminating the problems of diversity loss in the swarms and invalid memory will be deployed.

After detecting change in the environment in the proposed algorithm, the diversity increase mechanisms are executed first. In fact, the diversity problem in the proposed algorithm results from the fact that before the environment change, the existing particles in *tracker* swarms have converged into the peak positions they have covered. In this situation, the distance of particles positions in each of the *tracker* swarms are considerably close to each other and their *velocity* components will be near zero. Also, the distance of particles *Pbest* positions and the swarm *Gbest* will be substantially close to each other.

In such conditions, the particles positions and their *Pbest* will remain around the previous position of the peak after relocating the peak covered by a *tracker* swarm, and they cannot move toward the new position of the peak. The reason behind this problem is that in PSO, particles movement toward the new position is according to their velocity obtained by Eq. (1). As can be seen in Eq. (1), the particles velocity is determined by their previous velocity, the vector between particles positions and *Pbest*, and the vector between particles positions and *Gbest*, all of which are near zero after the environment change. As a result, the diversity in the swarm will be considerably low, and the particles cannot move.

To solve this problem in the proposed algorithm, a new mechanism is deployed on the active *tracker* swarms. In this mechanism, the positions of all the particles in the *tracker* swarms will change

according to their position before the environment change. Before the environment change, *Gbest* is the nearest position that every active *tracker* swarm has found to its covered peak. After the environment change, the new position of the peak will be inside the space with a radius of *Shift length* according to its previous position. Therefore, to increase the speed of finding the new position of the peak, the particles should be uniformly and randomly distributed in the vicinity of the swarm *Gbest* position and in the space whose difference value from its previous position is determined by *Shift length*. Consequently, a swarm will expand according to the space that should be searched for better values. After the environment change, the new position of particle *j* in the *tracker* swarm *i* can be calculated by Eq. (5):

$$X_{i,j} = Gbest_i + (Rand^D(-1, 1) \times P \times Severity) \quad (5)$$

in which *D* is the number of problem dimensions, and *Rand* function produces a *D*-dimension vector of random numbers with a uniform distribution in the range of  $[-1, 1]$ . *P* determines the radius based on which the particles should scatter around *Gbest* according to *Shift length*. Indeed,  $P \times Shift\ length$  indicates the maximum distance value that particles positions can have according to *Gbest* in every dimension.

As it was mentioned, the *velocity* components value of the particles is near zero after the environment change. To increase diversity in the proposed algorithm, the *velocity* components value of the particles is determined randomly, according to *Shift length* after detecting environment change. Therefore, by adjusting the *velocity* and position of the particles based on *Shift length*, we can put the *tracker* swarms in a condition where the new position of the peak in the new environment could be found quickly. The particles *velocity* vector is calculated by Eq. (6):

$$V_{i,j} = Rand^D(-1, 1) \times Q \times Severity \quad (6)$$

in which *Q* determines the maximum percentage of *Shift length* by which the particles *velocity* components should be chosen.

When the position of the particles in active *tracker* swarms has been chosen according to their *Gbest* and *Shift length*, every particle's *Pbest* is equal to its new position ( $Pbest_{i,j} = X_{i,j}$ ). Then, the new *Pbest* fitness values are calculated and their best will be considered as *Gbest*. So the previous memory of the particles in active *tracker* swarms resets and the invalid memory problem is also resolved.

There is no need to increase diversity in the *finder* swarm after the environment change, since this swarm is automatically reinitialized automatically after convergence, but its associated memory should be updated so that it can continue to search with a valid memory after the environment change. To update the memory of the *finder* swarm, *Pbest* position fitness for all particles in the *finder* swarm is calculated first, and then their best value is considered as *Gbest*. The pseudo-code for the mechanisms after detecting the environment change is shown in Fig. 6.

### 3.3. Mechanisms for performance improvement

Two different mechanisms are used in the proposed algorithm to improve its performance, which are discussed as follows.

#### 3.3.1. Awakening and sleeping active *tracker* swarms

After the environment change, every active *tracker* swarm moves toward the peak where it is placed by performing a local search. Among the active *tracker* swarms, this task is much more important for the swarm placed near the highest peak. In fact, the value of the *current error* is determined according the results of the swarm whose *Gbest* fitness value is better than the rest of the swarms. Therefore, other active *tracker* swarms do not contribute to determining the result in the current environment, but performing a local search is important for them as well. If these swarms do

```

After Environment Change()
for each active_tracker_swarm
    for each particle j
        Update  $X_{i,j}$  using Eq. (5)
        Update  $V_{i,j}$  using Eq. (6)
         $Pbest_{i,j} = X_{i,j}$ 
        Evaluate  $f(Pbest_{i,j})$ ;
    endfor
     $Gbest_i = \arg \max_{Pbest_{i,j}} f(Pbest_{i,j})$ 
endfor
for each particle k in finder_swarm
    Evaluate  $f(Pbest_{F,k})$ ;
endfor
 $Gbest_F = \arg \max_{Pbest_{F,k}} f(Pbest_{F,k})$ 

```

**Fig. 6.** The pseudo-code for mechanisms after detecting the environment change.

not perform a local search, their distances from their peaks may increase after several environment changes and peak relocations, and they may even lose it. As a result, performing a local search is essential for all active *tracker* swarms after the environment change.

After a swarm approaches its goal by performing a local search, it performs yet another local search in order to improve the accuracy of the result. As previously noted, the obtained result from every environment is determined by the swarm residing near the highest peak. Therefore, this swarm should perform more accurate local searches to improve results. One way to perform more accurate local searches is to give more chance for performing another local search. But for performing such a task in dynamic environments, there is a little time until the environment change, due to the high frequency of the environment change influenced by the number of fitness evaluation. As a result, each swarm will have a limited number of iterations to perform local searches before the next environment change, and this can turn into a challenge when the number of active *tracker* swarms in the problem space is considerable. In this case, the algorithm performs many fitness evaluations in every iteration of its execution, and so the swarms find a limited number of chances to be executed before the next environment change.

To improve performance and to give more chance for the swarm nearest to the highest peak, a *sleeping–awakening* mechanism is used for the active *tracker* swarms in the proposed algorithm. In this mechanism, the active *tracker* swarms can have two different states of *sleeping* and *awakening*. The *awake* *tracker* swarm is the swarm whose particles exist in the problem space and evaluate fitness in the search process, and whose particles relocate in every iteration of the algorithm execution. The *asleep* *tracker* swarm is the swarm whose particles exist in the problem space, but they do not move and do not perform fitness evaluation.

In this mechanism, all active *tracker* swarms are awakened after each environment change, and they perform the optimization process. As discussed earlier, the active *tracker* swarms placed at non-optimum peaks should perform a local search after each environment change so as not to increase their distance from their covered peaks. However, continuing the local search for improving accuracy, after drawing near their peaks, is not helpful for these swarms and has no effect on the algorithm outcome. Therefore, as soon as an active *tracker* swarm residing at a non-optimum peak gets close to its goal, it is asleep.

By applying this mechanism, after active *tracker* swarms placed at non-optimum peaks in the current environment have reached

```

sleeping–awakening ()
for each non best active_tracker_swarm
    If all dimension of V for each particle j is  $\epsilon [-L,L]$  then
        Sleep[i]=True
    endif
endfor

```

**Fig. 7.** Pseudo-code for *sleeping–awakening* mechanism.

their goals, they are asleep and they stop fitness evaluation until the next environment change. So a considerable number of ineffective fitness evaluations are eliminated and this time could be used to give more chances to the swarm placed at the global optimum peak. Therefore, it leads to performing more local searches in the neighborhood of the global optimum position, which results in an improvement in performance and accuracy of the algorithm.

In the proposed algorithm, particles *velocity* components are used to determine whether a swarm has reached its goal or not. Indeed, when a swarm converges to the position of its goal and approaches it, particles *velocity* decreases. In the proposed method, there exists an  $L$  parameter. When all particles *velocity* components in an active *tracker* swarm are in the range of  $[-L,L]$ , it means the swarm has approached its goal, and it must be asleep. It is worth mentioning that in every environment, the swarm including the best  $Gbest$  fitness value among all active *tracker* swarms will not be asleep. Furthermore, all asleep swarms will awaken after detecting environment change. The pseudo code for *sleeping–awakening* mechanism is depicted in Fig. 7.

Therefore, the movement of active *tracker* swarms is performed as illustrated in pseudo code in Fig. 8, according to the *sleeping–awakening* mechanism.

### 3.3.2. Using exploiter particle

In the proposed algorithm, a new strategy based on an exploiter particle is used in order to increase local search capability around the best position found by the active *tracker* swarms. For this purpose, there is a cloud with a radius  $r_{cloud}$ , whose center is the  $Gbest$  position of the *tracker* swarm involving the best  $Gbest$  fitness value among all *tracker* swarms. There is a exploiter particle which can relocate in the cloud  $n$  times. The exploiter particle position in the cloud is determined randomly according to Eq. (7):

$$X_R = Gbest_{best} + Rand^D(-r_{cloud}, r_{cloud}) \quad (7)$$

in which  $X_R$  shows the exploiter particle position, and  $Gbest_{best}$  is the best  $Gbest$  in all active *tracker* swarms. *Rand* function generates a  $D$ -dimensional vector of random numbers in the range of  $[-r_{cloud}, r_{cloud}]$  with a uniform distribution. Every time the exploiter particle position is obtained by Eq. (7), its fitness value is calculated and if its fitness value is better than  $Gbest_{best}$  fitness value,  $Gbest_{best}$  position is updated as shown in Eq. (8):

$$Gbest_{best} = X_R \quad (8)$$

```

Tracker_move()
for each Active_tracker_swarm j
    if Active_tracker_swarm j is awake then
        for each particle k in Active_tracker_swarmj
            update  $V_{j,k}$  based on Eq. (1)
            check Velocity boundaries
            update  $X_{j,k}$  based on Eq. (2)
            perform absorbing walls [44,60] for particle position ;
            update  $Pbest_{j,k}$  based on Eq. (3)
            update  $Gbest_j$  based on Eq. (4)
        endfor
    endif
endfor

```

**Fig. 8.** Pseudo-code for movement of tracker swarms.

```

Random_local_searching()
forE_try=1 : n
    Obtain Exploiter particle position XR by Eq. (7)
    if f(Gbestbest) < f(XR) then
        Gbestbest = XR
    endif
endfor
    Obtain CF using Eq.( 9)
    Apply Eq. (10) on rCloud based on CF

```

**Fig. 9.** Pseudo-code for random local searching.

If Eq. (8) is performed, the position of the cloud center, i.e.  $Gbest_{best}$ , changes. So the position of the cloud will also change. But if the position of the exploiter particle is worse than  $Gbest_{best}$ , Eq. (8) will not be performed, and  $Gbest_{best}$  position and the cloud position will not change either. Eq. (7) will perform  $E\_try$  times for the exploiter particle, and in the best case,  $Gbest_{best}$  position will improve  $E\_try$  times and in the worst case, its position will not encounter any changes.

Since the exploiter particle is used for performing local searches in the proposed algorithm, the size of the cloud should match the space which is searched for finding better positions. For this purpose, the cloud should decrease as the local searches improve and the space explored for finding better values decreases. Consequently, the probability of finding better positions increases by using Eq. (7).

To decrease  $r_{cloud}$  value, we should multiply its value by a positive number less than one called *contraction factor* (CF) in every iteration. In the proposed algorithm, an equation for generating a random number is used to calculate this number in every iteration, as shown in Eq. (9):

$$CF = CF_{min} \times (Rand \times (1 - CF_{min})) \quad (9)$$

in which  $Rand$  is a function for random number generation in the range of [0,1] with a uniform distribution, and  $CF_{min}$  is the lower bound of contraction factor value. The reason for using Eq. (9) is that this equation can generate the contraction factor without considering the problem information, particularly the environment *change frequency*. So it can be used in different situations in dynamic problems. Determining  $CF$  should be randomly done in an adequate range, i.e.  $[CF_{min}, 1]$ . Therefore choosing a suitable  $CF_{min}$  will have an important role in generating an adequate  $CF$ . After choosing a  $CF$  by Eq. (9),  $r_{cloud}$  value will be updated according to Eq. (10):

$$r_{cloud} = r_{cloud} \times CF \quad (10)$$

It must be noted that after the environment change,  $r_{cloud}$  value will be reset so as to perform a local search in the neighborhood of a new position. The pseudo-code of the final module of the proposed algorithm, i.e. exploiter particle is illustrated in Fig. 9. Finally, after describing all modules of the proposed algorithm, the main procedure for performing the proposed algorithm is demonstrated in Fig. 10.

#### 4. Experiments

In this section, we have tested proposed algorithm on Moving Peaks Benchmark (MPB) [45,53], which is the best-known benchmark in dynamic environments optimization [54]. The reason for its popularity among the researchers is that this benchmark can produce numerous and various conditions and situations of dynamic environments. By utilizing this benchmark, we can study algorithms performances from different aspects.

In order to measure the efficiency of the algorithms, *offline error* that is the average of the difference between fitness of the best

```

Proposed Algorithm:
//finder_swarm Initialization
for each particle I in finder_swarm
    Randomly initialize VF,i, PbestF,i = XF,i
    Evaluate f(PbestF,i)
    GbestF=arg max(f(PbestF,i))
endfor
Randomly initialize Test_point
Evaluate f(Test_point)
Repeat
    Finder_move();
    Finder_exclusion();
    Activation();
    Trackers_move();
    Random_local_searching();
    Trackers_exclusion();
    Sleeping_awakening();
    Evaluate f(Test_point)
    if new value is different from previous value then
        Wake up all asleep active_tracker_swarm;
        After_enviroment_change();
    endif
Until number of performed function evaluations > max

```

**Fig. 10.** Pseudo-code of main procedure of the proposed algorithm.

solution found by the algorithm and fitness of the global optimum is used [45,53].

$$offline\_error = \frac{1}{FES} \sum_{t=1}^{FES} fitness(Gbest(t)) - fitness(global\ optimum(t)) \quad (11)$$

where  $FES$  is the maximum *function evaluations*, and  $Gbest(t)$  and  $global\ optimum(t)$  are the best position found by the algorithm and the global optimum at the  $t$ th fitness evaluation, respectively. In other word, the amount of *offline error* equals to the average of all *current errors* which is defined in time  $t$  as deviance between the best found position by the algorithm in time  $t$  in the current environment and the position of the global optimum in the current environments. The value of *Offline error* is constantly a non-negative number, where it is zero in the ideal situation.

#### 4.1. Reflections on parameter settings

In this section, the effect of different values of the parameters of the proposed algorithm is studied on the algorithm efficiency. The main objective in this section is to determine appropriate parameter values by performing various experiments. The experiments in this section are executed in MPB Scenario 2, which is considered as one of the best-known benchmarks. MPB parameters setting for this scenario, also called standard setting for this benchmark [45,53], is illustrated in Table 1.

The initial configuration of the proposed algorithm is as follows:  $r_{excl}$  parameter is considered according to [39]; the number of particles in *tracker swarms* is equal to 5 as mentioned in [40]. The number of particles in the *finder swarm* is 10 by default. The values of  $P$  and  $Q$  in Eqs. (5) and (6) are set to 0.5 by default. In addition, two mechanisms including exploiter particle and *awakening–sleeping* are inactive. Every experiment is executed 50 times. Also, in every experiment the initial position of the particles and the peaks are determined randomly and with different random seeds. Termination condition of the algorithm is 100 environment changes or, in other words, 500,000 fitness evaluations.

**Table 1**

Setting standard parameters for MPB.

Parameter	Value
Number of peaks, $M$	10
Change frequency	5000
Height change	7.0
Width change	1.0
Peaks shape	Cone
Basic function	No
Shift length	1.0
Number of dimensions, $D$	5
Correlation coefficient, $\lambda$	0
Peaks location range	[0–100]
Peak height	[30.0–70.0]
Peak width	[1–12]
Initial value of peaks	50.0

#### 4.1.1. The effect of different values of Conv\_limit

In this sub-section, different cases are studied for determining the convergence of the *finder* swarm. *Conv\_limit* parameter determines the threshold for *finder* convergence. To determine convergence, the Euclidean distance of the swarm *Gbest* position in its current iteration and its position in  $k$  previous iterations (i.e. the  $(m-k)$ th and  $m$ th iterations) are calculated and are then compared with *Conv\_limit*. In fact, if the amount of relocation for *Gbest* position of the *finder* swarm in  $k+1$  consecutive iterations is less than *Conv\_limit*, it implies that the *finder* swarm has converged. *Offline error* and *standard error* results along with the average of the number of active *tracker* swarms during the optimization process are tabulated in **Table 2**, obtained from different values of *Conv\_limit*. Here, the value of  $k$  is set to 2.

As it is evident in the results presented in **Table 2**, decreasing the values of *Conv\_limit* leads to degrading the efficiency of the proposed algorithm. The reason for such deterioration in efficiency is that the *finder* convergence condition will not be met in an appropriate time, and it will take more time for the *finder* swarm to place a *tracker* swarm in its position and continue its search for finding other peaks. Thus, the rate of finding various peaks by the *finder* is reduced. As a result, it will take more time for all the peaks to be fully covered and the efficiency of the algorithm decreases. By increasing the value of *Conv\_limit* up to 1, the algorithm efficiency increases as well, but increasing its value beyond 1 will decrease efficiency. In fact, a high value of *Conv\_limit* results in a quick convergence for the *finder* and this problem activates those *tracker* swarms in the problem space whose existence is not useful. In other words, it will be possible for a *finder* to meet its convergence condition before approaching a peak, and as a result it activates a *tracker* swarm. So there will be more active *tracker* swarms in the problem space than what is required, and these swarms will perform too many useless fitness evaluations before they are deactivated by *trackers\_exclusion()* function.

An average number of active *tracker* swarms for different values of *Conv\_limit* is presented in **Table 2**. We can observe that decreasing the value of *Conv\_limit* leads to more time for the peaks

**Table 2**The obtained results from the proposed algorithm with different values of *Conv\_limit* parameter on Scenario 2 of MPB.

<i>Conv_limit</i>	<i>Offline error ± standard error</i>	Average number of active <i>tracker</i> swarm
0.01	1.1699 ± 0.0612	9.1634
0.05	1.1304 ± 0.0686	9.3519
0.1	1.2096 ± 0.0324	9.5137
0.5	1.0792 ± 0.0555	9.7546
1	<b>1.0104 ± 0.0353</b>	<b>9.9718</b>
2	1.0489 ± 0.0512	11.4867
5	1.1094 ± 0.0403	13.5680

**Table 3**The obtained results from the proposed algorithm with different values of  $k$  on Scenario 2 of MPB.

$k$	<i>Offline error ± standard error</i>
1	1.2365 ± 0.0844
2	<b>1.0104 ± 0.0353</b>
3	1.0408 ± 0.0772
4	1.1832 ± 0.1117
5	1.2116 ± 0.1260

to be covered by *tracker* swarms. Furthermore, if this parameter increases exceedingly, the number of useless active *tracker* swarms will increase as well. For instance, when *Conv\_limit* is set to 5, there will be 3.5680 active *tracker* swarms in the problem space in average, regarding this fact that there are 10 peaks in the space. To conclude, a value of 1 for *Conv\_limit* will be considered as an appropriate value, yielding better results compared to other values.

Concerning the details presented in Section 4.1, the metric for convergence is determined by the Euclidean distance of the *finder* swarm's *Gbest* in its  $(m-k)$ th and  $m$ th iterations. The reason we have for setting  $k$  as 2 is that this level of difference is most appropriate for determining convergence in this swarm. **Table 3** shows the effect of different values of  $k$  on the efficiency of the proposed algorithm, with a *Conv\_limit* equal to 1.

As observed from **Table 3**, the efficiency of the proposed algorithm with  $k=2$  is the best state, compared to other values of  $k$ . When *Gbest* positions of the *finder* swarm are compared in two consecutive iterations of the algorithm ( $k=1$ ), for determining convergence, we will be unable to determine the convergence properly and it will be possible to wrongly consider the *finder* swarm as converged while it has not converged yet. Because it will be possible that before convergence, the particles of the *finder* swarm cannot find a better position, compared to *Gbest* in one iteration. The algorithm efficiency increases with  $k=2$ . Increasing  $k$  to values more than 2 results in efficiency reduction because more iterations will be needed to determine whether the *finder* swarm has converged, and generating new swarms for discovering other uncovered peaks will be postponed, which will lead to a decrease in efficiency.

#### 4.1.2. The effect of different values of *P* and *Q*

Two parameters of *P* and *Q* have been used in Eqs. (5) and (6) to control the diversity increase after discovering change in the environment. As it was discussed earlier, the new position of each peak after the environment change in every dimension will be at most in *Shift length* distance from its previous position. So it can be concluded that it is better for the particles to distribute in the same distance in order to increase diversity. This process is done by placing the particles at the previous position of the peak before the environment change. To increase diversity in the proposed algorithm, both *velocity* and *position* of the particles are modified using Eqs. (5) and (6). Since the particles should distribute at most in an area with a *Shift length* radius from their previous *Gbest* position, and considering the structure of Eqs. (5) and (6), it can be concluded that there is a tradeoff between two parameters *P* and *Q* and their summation should be equal to *Shift length*. Therefore, according to the new positions of the particles determined by Eq. (5) and their new velocity determined by Eq. (6), all the particles will be placed at *Shift length* radius of their previous *Gbest*, after one execution. **Table 4** illustrates the obtained results with different values of parameters *P* and *Q*.

As we can observe in **Table 4**, when *P* and *Q* are equal to 0.5, better results are obtained. Worst results are acquired when one of these parameters equals 0 and the other one equals 1. Indeed, in the latter case, diversity is tried to increase either by evaluating *velocity* using Eq. (6) or by distributing particles using Eq. (5). The results in **Table 4** show that as *P* and *Q* values move toward each other,

**Table 4**

Obtained results from the proposed algorithm with different values of parameters  $P$  and  $Q$  on Scenario 2 of MPB.

$P$	$Q$	Offline error ± standard error
0.0	1.0	1.3356 ± 0.0569
0.1	0.9	1.2027 ± 0.0909
0.2	0.8	1.1944 ± 0.0947
0.3	0.7	1.1393 ± 0.1087
0.4	0.6	1.0197 ± 0.0672
0.5	0.5	<b>1.0104 ± 0.0353</b>
0.6	0.4	1.0181 ± 0.1005
0.7	0.3	1.1349 ± 0.1418
0.8	0.2	1.1430 ± 0.0729
0.9	0.1	1.1958 ± 0.0538
1.0	0.0	1.2937 ± 0.0754

the efficiency of the mechanism for increasing diversity improves, and consequently, the swarms will find the new peak position more quickly after the environment change. Therefore the algorithm efficiency increases and it will yield better results. So the values of  $P$  and  $Q$  will be considered 0.5 from now on.

#### 4.1.3. The effect of different population size in the finder swarm

In this section we study the effect of the number of existing particles in the *finder* swarm on the efficiency of the proposed algorithm. **Table 5** illustrates the obtained results from the proposed algorithm with different number of particles in the *finder* swarm. As it can be observed, the efficiency of the algorithm is in its highest value with 10 particles in the *finder* swarm. When the number of particles in the *finder* swarm is less than 10, the efficiency of the swarm in finding peaks decreases. In fact, the convergence rate decreases when the population size decreases and in this case the swarm discovers the peaks at a later time and therefore the efficiency of the algorithm decreases as well.

By increasing the number of swarm particles beyond 10, once more the efficiency decreases. Indeed, by increasing the number of swarm particles the amount of fitness evaluation increases in the swarm and it leads to a high volume of fitness evaluations being executed by the swarm. Thus, there is no chance for performing a better search using other active *tracker* swarms. The following example completely clarifies this issue. If we consider 10 peaks in an environment with a *change frequency* of 5000 fitness evaluations, and 10 active *tracker* swarms, each *tracker* swarm has 5 particles, it means they perform 50 fitness evaluations in total in every iteration. This means there is possibility to execute 100 iterations before the environment change and they can move toward the goal. So, if the number of the particles in the *finder* swarm is 50, the number of iterations reduces to half and they may execute up to only 50 times before the next environment change. It implies that 2500 fitness evaluations are performed by the *finder* swarm in 50 iterations. This example shows how an excessive increase in the number of particles in the *finder* swarm can decrease the algorithm efficiency. So the number of the particles in the *finder* swarm will be considered 10 from now on.

**Table 5**

Obtained results from the proposed algorithm with different population sizes for the *finder* swarm on Scenario 2 of MPB.

Finder's population size	Offline error ± standard error
5	1.1100 ± 0.0518
7	1.0703 ± 0.0492
<b>10</b>	<b>1.0104 ± 0.0353</b>
12	1.0380 ± 0.0688
15	1.1045 ± 0.0754
20	1.1888 ± 0.1159
30	1.3775 ± 0.1518
50	1.6731 ± 0.1475

**Table 6**

Obtained results from the proposed algorithm with different values of  $E_{try}$  parameter on Scenario 2 of MPB.

$E_{try}$	Offline error ± standard error
5	0.9803 ± 0.0580
10	0.9760 ± 0.0921
15	0.9712 ± 0.0889
20	<b>0.9661 ± 0.0972</b>
30	1.0206 ± 0.1488
50	1.0647 ± 0.0550

#### 4.1.4. The effect of parameters involved exploiter particle mechanism

In this section, the exploiter particle mechanism is added to the proposed algorithm to improve its efficiency. Three parameters are effective on this efficiency:  $E_{try}$ ,  $r_{cloud}$  and  $CF_{min}$ . In the following, the effect of different values of these three parameters on the algorithm efficiency is studied.

##### (a) The effect of $E_{try}$

In this section,  $r_{cloud}$  value is considered 0.5 by default, according to [40] and we do not apply Eq. (10) on it. The goal in this section is to study the effect of the maximum number of times the exploiter particle may improve  $G_{best,best}$  in each iteration. **Table 6** illustrates the results obtained from different values of  $E_{try}$ . As it can be seen, the efficiency of the algorithm has improved by adding the exploiter particle mechanism for values 5, 10, 15, and 20, and it has decreased for values 30 and 50. In fact, excessively increasing the value of  $E_{try}$  leads to the same problem as excessively increasing the number of the *finder* swarm particles which was discussed in the previous section. Here, by raising the value of  $E_{try}$ , the number of fitness evaluations performed for the exploiter particle increases and it eliminates the chance for performing a better search using other active *trackers*. As a result, their distances from their covered peaks increase, and the efficiency of the algorithm accordingly decreases.

By determining a proper value for  $E_{try}$ , the algorithm efficiency will improve as a result of more local searches in the neighborhood of  $G_{best,best}$ . As we can see in **Table 6**, the result improves when  $E_{try}$  is equal to 20. Therefore the value of  $E_{try}$  will be 20 from now on.

##### (b) The effect of $cloud$ radius

$r_{cloud}$  shows the region in which the exploiter particle performs local searches. The value of this parameter should be set so that the probability of finding better positions increases by using Eq. (7). When  $r_{cloud}$  value is low, the convergence rate toward the position of the goal reduces, and when it is high, the chance for finding better positions using Eq. (7) reduces. Therefore the value of  $r_{cloud}$  should be set so that the search space of the exploiter particle is in accordance with the space the exploiter particle should search in pursuit of better positions. It can be concluded that the value of this parameter should be determined according to the value of *Shift length*. To this end,  $r_{cloud}$  should be a modulus of *Shift length*. In other words,  $r_{cloud}$  is obtained from the  $R \times Shift\ length$ . In **Table 7**, the effect of different values of  $R$  on the algorithm efficiency is illustrated with  $Shift\ length = 1$  and  $Shift\ length = 5$ .

As the results in **Table 7** show, when the initial value of  $r_{cloud}$  is considered less than or equal to *Shift length* after the environment change, the efficiency of the algorithm improves. In this case, the *local search* is performed well and the probability of success in Eq. (7) rises. However, when the initial value of  $r_{cloud}$  is considered greater than *Shift length* after the environment change, the probability of finding better positions using Eq. (7) declines. Therefore the exploiter particle cannot improve

**Table 7**

Obtained results from the proposed algorithm with different values of R and with Shift length equal to 1 and 5 on Scenario 2 MPB.

R	Offline error ± standard error	
	Shift length = 1	Shift length = 5
0.01	0.9141 ± 0.0811	3.0253 ± 0.1607
0.05	0.7906 ± 0.1031	2.8352 ± 0.1345
0.1	0.7533 ± 0.0923	2.8215 ± 0.2032
0.2	<b>0.7494 ± 0.0368</b>	<b>2.6343 ± 0.1944</b>
0.5	0.9440 ± 0.0889	3.7869 ± 0.2183
1	0.9588 ± 0.0809	4.2937 ± 0.2881
2	0.9665 ± 0.0897	4.3615 ± 0.2411
5	1.0640 ± 0.0736	4.5590 ± 0.2811

the efficiency of the algorithm by performing appropriate local searches. In this situation, not only does not the exploiter particle improve the efficiency of the algorithm, but also it causes the efficiency to deteriorate. The reason for such deterioration in the efficiency is performing useless fitness evaluations by the exploiter particle, and again there is no chance for performing a better search using other active *tracker* swarms. As can be seen in Table 7, the best results are obtained when the initial value of  $r_{\text{cloud}}$  is 0.2 of the Shift length after discovering the environment change. Henceforth, the initial value of  $r_{\text{cloud}}$  is considered as the mentioned value after discovering the environment change. This value will decrease after each iteration by Eq. (10).

#### (c) The effect of $CF_{\min}$

In the previous section, an appropriate value for  $r_{\text{cloud}}$  was specified for the exploiter particle. In this section, Eqs. (9) and (10) are added to the exploiter particle mechanism. By introducing these two equations, the value of  $r_{\text{cloud}}$  decreases during the local search process, and as was discussed in Section 4.3.2, by proper reduction of  $r_{\text{cloud}}$ , the ability of the exploiter particle for performing local searches will increase. In Table 8, the effect of different values of  $CF_{\min}$  parameter is illustrated, where  $CF_{\min}$  is the main parameter in determining the amount of decrease in  $r_{\text{cloud}}$ .

As it can be observed, the efficiency of the exploiter particle is improved when  $CF_{\min}$  is greater than or equal to 0.7, compared to the situation in which the  $r_{\text{cloud}}$  value is fixed (that is  $CF_{\min} = 1$ ). When  $CF_{\min}$  is less than 0.7, the efficiency of the exploiter particle, and consequently the efficiency of the algorithm decreases. In this case, the cloud decreases at a far higher rate than the progress of exploiter particle by Eq. (10), and it results in a decrease in the convergence rate of the exploiter particle. Therefore, the number of fitness evaluations performed by the exploiter particle in this case not only does not improve the efficiency, but also it eliminates the chance for performing a better search using other active *tracker* swarms. As we can see in Table 8, the most proper value for  $CF_{\min}$  is 0.8. As a result, the value of this parameter is considered 0.8 from now on.

**Table 8**

Obtained results from the proposed algorithm with different values of  $CF_{\min}$  parameter on Scenario 2 of MPB.

$CF_{\min}$	Offline error ± standard error
0.1	1.0661 ± 0.0625
0.2	1.0470 ± 0.1326
0.3	0.9854 ± 0.1047
0.4	0.9158 ± 0.0683
0.5	0.8370 ± 0.0839
0.6	0.7797 ± 0.1040
0.7	0.7387 ± 0.0450
0.8	<b>0.7183 ± 0.0977</b>
0.9	0.7242 ± 0.0583
0.95	0.7309 ± 0.0649
1	0.7494 ± 0.0368

**Table 9**

Obtained results from the proposed algorithm with different values of L on Scenario 2 of MPB.

L	Offline error ± standard error
0.01	0.7102 ± 0.0977
0.02	0.7079 ± 0.0697
0.05	0.7013 ± 0.0911
0.1	0.6973 ± 0.0661
0.2	0.6909 ± 0.0712
0.3	0.6843 ± 0.0637
0.4	<b>0.6752 ± 0.0394</b>
0.5	0.7345 ± 0.0577
0.6	0.7406 ± 0.0725

#### 4.1.5. The effect of parameter L

In this section, the *sleeping–awakening* mechanism is appended to the proposed algorithm. The efficiency of the mechanism depends on parameter L. In fact, L determines how long an active *tracker* swarm residing at a local optimum peak in the current environment is able to perform local searches. Table 9 shows the algorithm efficiency with different values of L. As we can see, introducing the *sleeping–awakening* mechanism with a proper value of L can improve the algorithm efficiency. When L is considered greater than or equal to 0.5, the algorithm efficiency declines. The reason for such degradation is that in this case, the active *tracker* swarms which are placed at local optimums in the current environment will be asleep too early, before they can find the opportunity to sufficiently approach the peak. Therefore, some swarms may get farther from their covered peaks. In this case, if one of these peaks becomes the global optimum after the environment change, it needs more time for reducing error, since the distance of the covering swarm is high. Thus, the efficiency is degraded.

On the other hand, considerable reduction in the value of L deteriorates the efficiency of *sleeping–awakening* mechanism as well, because it will take a long time for those active *tracker* swarms, which are awake in non-optimum peaks, to be asleep. Therefore, the number of fitness evaluations performed by the best swarm will decrease, and consequently the algorithm efficiency will also decrease. Among the values of L, the algorithm efficiency is improved, when L = 0.4. Henceforth, the value of L will be considered 0.4.

#### 4.2. Comparison with other algorithms

In this section, the proposed algorithm is tested on MPB with various configurations. The parameters values of the proposed algorithm are shown in Table 10. The obtained results are the outcome

**Table 10**

Parameter values of the proposed algorithm.

Parameters	Initial value
$C_1, C_2$	2.05 [43]
X	0.729843788 [43]
Trackers' population size	5 [40]
Finder's population size	10
Conv_limit (for determining finder convergence)	1
P (in Eq. (6))	0.5
Q (in Eq. (7))	0.5
$r_{\text{excl}}$ (for exclusion)	$0.5 \times (100 / (\text{peak.number}^{1/\text{dimension}}))$ [39]
E_try (in exploiter particle mechanism)	20
$R_{\text{cloud}}$ (radius of cloud)	$0.2 \times \text{Shift length}$
$CF_{\min}$ (in Eq. (10))	0.8
L (in sleep.wake mechanism)	0.4
Termination condition	Reaching “change frequency × 100” function evaluations

**Table 11**

Comparison of offline error [45] (standard error) of 6 algorithms on MPB problem with different number of peaks and change frequency and Shift length = 1.

Algorithm	C-F	Number of peaks							
		1	5	10	20	30	50	100	200
mQSO(5,5q) [39]	500	33.67(3.42)	11.91(0.76)	9.62(0.34)	9.07(0.25)	8.80(0.21)	8.72(0.20)	8.54(0.16)	8.19(0.17)
Am SO [40]		3.02(0.32)	5.77(0.56)	5.37(0.42)	6.82(0.34)	7.10(0.39)	7.57(0.32)	7.34(0.31)	7.48(0.19)
mPSO [29]		8.71(0.48)	6.69(0.26)	7.19(0.23)	8.01(0.19)	8.43(0.17)	8.76(0.18)	8.91(0.17)	8.88(0.14)
HmPSO [46]		8.53(0.49)	7.40(0.31)	7.56(0.27)	7.81(0.20)	8.33(0.18)	8.83(0.17)	8.85(0.16)	8.85(0.16)
APSO [47]		4.81(0.14)	4.95(0.11)	5.16(0.11)	5.81(0.08)	6.03(0.07)	5.95(0.06)	6.08(0.06)	6.20(0.04)
FTMPSO		<b>1.76(0.09)</b>	<b>2.93(0.18)</b>	<b>3.91(0.19)</b>	<b>4.83(0.19)</b>	<b>5.05(0.21)</b>	<b>4.98(0.15)</b>	<b>5.31(0.11)</b>	<b>5.52(0.21)</b>
mQSO(5,5q) [39]	1000	18.60(1.63)	6.56(0.38)	5.71(0.22)	5.85(0.15)	5.81(0.15)	5.87(0.13)	5.83(0.13)	5.54(0.11)
Am SO [40]		2.33(0.31)	2.90(0.32)	4.56(0.40)	5.36(0.47)	5.20(0.38)	6.06(0.14)	4.77(0.45)	5.75(0.26)
mPSO [29]		4.44(0.24)	3.93(0.16)	4.57(0.18)	4.97(0.13)	5.15(0.12)	5.33(0.10)	5.60(0.09)	5.78(0.09)
HmPSO [46]		4.46(0.26)	4.27(0.08)	4.61(0.07)	4.66(0.12)	4.83(0.09)	4.96(0.03)	5.14(0.08)	5.25(0.08)
APSO [47]		2.72(0.04)	2.99(0.09)	3.87(0.08)	4.13(0.06)	4.12(0.04)	4.11(0.03)	4.26(0.04)	4.21(0.02)
FTMPSO		<b>0.89(0.05)</b>	<b>1.70(0.10)</b>	<b>2.36(0.09)</b>	<b>3.01(0.12)</b>	<b>3.06(0.10)</b>	<b>3.29(0.10)</b>	<b>3.63(0.09)</b>	<b>3.74(0.09)</b>
mQSO(5,5q) [39]	2500	7.64(0.64)	3.26(0.21)	3.12(0.14)	3.58(0.13)	3.63(0.10)	3.63(0.08)	3.58(0.08)	3.30(0.06)
Am SO [40]		0.87(0.11)	2.16(0.19)	2.49(0.10)	2.73(0.11)	3.24(0.18)	3.68(0.15)	3.53(0.14)	3.07(0.12)
mPSO [29]		1.79(0.10)	2.04(0.12)	2.66(0.16)	3.07(0.11)	3.15(0.08)	3.26(0.07)	3.31(0.05)	3.36(0.05)
HmPSO [46]		1.75(0.10)	1.92(0.11)	2.39(0.16)	2.46(0.09)	2.57(0.05)	2.65(0.05)	2.72(0.04)	2.81(0.04)
APSO [47]		1.06(0.03)	1.55(0.05)	2.17(0.07)	2.51(0.05)	2.61(0.02)	2.66(0.02)	2.62(0.02)	2.64(0.01)
FTMPSO		<b>0.39(0.02)</b>	<b>0.91(0.08)</b>	<b>1.21(0.06)</b>	<b>1.66(0.05)</b>	<b>1.87(0.05)</b>	<b>2.09(0.07)</b>	<b>2.22(0.06)</b>	<b>2.22(0.07)</b>
mQSO(5,5q) [39]	10,000	1.90(0.18)	1.03(0.06)	1.10(0.07)	1.84(0.08)	2.00(0.09)	1.99(0.07)	1.85(0.05)	1.71(0.04)
Am SO [40]		0.19(0.02)	0.45(0.04)	0.76(0.06)	1.28(0.12)	1.78(0.09)	1.55(0.08)	1.89(0.14)	2.52(0.10)
mPSO [29]		0.27(0.02)	0.70(0.10)	0.97(0.04)	1.34(0.08)	1.43(0.05)	1.47(0.04)	1.50(0.03)	1.48(0.02)
HmPSO [46]		–	–	–	–	–	–	–	–
APSO [47]		0.25(0.01)	0.57(0.03)	0.82(0.02)	1.23(0.02)	1.39(0.02)	1.46(0.01)	1.38(0.01)	1.36(0.01)
FTMPSO		<b>0.09(0.00)</b>	<b>0.31(0.04)</b>	<b>0.43(0.03)</b>	<b>0.56(0.01)</b>	<b>0.69(0.09)</b>	<b>0.86(0.02)</b>	<b>1.08(0.03)</b>	<b>1.13(0.04)</b>

of 50 times execution of the proposed algorithm. The results of the proposed algorithm and other related methods are obtained until 100 environment changes. As a result, performing *change frequency* × 100 function evaluations is the Termination condition of the proposed algorithms and comparative studies.

In the following, the efficiency of the proposed algorithm, which is called finder–tracker PSO (FTMPSO), will be tested on different MPB configurations and it will be compared with the efficiency of several state-of-the-art algorithms.

**Table 11** illustrates the comparison between the efficiency of the proposed algorithm and that of five other algorithms with different peak numbers and various change frequencies. Other MPB parameters are configured as it was shown in **Table 1**. In the following, the results of mQSO [39] and AmQSO [40] algorithms are obtained from their execution, and the results of other algorithms

are obtained from their respective papers. The *change frequency* of 5000 has been more widely used in different papers, and the results of the proposed algorithm FTMPSO and eighteen other algorithms on MPB are illustrated in **Table 12** with a *change frequency* of 5000 and different peak numbers.

As the results in **Tables 11** and **12** show, algorithms performances decline by increasing the number of peaks. However, in some algorithms the results improve when the number of peaks is remarkable, because the difference between local and global fitness values decreases. Furthermore, increasing the frequency change leads to more chance for the algorithms to find better values and discover the peaks, which results in their improved efficiency. In effect, when the *change frequency* is low, it will be possible to lose the optimums or to never reach them because of rapid environment changes, which reduces the algorithms efficiency. The results

**Table 12**Comparison of offline error [45] (standard error) of 19 algorithms on MPB problem with different number of peaks with *change frequency* = 5000 and Shift length = 1.

Algorithm	Number of peaks							
	1	5	10	20	30	50	100	200
mQSO(5,5q) [39]	4.92(0.21)	1.82(0.08)	1.85(0.08)	2.48(0.09)	2.51(0.10)	2.53(0.08)	2.35(0.06)	2.24(0.05)
Am SO [40]	0.51(0.04)	1.01(0.09)	1.51(0.10)	2.00(0.15)	2.19(0.17)	2.43(0.13)	2.68(0.12)	2.62(0.10)
CLPSO [12]	2.55(0.12)	1.68(0.11)	1.78(0.05)	2.61(0.07)	2.93(0.08)	3.26(0.08)	3.41(0.07)	3.40(0.06)
FMSO [30]	3.44(0.11)	2.94(0.07)	3.11(0.06)	3.36(0.06)	3.28(0.05)	3.22(0.05)	3.06(0.04)	2.84(0.03)
RPSO [8]	0.56(0.04)	12.22(0.76)	12.98(0.48)	12.79(0.54)	12.35(0.62)	11.34(0.29)	9.73(0.28)	8.90(0.19)
mCPSO [39]	4.93(0.17)	2.07(0.08)	2.08(0.07)	2.64(0.07)	2.63(0.08)	2.65(0.06)	2.49(0.04)	2.44(0.04)
SPSO [36]	2.64(0.10)	2.15(0.07)	2.51(0.09)	3.21(0.07)	3.64(0.07)	3.86(0.08)	4.01(0.07)	3.82(0.05)
rSPSO [35]	1.42(0.06)	1.04(0.03)	1.50(0.08)	2.20(0.07)	2.62(0.07)	2.72(0.08)	2.93(0.06)	2.79(0.05)
mPSO [29]	0.90(0.05)	1.21(0.12)	1.61(0.12)	2.05(0.08)	2.18(0.06)	2.34(0.06)	2.32(0.04)	2.34(0.03)
HmPSO [46]	0.87(0.05)	1.18(0.04)	1.42(0.04)	1.50(0.06)	1.65(0.04)	1.66(0.02)	1.68(0.03)	1.71(0.02)
PSO-CP [41]	3.41(0.06)	–	1.31(0.06)	–	2.02(0.07)	–	2.14(0.08)	2.04(0.07)
CESO [48]	1.04(0.00)	–	1.38(0.02)	1.72(0.02)	1.24(0.01)	1.45(0.01)	<b>1.28(0.02)</b>	–
ESCA [49]	0.98(0.00)	–	1.54(0.02)	1.89(0.04)	1.52(0.02)	1.67(0.02)	1.61(0.01)	–
RVDEA [7]	1.02(–)	–	3.54(–)	3.87(–)	3.92(–)	3.78(–)	3.37(–)	3.54(–)
SFA [50]	0.42(0.07)	0.89(0.09)	1.05(0.04)	1.48(0.05)	1.56(0.06)	1.87(0.05)	2.01(0.04)	1.99(0.06)
APSO [47]	0.53(0.01)	1.05(0.06)	1.31(0.03)	1.69(0.05)	1.78(0.02)	1.95(0.02)	1.95(0.01)	1.90(0.01)
CLDE [51]	1.53(0.07)	1.50(0.04)	1.64(0.03)	2.46(0.05)	2.62(0.05)	2.75(0.05)	2.73(0.03)	2.61(0.02)
DynPopDE [59]	–	1.03(0.13)	1.39(0.07)	–	–	2.10(0.06)	2.34(0.05)	2.44(0.05)
DMAFSA [58]	0.55(0.06)	0.78(0.06)	1.01(0.05)	1.42(0.06)	1.63(0.06)	1.84(0.07)	1.95(0.05)	1.99(0.04)
FTMPSO	<b>0.18(0.01)</b>	<b>0.47(0.05)</b>	<b>0.67(0.04)</b>	<b>0.93(0.04)</b>	<b>1.14(0.04)</b>	<b>1.32(0.04)</b>	1.61(0.03)	<b>1.67(0.03)</b>

**Table 13**

Comparison of offline error [38] (standard error) of 5 algorithms on MPB problem with different number of peaks with change frequency of 5000 and Shift length of 1.

Algorithm	Number of peaks							
	1	5	10	20	30	50	100	200
CPSO [38]	2.29e-04 (1.04e-04)	0.36(0.15)	0.71(0.10)	1.18(0.09)	1.34(0.07)	1.42(0.07)	1.09(0.03)	0.95(0.04)
CGAR [52]	2.02(0.05)	2.56(0.10)	2.60(0.13)	3.66(0.14)	3.12(0.10)	3.26(0.11)	2.68(0.07)	2.39(0.07)
CDER [52]	0.90(0.20)	8.02(0.34)	5.52(0.16)	7.49(0.27)	5.51(0.12)	5.79(0.15)	4.12(0.10)	3.71(0.11)
CPSOR [52]	0.03(0.00)	0.54(0.04)	0.59(0.04)	0.79(0.05)	1.05(0.06)	0.98(0.05)	1.06(0.04)	0.94(0.04)
FTMPSO	<b>3.48e-05 (9.01e-06)</b>	<b>0.20(0.10)</b>	<b>0.25(0.05)</b>	<b>0.42(0.05)</b>	<b>0.48(0.04)</b>	<b>0.61(0.03)</b>	<b>0.83(0.04)</b>	<b>0.91(0.06)</b>

in Tables 11 and 12 show that FTMPSO involves a very good efficiency in different environments using different peak and different high and low frequency changes, i.e. its robustness against their changes.

As it is evident in the results presented in Table 2, decreasing the values of *Conv\_limit* leads to degrading the efficiency of the proposed algorithm. The reason for such deterioration in efficiency is that the *finder* convergence condition will not be met in an appropriate time, and it will take more time for the *finder* swarm to place a *tracker* swarm in its position and continue its search for finding other peaks. Thus, the rate of finding various peaks by the *finder* is reduced. As a result, it will take more time for all the peaks to be fully covered and the efficiency of the algorithm decreases. By increasing the value of *Conv\_limit* up to 1, the algorithm efficiency increases as well, but increasing its value beyond 1 will decrease efficiency. In fact, a high value of *Conv\_limit* results in a quick convergence for the *finder* and this problem activates those *tracker* swarms in the problem space whose existence is not useful. In other words, it will be possible for a *finder* to meet its convergence condition before approaching a peak, and as a result it activates a *tracker* swarm. So there will be more active *tracker* swarms in the problem space than what is required, and these swarms will perform too many useless fitness evaluations before they are deactivated by *trackers\_exclusion()* function.

In [28] and [51], *Offline error* has been calculated in a different approach. In these papers, instead of using the mean value of *current error* in every fitness evaluation for *Offline error* calculation, the mean value of *current error* at the end of each environment has been used. However, in these researches, *Offline error* is considered as the mean value of the best results that the algorithm has obtained in every environment. In other word, it is the mean value of *current error* at the final fitness evaluation before each environment change. In Table 13, the obtained *Offline error* results from the proposed algorithm (FTMPSO) are calculated, and the results are compared with those from 4 other algorithms in which *Offline error* is also calculated using the same approach. As the results in the table show, the efficiency of the algorithm using this method, in term of *Offline error*, outperforms the other 4 algorithms.

**Table 14**

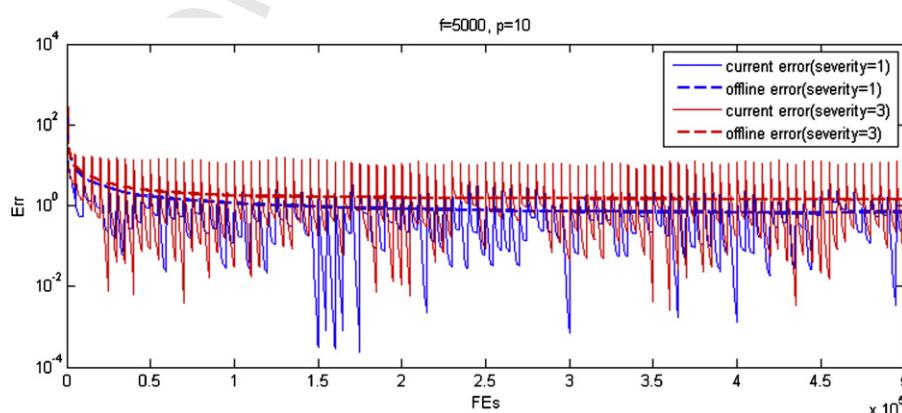
Comparison of offline error [45] (standard error) of 10 algorithms on MPB problem with different Shift length, change frequency = 5000 and 10 peaks.

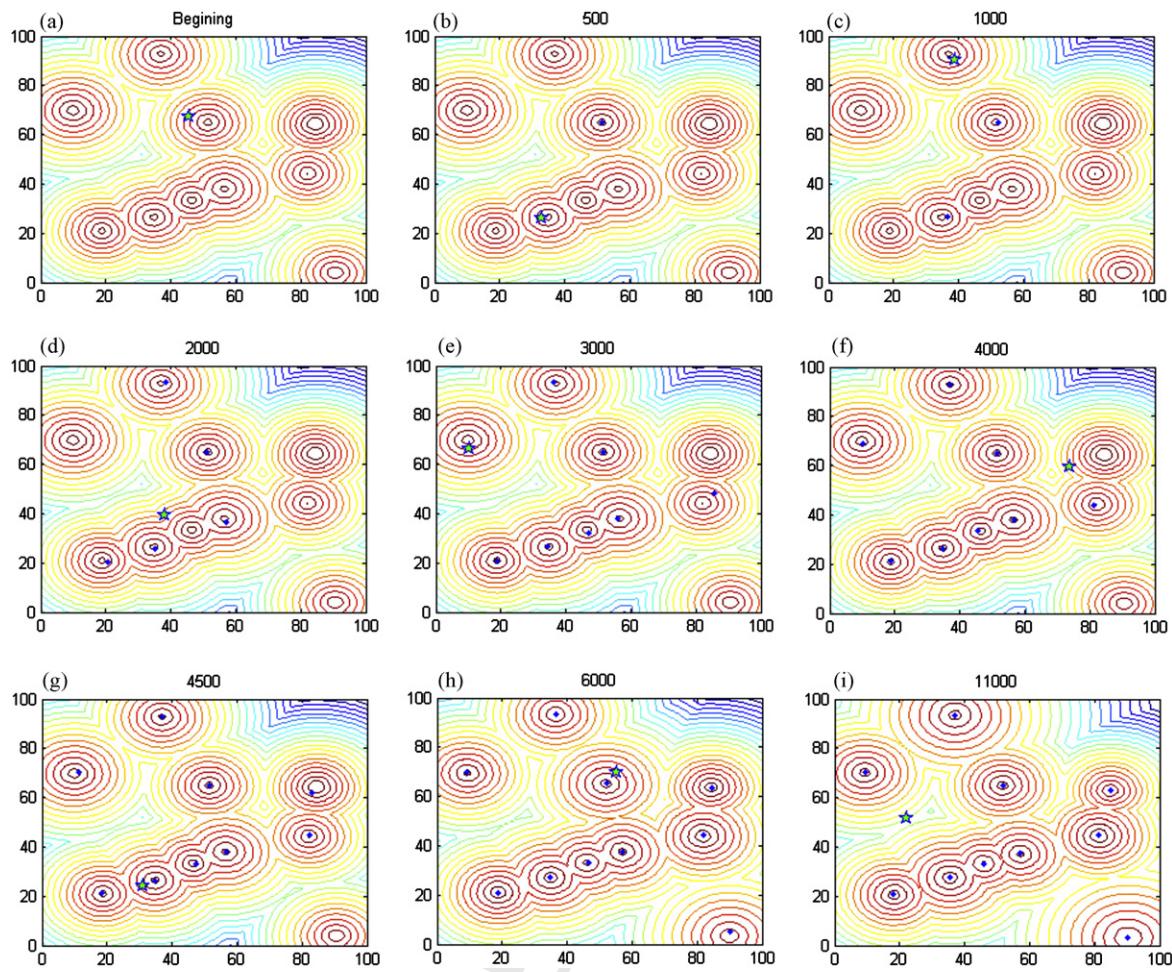
Algorithm	Shift length			
	1	2	3	5
mQSO(5.5q) [39]	1.85(0.08)	2.40(0.06)	3.00(0.06)	4.24(0.10)
AmQSO [40]	1.51(0.10)	2.09(0.08)	2.72(0.09)	3.71(0.11)
mCPSO [39]	2.08(0.07)	2.80(0.07)	3.57(0.08)	4.89(0.11)
SPSO [36]	2.51(0.09)	3.78(0.09)	4.96(0.12)	6.76(0.15)
rSPSO [35]	1.50(0.08)	1.87(0.05)	2.40(0.08)	3.25(0.09)
PSO-CP [41]	1.31(0.06)	1.98(0.06)	2.21(0.06)	3.20(0.13)
CESO [48]	1.38(0.02)	1.78(0.02)	2.03(0.03)	2.52(0.06)
ESCA [49]	1.54(0.02)	1.57(0.01)	1.67(0.01)	1.78(0.06)
SFA [50]	1.05(0.04)	1.44(0.06)	2.06(0.07)	2.89(0.13)
FTMPSO	<b>0.67(0.04)</b>	<b>1.20(0.06)</b>	<b>1.40(0.09)</b>	<b>1.69(0.07)</b>

In Table 14, the efficiency of the proposed algorithm on MPB with different *Shift lengths* has been illustrated and has been compared with 9 other algorithms. MPB configuration has been done according to Table 1, and only the value of *Shift length* parameter has changed in it. By increasing *Shift length*, it becomes more difficult for algorithms to follow the peaks, because after each environment change, the peaks move into farther distances.

As the results in Table 14 illustrate, the efficiency of all the algorithms dramatically decreases by increasing *Shift length*, but the efficiency of the proposed algorithm FTMPSO shows less degradation as a result of *Shift length* increment, and shows more robustness in these conditions, compared to other algorithms.

Fig. 11 shows the chart for *Offline error* and *current error* of the proposed algorithm during 100 environment changes in MPB with 10 peaks, the environment *change frequency* of 5000, and *Shift lengths* of 1 and 3. Other MPB parameters have been configured according to Table 1 and the amounts of the chart are calculated using average of 50 executions. As observed in Fig. 11, the degradation in the proposed algorithm efficiency on MPB with higher *Shift lengths* is the result of a considerable amount of relocations of the peaks after each environment change. In these conditions, the value of *current error* increases more as a result of *Shift length* increment

Fig. 11. Graph related to *offline error* and *current error* in the proposed algorithm in MPB with *Shift lengths* 1 and 3.

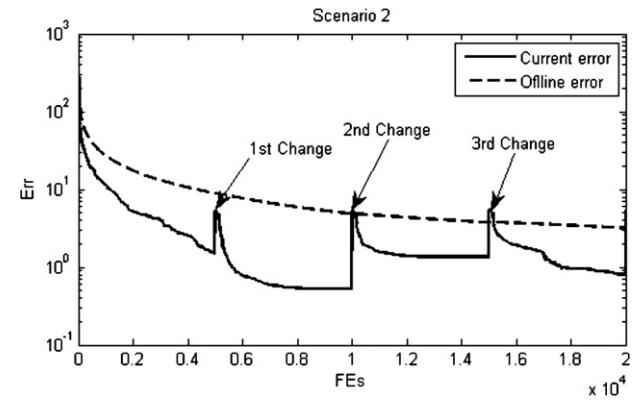


**Fig. 12.** Discovering peaks in Scenario 2 of MPB by the *finder* swarm and covering them by the *tracker* swarms.

after each environment change, and therefore the efficiency of the algorithm declines.

As the results of the experiments show, the efficiency of the proposed method is very high because of utilizing different mechanisms for improving its performance and eliminating the challenges in dynamic environments. The peaks are rapidly covered by *tracker* swarms in the proposed algorithm, and therefore the algorithm can quickly discover the position of the global optimum after the environment change. Fig. 12 shows the behavior of the proposed algorithm in two dimensions of Scenario 2 of MPB. In this figure, *Gbest* related to the *finder* swarm is marked by a star, and *Gbest* related to *tracker* swarms in the problem space is shown by a blue point. This figure shows the *finder* swarm speed in discovering the peaks. As could be seen in the figure, 9 peaks have been discovered by the *finder* swarm by 4500 fitness evaluations, and they have been covered by *tracker* swarms. Finally in the figure related to the fitness evaluation number of 6000, which belongs to the environment after the first change, all the peaks have been covered by the *tracker* swarms. It can be observed that all the peaks are covered by *tracker* swarms and these swarms have successfully followed their peaks.

One of the issues leading to improvement in *Offline error* in optimization algorithms is the values of *current error* at the beginning of the optimization process, particularly in the first environment. One of the reasons for the high efficiency of the proposed algorithm is its high speed in reducing the value of the *current error* at the beginning of its execution. Fig. 13 illustrates the values of *Offline error* and *current error* in the proposed algorithm on Scenario 2 of MPB



**Fig. 13.** *Offline error* and *current error* in the proposed algorithm for Scenario 2 of MPB in the first four environments.

in the first four environments (i.e. three environment changes). As it can be observed in this figure, the proposed algorithm is able to significantly reduce *current error*, even in the first environment.

## 5. Conclusion

In this paper, a novel optimization algorithm based on particle swarm optimization approach was proposed in dynamic environments, in which several mechanisms were employed to conquer challenges and necessities of dynamic environments. In

the proposed algorithms, the swarms in the problem space were divided into two categories: *finder* and *tracker*. Finder category was configured in order to find the peaks in an appropriate time. On the other hand, tracker category was configured for appropriately performing local optimization process on the covered peaks and subsequently tracking them after an environment change. For increasing diversity of the swarms in the proposed algorithm, a novel method based on change in velocity vector and particle positions was presented and in order to improve the efficiency of the algorithm, a local search based on adaptive exploiter particle around the best found position was suggested. Finally, a novel *awakening–sleeping* mechanism has been presented to focus the entire load of proposed algorithm on the global optimum peak.

The efficiency of the proposed method were evaluated utilizing more than 40 different configurations of Moving Peak Benchmark which is the most well-known benchmark in this domain and results have been compared with those of the 20 state-of-the art methods. The results show the superiority of the proposed methods in terms of efficiency and accuracy.

It is worth to mentioning that the proposed algorithm has been designed for dynamic environments in which changes in problem space have been occurred in discrete intervals. However, there are several real-world applications in which changes are continuously occurred. In order to employ the proposed algorithm in such environments, the structure of the algorithm must be changed. This issue could be pursued in the future works.

A primary knowledge concerning several parameters of the problem space including particularly number of peaks, *Shift length* and *change frequency* needed to be determined in almost all previous algorithms in dynamic environments. However, in the proposed algorithm, it was tried to solve dependency to the primary knowledge. Nevertheless, *Shift length* parameter must be determined to perform the proposed algorithm. By adding online learning algorithms or self-adaptive mechanisms to the proposed algorithm, the algorithm could be completely independent from the primary knowledge which will be followed in future works.

## References

- [1] K. Krishnakumar, Micro genetic algorithms for stationary and nonstationary function optimization, in: Proceedings of SPIE International Conference Adaptive Systems, 1989, pp. 289–296.
- [2] A. Carlisle, G. Dozier, Adapting particle swarm optimization to dynamic environments, in: International Conference on Artificial Intelligence, 2000.
- [3] H.G. Cobb, An investigation into the use of hypermutation as an adaptive operator in genetic algorithms having continuous, time-dependent nonstationary environments, in: NRL Memorandum Report, vol. 6760, 1990, pp. 523–529.
- [4] T. Nanayakkara, K. Watanabe, K. Izumi, Evolving in dynamic environments through adaptive chaotic mutation, in: Third International Symposium on Artificial Life and Robotics, vol. 2, 1999, pp. 520–523.
- [5] F. Vavak, K.A. Jukes, T.C. Fogarty, Performance of a genetic algorithm with variable local search range relative to frequency of the environmental changes, in: Proceedings of the Third Genetic Programming, 1998, pp. 602–608.
- [6] E.L. Yu, P.N. Suganthan, Evolutionary programming with ensemble of explicit memories for dynamic optimization, in: IEEE Congress on Evolutionary Computation, 2009. CEC '09, 2009, pp. 431–438.
- [7] Y.G. Woldesenbet, G.G. Yen, Dynamic evolutionary algorithm with variable relocation, IEEE Transactions on Evolutionary Computation 13 (2009) 500–513.
- [8] X. Hu, R.C. Eberhart, Adaptive particle swarm optimization: detection and response to dynamic systems, in: IEEE Congress on Evolutionary Computation, CEC2002, 2002, pp. 1666–1670.
- [9] S. Yang, X. Yao, Experimental study on population-based incremental learning algorithms for dynamic optimization problems, Soft Computing: A Fusion of Foundations, Methodologies and Applications 9 (2002) 815–834.
- [10] J. Kennedy, R. Mendes, Population structure and particle swarm performance, in: IEEE Congress on Evolutionary Computation, 2002, pp. 1671–1676.
- [11] S. Janson, M. Middendorf, A hierarchical particle swarm optimizer for dynamic optimization problems, in: Applications of Evolutionary Computing, 2002, pp. 513–524.
- [12] A.B. Hashemi, M.R. Meybodi, Cellular PSO: a PSO for dynamic environments, in: Advances in Computation and Intelligence, 2009, pp. 422–433.
- [13] A.B. Hashemi, M.R. Meybodi, A multi-role cellular PSO for dynamic environments, in: 14th International CSI Computer Conference, CSICC 2009, 2009, pp. 412–417.
- [14] S. Yang, On the design of diploid genetic algorithms for problem optimization in dynamic environments, in: IEEE Congress on Evolutionary Computation, 2006. CEC 2006, 2006, pp. 1362–1369.
- [15] A.S. Uyar, A.E. Harmanci, A new population based adaptive domination change mechanism for diploid genetic algorithms in dynamic environments, Soft Computing: A Fusion of Foundations, Methodologies and Applications 9 (2005) 803–814.
- [16] S. Yang, Memory-based immigrants for genetic algorithms in dynamic environments, in: Proceedings of the 2005 Conference on Genetic and Evolutionary Computation, 2005, pp. 1115–1122.
- [17] A. Simões, E. Costa, Evolutionary algorithms for dynamic environments: prediction using linear regression and Markov chains, in: Parallel Problem Solving from Nature, 2008, pp. 306–315.
- [18] H. Richter, S. Yang, Memory based on abstraction for dynamic fitness functions, in: Evo'08 Proceedings of the 2008 Conference on Applications of Evolutionary Computing, 2008, pp. 596–605.
- [19] S. Yang, Y. Xin, Population-based incremental learning with associative memory for dynamic environments, IEEE Transactions on Evolutionary Computation 12 (2008) 542–561.
- [20] H. Richter, Memory design for constrained dynamic optimization problems, in: Applications of Evolutionary Computation, 2010, pp. 552–561.
- [21] R.W. Morrison, Designing Evolutionary Algorithms for Dynamic Environments, Springer-Verlag New York Inc., 2004.
- [22] J. Grefenstette, Genetic algorithms for changing environments, Parallel Problem Solving from Nature 2 (1992) 137–144.
- [23] L.T. Bui, H.A. Abbass, J. Branke, Multiobjective optimization for dynamic environments, in: IEEE Congress on Evolutionary Computation, 2005, pp. 2349–2356.
- [24] H. Andersen, An investigation into genetic algorithms, and the relationship between speciation and the tracking of optima in dynamic functions, Honours Thesis, Queensland University of Technology, Brisbane, Australia, 1991.
- [25] F. Oppacher, M. Wineberg, The shifting balance genetic algorithm: improving the GA in a dynamic environment, in: Proceedings of the Genetic and Evolutionary Computation Conference, 1999, pp. 504–510.
- [26] J. Branke, H. Kaußler, C. Schmidt, H. Schmeck, A multi-population approach to dynamic optimization problems, in: Adaptive Computing in Design and Manufacturing, 2000.
- [27] H. Cheng, S. Yang, Multi-population genetic algorithms with immigrants scheme for dynamic shortest path routing problems in mobile ad hoc networks, in: Applications of Evolutionary Computation, 2010, pp. 562–571.
- [28] R.I. Lung, D. Dumitrescu, A new collaborative evolutionary-swarm optimization technique, in: Companion on Genetic and Evolutionary Computation GECCO, 2007, pp. 2817–2820.
- [29] M. Kamosi, A.B. Hashemi, M.R. Meybodi, A new particle swarm optimization algorithm for dynamic environments, in: Swarm, Evolutionary, and Memetic Computing, 2010, pp. 129–138.
- [30] L. Changhe, S. Yang, Fast multi-swarm optimization for dynamic optimization problems, in: 4th International Conference on Natural Computation, 2008, pp. 624–628.
- [31] R.K. Ursem, Multinational GAs: multimodal optimization techniques in dynamic environments, in: Proceedings of the Genetic and Evolutionary Computation Conference, 2000, pp. 19–26.
- [32] X. Li, J. Branke, T. Blackwell, Particle swarm with speciation and adaptation in a dynamic environment, in: Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation, 2006, pp. 51–58.
- [33] D. Parrott, X. Li, Locating and tracking multiple dynamic optima by a particle swarm model using speciation, IEEE Transactions on Evolutionary Computation 10 (2006) 440–458.
- [34] D. Parrott, X. Li, A particle swarm model for tracking multiple peaks in a dynamic environment using speciation, in: IEEE Congress on Evolutionary Computation, CEC2004, vol. 1, 2004, pp. 98–103.
- [35] S. Bird, X. Li, Using regression to improve local convergence, in: IEEE Congress on Evolutionary Computation, CEC 2007, 2007, pp. 592–599.
- [36] W. Du, B. Li, Multi-strategy ensemble particle swarm optimization for dynamic optimization, Information Sciences 178 (2008) 3096–3109.
- [37] L. Changhe, S. Yang, A clustering particle swarm optimizer for dynamic optimization, in: IEEE Congress on Evolutionary Computation, CEC '09, 2009, pp. 439–446.
- [38] S. Yang, C. Li, A clustering particle swarm optimizer for locating and tracking multiple optima in dynamic environments, IEEE Transactions on Evolutionary Computation 14 (6) (2010) 959–974.
- [39] T. Blackwell, J. Branke, Multiswarms, exclusion, and anti-convergence in dynamic environments, IEEE Transactions on Evolutionary Computation 10 (2006) 459–472.
- [40] T. Blackwell, J. Branke, X. Li, Particle swarms for dynamic optimization problems, in: Swarm Intelligence: Introduction and Applications, 2008, pp. 193–217.
- [41] L. Liu, S. Yang, D. Wang, Particle swarm optimization with composite particles in dynamic environments, IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics 40 (2010) 1634–1648.
- [42] J. Kennedy, R.C. Eberhart, Particle swarm optimization, in: Proceedings of IEEE International Conference on Neural Networks, 1995, 1995, pp. 1942–1948.
- [43] R.C. Eberhart, Y. Shi, Comparing inertia weights and constriction factors in particle swarm optimization, in: IEEE Congress on Evolutionary Computation, vol. 1, 2001, pp. 84–88.

- [44] J. Robinson, Y. Rahmat-Samii, Particle swarm optimization in electromagnetics, *IEEE Transaction on Antennas and Propagation* 52 (2004) 397–407, <http://dx.doi.org/10.1109/TAP.2004.823969>.
- [45] Available at: <http://people.aifb.kit.edu/jbr/MovPeaks>
- [46] M. Kamosi, A.B. Hashemi, M.R. Meybodi, A hibernating multi-swarm optimization algorithm for dynamic environments, in: *Proceedings of World Congress on Nature and Biologically Inspired Computing (NaBIC2010)*, Kitakyushu, Japan, 2010, pp. 370–376.
- [47] I. Rezazadeh, M.R. Meybodi, A. Naebi, Adaptive particle swarm optimization algorithm for dynamic environments, in: *Advances in Swarm Intelligence*, 2011, pp. 120–129.
- [48] R.I. Lung, D. Dumitrescu, A collaborative model for tracking optima in dynamic environments, in: *IEEE Congress on Evolutionary Computation*, 2007, pp. 564–567.
- [49] R.I. Lung, D. Dumitrescu, Evolutionary swarm cooperative optimization in dynamic environments, *Natural Computing* 9 (2010) 83–94.
- [50] B. Nasiri, M.R. Meybodi, Speciation based firefly algorithm for optimization in dynamic environments, *International Journal of Artificial Intelligence* 8 (2012) 118–132.
- [51] N. Noroozi, A.B. Hashemi, M.R. Meybodi, CellularDE: a cellular based differential evolution for dynamic optimization problems, in: *Adaptiveand Natural Computing Algorithms*, 2011, pp. 340–349.
- [52] C. Li, S. Yang, A general framework of multi-population methods with clustering in undetectable dynamic environments, *IEEE Transactions on Evolutionary Computation* (2011).
- [53] J. Branke, Memory enhanced evolutionary algorithms for changing optimization problems, in: *IEEE Congress on Evolutionary Computation*, vol. 3, 1999, pp. 1875–1882.
- [54] Y. Jin, J. Branke, Evolutionary optimization in uncertain environments – a survey, *IEEE Transactions on Evolutionary Computation* 9 (3) (2005).
- [55] L. Liu, D. Wang, J. Tang, Composite particle optimization with hyper-reflection scheme in dynamic environments, *Applied Soft Computing* 11 (8) (2011) 4626–4639.
- [56] J. Karimi, H. Nobahari, S.H. Pourtakdoust, A new hybrid approach for dynamic continuous optimization problems, *Applied Soft Computing* 11 (2011) 4626–4639.
- [57] J.M. Cadena, M.C. Garrido, E. Munoz, Facing dynamic optimization using a cooperative metaheuristic configured via fuzzy logic and SVMs, *Applied Soft Computing* 11 (8) (2011) 5639–5651.
- [58] D. Yazdani, M.R. Akbarzadeh, B. Nasiri, M.R. Meybodi, A new artificial fish swarm algorithm for dynamic optimization problems, in: *IEEE Congress on Evolutionary Computation (CEC 2012)*, 2012, pp. 1–8, <http://dx.doi.org/10.1109/CEC.2012.6256169>.
- [59] M.C. Plessis, A.P. Engelbrecht, Differential evolution for dynamic environments with unknown numbers of optima, *Journal of Global Optimization* (2012), <http://dx.doi.org/10.1007/s10898-012-9864-9>.
- [60] T. Huang, A.S. Mohan, A hybrid boundary condition for robust particle swarm optimization, *IEEE Antenna and Wireless Propagation Letters* 4 (2005) 112–117.