Operations Research Forum (2022) 3: 28
https://doi.org/10.1007/s43069-021-00116-6

ORIGINAL RESEARCH

Nature-Inspired Techniques for Dynamic Constraint Satisfaction Problems

Mahdi Bidar[1] · **Malek Mouhoub**[2]

Received: 5 June 2021 / Accepted: 14 December 2021 / Published online: 27 April 2022

Abstract

Combinatorial applications such as configuration, transportation and resource allocation often operate under highly dynamic and unpredictable environments. In this regard, one of the main challenges is to maintain a consistent solution anytime constraints are (dynamically) added. While many solvers have been developed to tackle these applications, they often work under idealized assumptions of environmental stability. In order to address limitation, we propose a methodology, relying on nature-inspired techniques, for solving constraint problems when constraints are added dynamically. The choice for nature-inspired techniques is motivated by the fact that these are iterative algorithms, capable of maintaining a set of promising solutions, at each iteration. Our methodology takes advantage of these two properties, as follows. We first solve the initial constraint problem and save the final state (and the related population) after obtaining a consistent solution. This saved context will then be used as a resume point for finding, in an incremental manner, new solutions to subsequent variants of the problem, anytime new constraints are added. More precisely, once a solution is found, we resume from the current state to search for a new one (if the old solution is no longer feasible), when new constraints are added. This can be seen as an optimization problem where we look for a new feasible solution satisfying old and new constraints, while minimizing the differences with the solution of the previous problem, in sequence. This latter objective ensures to find the least disruptive solution, as this is very important in many applications including scheduling, planning and timetabling. Following on our proposed methodology, we have developed the dynamic variant of several nature-inspired techniques to tackle dynamic constraint problems. Constraint problems are represented using the well-known constraint satisfaction problem (CSP) paradigm. Dealing with constraint additions in a dynamic environment can then be expressed as a series of static CSPs, each resulting from a change in the previous one by adding new constraints. This sequence of CSPs is called the dynamic CSP (DCSP). To assess the performance of our proposed methodology, we conducted several experiments on randomly generated DCSP instances, following the RB model. The results of the experiments are reported and discussed.

Extended author information available on the last page of the article

Keywords Constraint Satisfaction Problems (CSPs) · Nature-Inspired Techniques · Dynamic Constraints · Combinatorial Optimization

1 Introduction

1.1 Background

A constraint satisfaction problem (CSP) is a well-known framework for representing and solving discrete combinatorial problems [1]. Using the CSP paradigm, a problem under constraints is represented by a set of variables defined on finite and discrete domains, and a set of constraints restricting the values that the variables can simultaneously take. Checking for a consistent scenario to the problem will then consist of looking for a complete assignment of values to all the variables such that all the constraints are satisfied. Given that the CSP is an NP-complete problem, checking for its feasibility requires a backtrack search algorithm of exponential cost. In order to address this challenge in practice, constraint propagation has been proposed to detect inconsistencies earlier, which will save time for the backtrack search algorithm. Note that metaheuristics have also been proposed to solve the CSP by trading running time for the quality of the solution returned. Indeed, while these techniques do not guarantee to return a complete solution, they can be a good alternative, especially for hard-to-solve problems.

1.2 Problem Statement and Motivations

One of the main challenges we face when solving a CSP is when the related problem needs to be solved in a dynamic environment. In this regard, constraints might be added or removed dynamically. This can be the case of interactive applications such as configuration systems, where requirements are added or removed by the user [2–4]. Constraint restriction and relaxation can also occur due to external events, such as a faulty machine in scheduling applications or a blocked road in transportation systems [5]. In the case of over-constrained CSPs, we might need to relax some of the constraints in order to establish the feasibility of problem.

In this paper, we focus on constraint addition (or restriction). Here, we define the related problem as an optimization one where we look for a new scenario satisfying the old and new constraints, while minimizing the differences between the new scenario and the old feasible solution. This latter objective ensures to find the least disruptive solution, as this is very important in many applications as we stated earlier. In this regard, we define the dynamic CSP (DCSP) as a series of static CSPs, each resulting from a change in the previous one, as a result of adding new constraints. The goal, when solving a DCSP, is to maintain the consistency of the CSP every time constraints are added. In this regard, if a current solution needs to be updated in order to meet new constraints, then the related process should be performed in an incremental manner to reduce the time cost. In addition, in applications such as scheduling, timetabling or resource allocation, the revised solution should be as

close as possible to the old one, i.e., a new consistent configuration requiring minimum changes from the previous one. For instance, if we need to reschedule hospital personnel, for instance, we should look for the least disruptive solution (the one with minimal changes) [6]. The same should apply for flights rescheduling, as a considerable change in the assigned flights gates in an airport will have a large impact on passengers' comfort and can increase the chances for missing flights.

In the past years, several attempts have been made to address these two objectives. One of these works consists of constraint recording methods that record any constraint which can be deduced from the previous CSP (in sequence) and used in the new one [7, 8]. Local repair methods have been proposed and work through local modifications of the previous solution in order to obtain a new one [6, 7]. This basically consists in resuming the search in the neighborhood of the old solution. There is of course no guarantee in this case to find the new solution as the algorithm can be trapped in a local optimum. This challenge has been addressed through random walk strategies as well as looking for a good balance between a global and a local search through exploration and exploitation, as done in metaheuristics [9].

1.3 Our Contributions

Given the incremental nature of a DCSP, we propose a solving methodology that works as follows. After solving the initial CSP (the first one in the sequence of CSPs, forming the DCSP), we save the final state that leads to the obtention of the corresponding solution. This saved state will then be used as a resume point for finding new solutions to subsequent CSPs, any time new constraints are added. More precisely, once a solution is found for a given CSP problem, we check if this latter is consistent with the new added constraints. If this is not the case, then we resume the search from the current saved state to search for the closed solution satisfying the old and new constraints.

The above strategy requires an algorithm that works in an iterative manner, to maintain the consistency of the problem anytime new constraints are added. Metaheuristics and especially nature-inspired techniques are iterative and therefore suited to implement our iterative approach. In addition, unlike stochastic local search (SLS) (the other metaheuristics approaches) [6], nature-inspired techniques maintain a set of potential solutions. Having this set of alternatives will make the dynamic process of looking for the next solution more efficient in terms of search time. Indeed, this process is performed by a strategy that looks for a good balance between exploitation and exploration. In this regard, we first apply more exploitation in order to find the solution locally (so it will be as close as possible to the previous solution). In case we are not successful, we start conducting more exploration until a new feasible scenario is found.

Following on this proposed methodology, we have developed the dynamic variant of several nature-inspired techniques and reported the details of each in Sects. 4 to 9.

To assess the performance of these techniques, in terms of running time and quality of the obtained solutions, we conducted several experiments on DCSP instances,

randomly generated using a modified version of the known RB model [10]. The results of the experiments are reported and discussed.

This paper is a continuation of previous works we conducted to solving DCSPs, respectively, using particle swarm optimization (PSO) [11] and discrete fireflies [12]. While the focus of these two papers is on particular nature-inspired techniques, we report in this submission a general methodology that we apply to different nature-inspired techniques.

2 Related Works

Over the past three decades, several research works for tackling DCSPs have been reported. All the methods proposed either rely on iterative local search methods or a dynamic variant of systematic search techniques. In [6], the authors proposed a repair-based approach to manage dynamic constraints. This approach relies on a local search method based on iterative deepening. The latter method explores the neighborhood of the initial solution in order to find the closest one satisfying the old and the new constraints. In [13], the authors report on a backtrack search method, called nogood recording, to deal with both static and dynamic CSPs. Here, the standard backtrack search algorithm has been adapted as follows. When the algorithm reaches a solution in a leaf node that violates some of the constraints, it will be tagged as a nogood solution. Every time a nogood solution is built, it will be enhanced through different strategies like backjumping and constraint forbidding. In [8], the authors proposed an approach that reuses the solution of the initial CSP to solve the new one. The idea was motivated by the design of a scheduling system for a remote sensing satellite. The main goal here is to minimize the changes between the old and the new schedules. In [7], we investigated the applicability of systematic and approximation methods for dealing with dynamic temporal CSPs, where both numeric and symbolic temporal information are considered. The dynamic systematic search method relies on incremental constraint propagation. The latter is based on new dynamic arc and path consistency algorithms that we propose. The dynamic systematic search method has been compared to both genetic algorithms (GAs) and stochastic local search techniques. In both of these methods, when constraints are added, the neighboring areas of the previous solution are explored until a new solution is found or the maximum number of iterations is reached. Finally, in [14], a branch-and-bound method has been proposed to solve DCSPs with minimal perturbation.

While the above methods were successful in solving DCSPs with small and medium size, when the problem scales up, these techniques suffer from their exponential time complexity (in the case of exact methods) or the fact of being trapped in local optimum (in the case of iterative local search). To address this challenge, we rely on nature-inspired techniques as they are more efficient in escaping local optimum, thanks to our proposed methodology including the right balance between exploration/exploitation, in addition to the fact of maintaining a set of promising potential solutions at each iteration.

Table 1 CSP constraints: lists of ineligible tuples

(X_1, X_2)	*(1,1)*	*(1,2)*	*(2,2)*	*(3,1)*	*(4,5)*
(X_2, X_3)	*(1,1)*	*(2,2)*	*(3,4)*	*(4,4)*	*(1,3)*
(X_3, X_4)	*(1,2)*	*(5,1)*	*(3,3)*	*(3,2)*	*(4,1)*
(X_3, X_5)	*(2,2)*	*(1,1)*	*(1,5)*	*(1,3)*	*(2,4)*
(X_4, X_6)	*(1,1)*	*(1,2)*	*(1,3)*	*(2,4)*	*(2,3)*
(X_5, X_6)	*(1,1)*	*(2,2)*	*(2,3)*	*(1,4)*	*(1,5)*

3 Proposed DCSPs Solving Methodology for Nature-Inspired Techniques

3.1 CSP Definition

A CSP is a tuple (X, D, C) including a set of variables $X = \{X_1, \ldots, X_n\}$, and their related domains, $D = \{D_1, \ldots, D_n\}$, where each domain D_i contains a finite set of possible values for variable X_i. The set of constraints C restricts the values that the variables can simultaneously take. A solution to a CSP is a complete assignment of values to all the variables such that all constraints are satisfied. Note that there might be one, many, or no solution to a given CSP [1]. A binary CSP is a CSP where all the constraints have arity less than or equal 2. While we consider CSPs where constraints can be of any arity, in the illustrative examples and experiments, we assume that CSPs are binary, where each constraint is defined in extension (defined as a subset of the Cartesian product of the domains of the involved variables) and has arity 2.

Table 1 lists a set of constraints for a given CSP with six variables defined on domain $D = \{1, 2, 3, 4, 5\}$. Note that, in this example, we represent each constraint with the list of ineligible tuples between the related pairs of variables.

3.2 CSP Representation in Nature-Inspired Techniques

We represent a CSP potential solution (complete assignment of values to all the variables) with a chromosome, as shown in Table 2. While "chromosome" is a term normally used for GAs, it corresponds to particles (individuals, fireflies, bees, ⋯, etc.) in other nature-inspired techniques. In this regard, each of the nature-inspired techniques we present in the next sections will start the search with a set of (randomly generated) particles.

We define the fitness function by the number of violated constraints implied by the related complete assignment (chromosome). For instance, the fitness function for the potential solution in Table 2 is equal to 2 as it violates two constraints listed in

Table 2 Potential solution represented as a chromosome

Variables	$\mathbf{X_1}$	$\mathbf{X_2}$	$\mathbf{X_3}$	$\mathbf{X_4}$	$\mathbf{X_5}$	$\mathbf{X_6}$
Chromosome	1	3	1	2	1	1

Table 1: (X_3, X_4) and (X_5, X_6). A chromosome with fitness 0 corresponds to a consistent solution.

To measure the similarity between two solutions, we use the Hamming distance which corresponds to the number of values that both solutions do not share. The Hamming distance between two solutions S_i and S_j with n variables, $d_H(S_1, S_2)$, is calculated, as follows:

$$d_H(S_i, S_j) = \sum_{k=1}^{n} (S_{i,k} \neq S_{j,k}) \tag{1}$$

In particular, the Hamming distance between two identical solutions is equal to 0, while the Hamming distance between two solutions not having any value in common is equal to the number of variables, n.

3.3 DCSPs Solving Methodology

As stated before, a DCSP is a sequence of CSPs, $CSP_1, \ldots, CSP_i, CSP_{i+1}, \ldots, CSP_n$, where CSP_{i+1} is obtained by adding C_{new} constraints to CSP_i. More precisely, the set of constraints for CSP_{i+1}, $C_{i+1} = C_i \bigcup C_{new}$ where C_i is the set of constraints of CSP_i. To solve a DCSP using a given nature-inspired technique, we propose a methodology that relies on a good balance between exploration and exploitation, to maintain a consistent solution for each CSP_i, with minimum perturbation. This is achieved using the following two steps.

1. Solve the initial CSP, CSP_1 and save the last population that leads to a consistent solution, S_1. In addition to S_1, the population contains other individuals (elite particles) that can be used in case of a constraint restriction.
2. Solve the sequence of CSPs as follows. After solving CSP_i, when adding a set of new constraints C_{new} to CSP_i, we first check if the solution to CSP_i is still feasible for CSP_{i+1}. If this is not the case, then we resume from the last population that lead to the solution of C_i and start searching locally, through exploitation, for a new solution satisfying C_{i+1} with minimum perturbation (the new solution should be as close as possible to the solution of CSP_i). In this regard, two fitness functions will be used: number of violated constraints (to ensure consistency) and similarity (enforced using the Hamming distance). We will rely here on elite particles that we saved from the previous state. If no new solution can be obtained, then we will start gradually performing more exploration until we get one satisfying the old and new constraints (C_{i+1}), while meeting the distance objective.

In the following sections, we will describe how the above methodology applied in the case of each nature-inspired technique we consider.

4 Self-Adaptive Discrete Firefly Algorithm (SADFA)

The firefly algorithm (FA) has been developed by [15] and is based on idealized behavior of fireflies flashing characteristics, according to the following three governing rules.

Gender: Fireflies are unisex meaning that they can be attracted to each other regardless of their gender.
Attractiveness: Each firefly is only attracted to the fireflies that are brighter than itself. The attractiveness is proportional to the brightness which attenuates over distance. The brightest firefly moves randomly.
Fitness function: The brightness of a firefly represents its solution quality and is calculated with a fitness function that is defined according to the problem being solved.

In order to tackle constraint satisfaction problems (CSPs), we developed the discrete version of FA [16] that we call discrete FA (DFA). In this regard, we redefined the distance, attractiveness, fireflies movement and the fitness function in order to consider discrete spaces. More precisely, we consider the Hamming distance (described earlier) and define the attractiveness parameter β as the probability of replacing the values assigned to some variables with the corresponding values of the better solution (according to the fitness function). β is here the parameter controlling the convergence rate of fireflies. Therefore, larger values of β will cause a faster convergence of the fireflies, often to a local optimum solution. In order to balance exploitation and exploration, the following solution diversification mutation operators are used:

1. *Random Resetting Mutation (RRM): RRM* assigns random values to randomly chosen variables and is an appropriate method for problem spaces consisting of lists or strings of arbitrary elements like integer values. The main advantage of *RRM* is that every point in the problem space can be reached from any arbitrary solution in the problem space, which will increase diversity.
2. *Scramble Mutation (SM):* Here, a subset of contiguous variables are selected and their values are shuffled or scrambled randomly (assuming all variables have the same domain).
3. *Inversion Mutation (IM):* In this method, a randomly chosen sequence of values (corresponding to a partial assignment) is reversed end to end.

Table 3 shows how RRM, SM and IM mutations are, respectively, applied to the potential solution in Table 2.

In [17], we have proposed a dynamic version of DFA in order to get a new solution for a DCSP with minimal perturbation. The dynamic variant of DFA is called self-adaptive discrete firefly algorithm (SADFA) and works, in an incremental way, following our general methodology we presented previously. More precisely, when

Table 3 A chromosome representing a potential solution

Variables	$\mathbf{X}_1$	$\mathbf{X}_2$	$\mathbf{X}_3$	$\mathbf{X}_4$	$\mathbf{X}_5$	$\mathbf{X}_6$
Chromosome (S)	1	3	1	2	1	1
RRM (S)	1	5	1	3	1	1
SM (S)	3	1	2	1	1	1
IM (S)	1	2	1	3	1	1

new constraints are added to a given CSP, we first check if the current solution is still feasible. If it is not the case, then the β movement will search locally for the new solution that is the closest possible to the current one. The key point here is to use elite fireflies, i.e., those good fireflies that have been already discovered by the algorithm when looking for the current solution. These correspond to potential solutions satisfying most of the constraints. In order to prevent the algorithm from being trapped in a local optimum, we control its progress trend and apply diversification anytime this situation is detected. This is performed by watching both the similarity and the number of violated constraints during the search. In this regard, we use two controlling parameters that we call *CPQ* and *CPS* to detect if SADFA has been trapped in a local optimum. Using Eqs. 2 and 3, these two parameters monitor the progress of the algorithm over a given number of iterations (corresponding to the window size). If there is not enough progress, then the diversification rate (*dr*) of the algorithm will be increased.

$$CPQ(IN) = \frac{\sum_{i=IN-WS+1}^{IN} |(GBQ(i) - GBQ(i-1))|}{WS} \tag{2}$$

$$CPS(IN) = \frac{\sum_{i=IN-WS+1}^{IN} |(GBS(i) - GBS(i-1))|}{WS} \tag{3}$$

In the above equations, *IN* is the current iteration number, *WS* is the window size, *GBQ(i)* is the quality of the global best solution at iteration *i*. *GBS* is the similarity between the global best solution at iteration *i* and the previous best solution. Here, the window size determines the number of iterations to be considered to determine if an acceptable progress has been made by the algorithm. If *CPQ* and *CPS* are less than the user-defined threshold values, *dr* is increased. Algorithm 1 lists the pseudocode of our diversification adjustment procedure according to the progress of the search algorithm.

StepSize, a number in [0,1), is the increasing or decreasing diversification rate of the algorithm. In our experiments, we consider *StepSize* = 0.05. *Diversification.Flag = 1* indicates that *dr* is changing. When the progress of the algorithm is weak, the procedure (as shown in Algorithm 1) starts to increase *dr* of the algorithm to help it escape the local optimum by producing diverse solutions. After escaping this trap, *dr* should return to normal to emphasize on the local search in order to maintain the similarity between the producing solutions and the previous solution found so far. Algorithm 2 lists the pseudocode of SADFA. According to the progress rate of the algorithm, the

values of the controlling parameter of the mutation rate will change dynamically and this will help the algorithm to escape the local optimum traps. The potential solutions achieved by the elite fireflies, we mentioned earlier, have high degree of similarity to the best solution, and the key idea is to search the neighboring areas of those solutions, unless a higher rate of diversification is required.

Algorithm 1 Diversification Adjustment Procedure [8]

```
procedure DIVERSIFY (IN)
    Compute CPQ(IN) and CPS(IN) as shown in Equations 2 and 3
    if CPQ(IN) < threshold_1 || CPS(IN) < threshold_2 then
        if Diversification.Rate<0.4 then
            Diversification.Rate ← Diversification.Rate + StepSize
            Diversification.Flag ← 1
        end if
    else
        if Diversification.Flag==1 then
            Diversification.Rate ← Diversification.Rate – StepSize
            if Diversification.Rate==0.1 then
                Diversification.Flag ← 0
            end if
        end if
    end if
end procedure
```

Since both fitness functions (number of constraints violations and similarity) are equally important and not conflicting, we consider the same weight for both when deciding on the diversification rate. As we can see in Algorithm 2, at each iteration, the global best is updated according to these two fitness functions.

5 Discrete Particle Swarm Optimization (DPSO)

Particle swarm optimization (PSO) is a nature-inspired swarm-based optimization algorithms following the collective intelligent behavior of systems like fish schooling and birds flocking [18, 19]. In PSO, every particle position is influenced by its own best position, *pbest*, as well as the best position made by other particles so far, *gbest*. Every particle position represents a potential solution for the given optimization problem and a set of particles form a swarm which moves throughout the problem spaces. The position of particle i is represented by a vector $X_i = (x_{i1}, \ldots, x_{id})$. Particle i moves according to its velocity defined as a vector $V_i = (v_{i1}, \ldots, v_{id})$. At each time step, velocities and positions of particles are updated according to Eqs. 4 and 5:

$$V_i^{t+1} = w \times V_i^t + c_1 r_1 \times (pbest_i^t - X_i^t) + c_2 r_2 \times (gbest_i^t - X_i^t) \tag{4}$$

$$X_i^{t+1} = X_i^t + V_i^{t+1},\ i = 1, \ldots, n \tag{5}$$

Here, n is the number of particles, w is the inertial weight, V_i^t is the velocity of particle i at iteration (time step) t, c_1, c_2 are acceleration coefficients, $r_1,\ r_2$ are random

values in [0,1], $pbest_i^t$ is particle i best achievement so far, X_i^t is the position of particle i at iteration t and $gbest_i^t$ is the best achievement of all particles so far.

To convert the continuous PSO to a discrete one (that we call discrete PSO or DPSO), we have defined the operators, $\times$, $+$ and $-$, listed in Eqs. 4 and 5. In this regard, we have proposed new operators, $\otimes$, $\oplus$ and $\ominus$, as shown in the new equations for discrete PSOs: 6 and 7 [11].

$$V_i^{t+1} = \underbrace{w \otimes V_i^t}_{exploration} \oplus \underbrace{c_1 r_1 \otimes (pbest_i^t \ominus X_i^t) \oplus c_2 r_2 \otimes (gbest_i^t \ominus X_i^t)}_{exploitation} \tag{6}$$

$$X_i^{t+1} = X_i^t \oplus V_i^{t+1},\ i = 1, \ldots, n \tag{7}$$

Algorithm 2 SADFA [8]

```
nPop ← population size
MaxIt← maximum number of iterations
f1(): number of violated constraints
f2(): Hamming distance between two solutions
θ ← User-defined parameter
M ← mutation type (IM, SM or RRM)
for It ← 1 to MaxIt do
    for i ← 1 to nPop do
        for j ← i+1 to nPop do
            if f1(firefly_j) < f1(firefly_i) && f2(firefly_j) < f2(firefly_i) then
                %%%%%%% Determining dr
                dr ← Diversify(i) %% See Algorithm 1
                %%% β movement
                d ← distance(firefly_i, firefly_j)
                β ← β0 × (−γ × d^m)

                for k ← 1 to n do
                    if firefly_i(k) ≠ firefly_j(k) then
                        b ← a random number in (0,1)
                        if b > β then
                            firefly_i(k) ← firefly_j(k)
                        end if
                    end if
                end for
                %%% Influence the similarity between current solutions and the previous CSP's Solution
                Dis ← distance(firefly_i, PreviousSolution)
                for k ← 1 to n do
                    if firefly_i(k) ≠ PreviousSolution(k) then
                        b ← random number in (0,1)
                        if b > θ then
                            firefly_i(k) ← PreviousSolution(k)
                        end if
                    end if
                end for
        %%%Mutation with dynamic rate
                firefly_i ← Mutation(M, firefly_i, dr)
                %%% Update the Best solution
                if f1(firefly_i) < f1(Bestsolution)&& f2(firefly_i) < f2(Bestsolution) then
                    Bestsolution ← firefly_i
                end if
            end if
        end for
    end for
end for
```

Table 4 w effect on V_i^t

V^{t_i}	4	3	2	5	1	3	4	5	5	4
$w = 0.4$				↓		↓	↓			↓
$w \otimes V^{t_i}$	2	1	2	5	3	3	4	2	4	4

As stated in Eq. 6, exploitation and exploration strategies are controlled by three controlling parameters, w, c_1 and c_2. A high value of w encourages exploration, while a low value encourages exploitation of the algorithm [20]. Similarly, low values of c_1 and c_2 allow the particles to stray from the promising areas trying to discover more quality solutions, while high values of c_1 and c_2 result in moving toward *pbest* and *gbest*. In DPSO, and using the operator $\otimes$, w determines the percentage of the variable values that will be passed from V^{t_i} to V_i^{t+1}. The rest of V_i^{t+1} values will be generated randomly. Table 4 reflects this process in a concrete example.

Similarly, $c_1 r_1$ and $c_2 r_2$ correspond to the percentage of values to keep from $(pbest_i^t \ominus X_i^t)$ and $(gbest_i^t \ominus X_i^t)$, respectively. This operation is again enforced through $\otimes$. The operator $\ominus$ allows the selection of the values that are in $pbest_i^t$ (respectively $gbest_i^t$) and not in X_i^t ($\ominus$ can be seen as a set difference operation). Table 5 reflects this process on a concrete example. The new position of particle i, X_i^{t+1}, is computed according to Eq. 7.

To solve a DCSP, we follow our general methodology presented in the Introduction section and rely on a good balance between exploration and exploitation with minimum perturbation, as described in [11].

Algorithm 3 DPSO Algorithm [6]

$nPop \leftarrow Population\ size$
$MaxIt \leftarrow Maximum\ number\ of\ iterations$
$f_1() \leftarrow Number\ of\ violated\ constraints$
$f_2() \leftarrow Hamming\ distance\ between\ two\ solutions$
$PS \leftarrow Solution\ of\ Previous\ CSP$
Set $c_1,\ c_2,\ r_1,\ r_2, pr_1, pr_2, pr_3$
for It ← 1 to MaxIt **do**
  for i ← 1 to nPop **do**
    $V_i^{It} \leftarrow w \otimes V_i^{It-1} \oplus c_1 r_1 \otimes (pbest_i^{It-1} \ominus X_i^{It-1}) \oplus c_2 r_2 \otimes (gbest_i^{It-1} \ominus X_i^{It-1})$
    $X_i^{It} = X_i^{It-1} \oplus V_i^{It}$
    if $f_1(X_i^{It}) < f_1(pbest_i^{It-1})$ && $f_2(X_i^{It}, X_i^{It-1}) < f_2(pbest_i^{It-1}, X_i^{It-1})$ **then**
      $pbest_i^{It} \leftarrow X_i^{It}$
      if $f_1(pbest_i^{It}) < f_1(gbest_i^{It-1})$ && $f_2(pbest_i^{It}, X_i^{It-1}) < f_2(gbest_i^{It-1}, X_i^{It-1})$ **then**
        $gbest_i^{It} \leftarrow pbest_i^{It}$
      end if
    end if
  end for
end for

Table 5 $c_1 r_1$ effect on X_i^t

$pbest_i^t$	4	3	2	5	1	3	4	5	5	4
X_i^t	2	3	1	5	1	4	2	3	5	2
$pbest_i^t \ominus X_i^t$	4		2			3	4	5		4
$c_1 r_1 = 0.5$			↓				↓	↓		
$c_1 r_1 \otimes (pbest_i^t \ominus X_i^t)$	2	3	2	5	1	4	4	5	5	2

6 Discrete Focus Group Optimization Algorithm (DFGOA)

FGOA is a new metaheuristic algorithm proposed in [21] for optimization problems. This algorithm is inspired by the collaborative behavior of a group's members sharing their ideas on a given subject.

In [22], we have proposed a discrete version of FGOA that we call DFGOA. We define the impact factor parameter for each potential solution based on its quality as shown in Eq. 8.

$$IF^{t+1}(i) = IF^t(i) + \sum_{j=1}^{nPop} \frac{rand() \times (|F(S(i)) - F(S(j)|) \times IC(j)^m}{Nvar} \tag{8}$$

IF(i) is the impact factor of participant i which will take an important role in the next steps to affect the other participants' solutions, $IF^{t+1}(i)$ is the new impact factor of participant i, *nPop* is the population size, *Nvar* is the number of variables of the problem, *rand()* generates a random number in (0,1) and $F(S_i)$ and $F(S_j)$ are the fitness values for solutions i and j, respectively. $IC(j)$, the impact coefficient, is a random number in (0,1) and is assigned to each solution. In this regard, a set of *nPop* random numbers is generated and based on the qualities of solutions are assigned to each solution. (The higher quality a solution has, the larger the value will be assigned to.)

Table 6 illustrates the process for a minimization problem for a set of given solutions with associated qualities.

In a discrete problem space, affecting a solution can be interpreted as replacing its variables' values with the corresponding variables' values of the better solution with an appropriate probability, in order to avoid the immature convergence of the algorithm. In our proposed algorithm, this replacement is done by considering $IF()$ as the probability of this replacement. We normalize the impact factor between 0 and 1 according to (9).

$$IF(i)_{Normalized} = 1 - \frac{F(s(i)) - F(Best\ solution)}{F(Worst\ solution) - F(Best\ solution)} \tag{9}$$

Table 6 Assignment of impact coefficient $IC(i)$ to each solution for a given instance problem

	S1	S2	S3	S4	S5	S6	S7	S8
F(S(i))	21	20	18	14	10	7	8	2
Generated Random values	0.71	0.51	0.07	0.18	0.40	0.59	0.24	0.14
IC(i)	0.18	0.21	0.24	0.40	0.51	0.63	0.59	0.71

Here, $F(Best\ solution)$ and $F(Worst\ solution)$ are the expected qualities of the best and the worst solutions. In fact, the larger $IF(i)$ is, the more chance participant i (S_i) has to impact the other participant' solutions. This replacement is done according to (10).

$$Rep(S_i, S_j) = \begin{cases} S_j(k) \leftarrow S_i(k) \\ if\ S_j(k) \neq S_i(k) \\ \text{and } rnd < IF(I) \end{cases} \tag{10}$$

$Rep(S_i, S_j)$ is the replacement equation, and rnd is a random number in (0,1). Table 7 indicates the steps through which S_2 is being affected by S_1. According to this figure, the corresponding variables in two solutions with equal values remain unchanged. However, the other variables' values of S_2 are replaced with probability $IF(13) = 0.3$ by the corresponding variables' values of S_1.

At the first, second and fourth steps above, the variables' values of S_2 remained unchanged. However, in the third and fifth steps S_2 variables' values are replaced by those of S_1, resulting in $S_2 = [3\ 1\ 2\ 4\ 2\ 1]$.

In the proposed method, we use a controlling parameter called CP to detect if the FGOA has been trapped in local optima solutions. In the case that an algorithm has been trapped in local optima solutions it cannot make further improvements. This parameter through (1) monitors the progress trend of the algorithm and if for some iterations not enough progress has been made by the algorithm, this parameter enables a randomization method to diversify the solutions.

$$CP = \frac{\sum_{i=IN-WS}^{IN} (GB(i) - GB(i-1))}{WS} \tag{11}$$

IN is the current iteration number, WS is the window size, and $GB(i)$ is the global best solution in iteration i. Here, window size determines the number of iterations to be considered to determine if an acceptable progress has been made by the algorithm. If CP is less than the user-defined threshold value, the algorithm activates a new randomization method called IF Randomization (IFR).

We have employed the *IF Randomization* method for diversifying the solutions. According to this method, based on the impact factor (IF) of a solution, a variable value of a given solution is replaced with another value which is randomly chosen from its

Table 7 Steps showing how participant 2 (S_2) is affected by participant [22] 1 (S_1)

	X_1	X_2	X_3	X_4	X_5	X_6
S_1 :	1	3	2	4	1	1
S_2:	3	1	3	4	2	3
$S_1 \rightarrow S_2$						
1^{st} step →	3			4		
2^{nd} step →	3	1		4		
3^{rd} step →	3	1	2	4		
4^{th} step →	3	1	2	4	2	
5^{th} step →	3	1	2	4	2	1

domain with probability $(1 - IF)^2$ (as shown in Table 8). The probability $(1 - IF)^2$ causes higher-quality solutions to be subject to less changes in their variables values.

Algorithm 5 lists the pseudocode of the proposed DFGOA for solving DCSPs. Here, the window size and threshold value determine the sensibility of the algorithm to stagnation of the algorithm. In fact, these two parameters control the probability of applying *IF randomization* to the solutions. By initializing these parameters appropriately, the emphasis of DFOGA will be on exploitation first and then exploration, as described in our general solving methodology.

Algorithm 4 IF Diversifier

```
if CP < Threshold then
    for k←1 to Nvar do
        rnd ← rand()
        if r thennd < (1 − IF(k))^2
            Temp ← randomly choose values d ∈ D_i
            S_i(k) ← Temp
        end if
    end for
end if
```

7 Dynamic Harmony Search (DHS) Algorithm

The harmony search (HS) optimization algorithm is a population-based metaheuristic algorithm which was developed by Geem et al. in 2001 [23] based on improvisation process of jazz musicians. Improvisation process stands for the attempt of a musician to find the best harmony that can be achieved in practice [24]. Three options can be considered when a skilled musician aims at improvising on a music instrument, a) to play a memorized piece of music exactly, b) to play a piece similar to what he/she has in their memory and c) to play newly composed notes [15].

These three options were considered by the Geem as the main components of the HS algorithm which were introduced as harmony memory, pitch adjustment and random search to the algorithm [15].

The harmony memory has a valuable role in HS algorithm and that is to ensure that good harmonies are considered when generating new solutions. This component is controlled by a parameter called harmony memory considering rate, $HMCR \in [0, 1]$. In fact, this parameter determines the ratio of considering elite solutions (harmonies) in generating a new solution. If this parameter is set to a small value, the algorithm considers a small number of elite solutions; therefore, it converges to the optimal solution too slowly. On the other hand, if it set to a large value, the emphasis of the algorithm will be on using the solutions in the memory and therefore other good solutions are not explored. This does not lead to discovering better solutions.

The next component is pitch adjustment, which has the same application as the mutation operator in genetic algorithms, is stated as follows:

$$X_{new} = X_{old} + BW * \epsilon \tag{12}$$

Algorithm 5 DFGOA

$nPop \leftarrow Population\ size$
$nVar \leftarrow number\ of\ variables$
MaxIt ← Maximum number of iterations
$f_1() \leftarrow Numberof\ violated\ constraints$
$f_2() \leftarrow Hamming\ distance\ between\ two\ solutions$
$PI(i).Q \leftarrow NVC(i) + (d(i) - n)\ Quality\ of\ solution\ PI(i)$
$PS \leftarrow Solution\ of\ Previous\ CSP$
w ← $User - defined\ parameters$
for It ← 1 to MaxIt **do**
  for i ← 1 to nPop **do**
    for j ← i+1 to nPop **do**
      if $f_1(PI(j)) < f_1(PI(i))$ & $f_2(PI(j)) < f_2(PI(i))$ **then**
        Compute $Impact\ coefficient\ (IC)\ for\ members$
        $IF(i) \leftarrow IF(i) + \sum_{j=1}^{nPop} ((\frac{rand(1)\times(|S(i).Q - S(j).Q|)\times IC(j)}{Nvar})^m)$
        $IF(i)_{Normalized} = 1 - \frac{s(i).Q - BestSol.Q}{WorstSol.Q - BestSol.Q}$
        $\boldsymbol{NPI(i) \leftarrow Move}\ (PI(i),\ PI(j), IF(i)_{Normalized})$
        Evaluate $NPI(i)$
      end if
      % Update personal best
      if $f_1(NPI(i)) < f_1(PI(i))$ && $f_2(NPI(i), PS) < f_2(PI(i), PS)$ **then**
        $PI(i) \leftarrow NPI(i)$
      end if
    % Update global best
      if $f_1(PI(i)) < f_1(BestSol)$ && $f_2(PI(i), PS) < f_2(BestSol, PS)$ **then**
        $BestSol \leftarrow PI(i)$
      else$CP = \frac{\sum_{i=It-WS}^{It} (BestSol(i).Q - BestSol(i-1).Q)}{WS}$
        $NPI(i) \leftarrow IFDiversifier(PI(i))$
      end if
      if $f_1(NPI(i)) < f_1(PI(i))$ && $f_2(NPI(i), PS) < f_2(PI(i), PS)$ **then**
        $PI(i) \leftarrow NPI(i)$
      end if
      if $f_1(PI(i)) < f_1(BestSol)$ && $f_2(PI(i), PS) < f_2(BestSol, PS)$ **then**
        $BestSol \leftarrow PI(i)$
      end if
    % Randomization
      $Sort(PI)$
      $WorstSol \leftarrow PI(nPop)$
    end for
  end for
end for

Function Sort(PI)
for $i \leftarrow 1\ to\ nPop$ **do**
  % NVC ← number of violated constraints by solution i
  $PI(i).Q \leftarrow NVC(i) + (d(i) - n)$
end for
% Sort the solutions based on the calculated Quality (Q) in the previous step
$PI \leftarrow sort(PI.Q)$

Function $Move\ (PI(i),\ PI(j), IF(i))$
for $k \leftarrow 1\ to\ nVar$ **do**
  $rnd \leftarrow Rand(0, 1)$
  if $PI(j,k) \neq PI(i,k)$ and $rnd < IF(i)$ **then**
    $PI(i,k) \leftarrow PI(j,k)$
  end if
end for

Table 8 Process of diversification of a solution considering probability $(1 - IF)^2$

	$\mathbf{X_1}$	$\mathbf{X_2}$	$\mathbf{X_3}$	$\mathbf{X_4}$	$\mathbf{X_5}$	$\mathbf{X_6}$
S_1	1	3	2	4	1	1
Select variables with probability $(1 - IF)^2$:	?	↓	↓	?	?	↓
Choose new values for selected variables:	4	3	2	2	3	1

Here, X_{old} is the solution (pitch) in the memory, BW is the bandwidth, ϵ is a random value in (0,1) and X_{new} is the new solution. This component generates solutions slightly different from those in the memory by adding small random values to the solutions in memory. The degree of pitch adjustment can be controlled by pitch adjustment rate parameter PAR. The low value of PAR together with the small value of BW can reduce the exploration which results in discovering a portion of the problem space instead of the whole problem space.

The third component of the HS algorithm is randomization. The main role of this component is to encourage the diversity of the solutions. Randomization ensures that all regions of the problem space are accessible by the algorithm.

7.1 Harmony Search Algorithm for CSPs

The HS algorithm was developed to tackle continuous optimization problems. In order to deal with CSPs which is a discrete problem, HS features must be changed to suit the discrete problem spaces. The steps of converting the HS algorithm to its discrete version are discussed below. At the initialization step, the algorithm randomly generates HMS (harmony search size) solutions as the initial population. In this step, potential solutions are generated by assigning random values (from the variables domains) to CSP variables.

The next step is to redefine the pitch adjustment equation presented in Eq. 12. The new definition of the pitch adjustment is presented in Eq. 13:

$$X_{new} = (BW \odot \epsilon) \otimes X_{old} \tag{13}$$

Through pitch adjustment, slight changes by adding small random values are made in the current solution in order to improve it. The new definition of the pitch adjustment component has the same impact on the current solutions. Through the new definition, the variables' values of the current solution will be replaced by randomly picking values from variables' domain in the hope to improve the current solution. In Eq. 13, we defined $\odot$ and $\otimes$ as follows. $\odot$ is the multiplier and $BW \odot \epsilon$ is a probability value in (0,1). In fact, this latter term determines the probability of replacing the variables' values of X_{old} with new values picked up randomly from variables domain. $\otimes$ applies these changes to X_{old}. Figure 1 shows this operation through an example. Here, $BW \odot \epsilon$ is equal to 0.6 which corresponds to 60% replacement of values.

The last component is randomization that encourages diversity of the solutions. This will ensure that a larger search space will be considered by the algorithm. To boost the diversity of the solutions, we employ the GA mutation operator. In this regard, different mutation operators have been developed so far, including the following three that we presented in Section 4: Random resetting mutation (RRM), scramble mutation (SM) and inversion mutation (IM).

7.2 DHS for DCSPs

Our solving method consists of taking advantage of the exploitation and exploration features of the HS algorithm. These two features are controlled by *PAR* and *HMCR*. High values of *HMCR* and *PAR* encourage exploration, and low values will favor exploitation [15]. By initializing these parameters appropriately, we will get a suitable strategy (with minimum perturbation) for the HS algorithm to efficiently solve DCSPs. In this regard, we consider our general solving methodology.

Since both objectives are equally important and not conflicting, we consider the same weight for both when deciding on the diversification rate. Algorithm 6 presents the pseudocode of our proposed DHS method. The algorithm first starts with the generation of *HMS* potential solutions (harmonies) and stores them in a harmony memory (HM). These harmonies are then evaluated (according to their number of violated constraints and their distance to the old solution). The global best solution is then identified. The algorithm then performs a series of iterations where new harmonies are generated and altered (following the exploration and exploitation strategies). The global best and the set of harmonies are then updated. The

X_{old} :	1	3	3	3	1	2	4	5	3	5
$BW \odot \epsilon = 0.6$:		↓			↓	↓		↓	↓	↓
$(BW \odot \epsilon) \otimes X_{old}$:	1	1	3	3	5	1	4	2	2	4

Fig. 1 Application of $(BW \odot \epsilon) \otimes X_{old}$ to a given individual

most challenging issue here is to sort the potential solutions since their qualities are assessed based on the two objectives. To do so, we propose an aggregation of the number of violated constraints and the distance to the best solution that can be achieved.

8 Bee Colony for DCSPs

8.1 Artificial Bee Colony Algorithm

The artificial bee colony (ABC) optimization algorithm, proposed by Karaboga [25], is a population-based optimization algorithm which simulates the foraging behavior of real bee colonies in the nature. Agents of the ABC algorithm are divided into three class of bees, recruited bees, onlooker bees and scout bees, and each class of bees shoulders responsibilities [26]. Recruited bees search the food sources around the location they have in their memories and keep the onlooker bees updated about the quality of the food sources they are visiting. Onlooker bees based on the information they are receiving from recruited bees select the new food sources (the higher-quality ones) and also search around the selected food sources in the hope of discovering new and more quality food sources. Scout bees are recruited bees that abandoned their food source in order to discover new food sources [25].

In the initialization step, SN number of food sources are generated according to Eq. (14):

$$x_{i,j} = x_{min,j} + Rand(0,1)(x_{max,j} - x_{min,j}) \tag{14}$$

where X_i is the food source i, $X_{min,j}$ and $X_{max,j}$ are lower and upper bounds of the dimension j ($j = 1, \ldots, n$). Each food source is assigned to a recruited bee, and it then generates a new food source in its neighborhood using Eq. (15):

$$v_{\mathrm{i,j}} = \mathrm{x_{i,j}} + \emptyset_{\mathrm{i,j}}(\mathrm{x_{i,j}} - \mathrm{x_{k,j}}) \tag{15}$$

$\emptyset_{\mathrm{i,j}}$ is a random value in $[-1, 1]$ which controls the step size of the algorithm. The large value of $\emptyset$ will result in a large difference between the previous solution and the current one. In this situation, there is possibility that the algorithm loses the right path to the optimal solution. On the other hand, the small step size will result in immature convergence. In other words, the exploration and exploitation balance of the algorithm is controlled by $\emptyset$. $k \neq i \in \{1, \ldots, SN\}$ which is chosen randomly. $\mathrm{V_i}$ then will be evaluated and compared to the $\mathrm{X_i}$, if it is of higher quality than it, $\mathrm{X_i}$ will be replaced by it.

Algorithm 6 DHS

$nVar$: *Number of variables*
$Varmin$: *lower bound for the variables' domain*
$Varmax$: *upper bound for the variables' domain*
$nNew$: *Number of new harmonies*
$HMCR \leftarrow$ *Harmony memory consideration rate*
HM : *Harmony memory set of solutions*
$HMS \leftarrow$ *Harmony memory size*
NH : *New Harmonies*
$PAR \leftarrow$ *Pitch adjusment rate*
Δ : *Pitch adjusment parameter*
$BW \leftarrow 0.001 * (VarMax - Varmin)$
MaxIt $\leftarrow$ Maximum number of iterations

$f_1() :$ *Number of violated constraints*
$f_2() :$ *Hamming distance between two solutions*
$PS :$ *Solution of previous CSP*
$HM \leftarrow$ *initial random harmonies*

Evaluate the solutions % compute $f_1()$ *and* $f_2()$ *for each solution*
$BestSol \leftarrow$ *best solution minimizing* $f_1()$ *and* $f_2()$
for $It \leftarrow 1$ *to* $MaxIt$ **do**
  %Create New Harmonies
  for $k \leftarrow 1$ *to* $nNew$ **do**
    $NH(k).solution \leftarrow$ *generate a random solution*
    for $j \leftarrow 1$ *to* $nVar$ **do**
      $rnd \leftarrow Rand(0, 1)$
      if $rnd < HMCR$ **then**
        $i \leftarrow$ *randomly choose a number in* $[1, HMS]$
        $NH(k).solution(j) \leftarrow HM(i).solution(j)$
      end if
      $rnd \leftarrow Rand(0, 1)$
      %pitch adjustment
      if $rnd < PAR$ **then**
        $\Delta \leftarrow BW * Rand(0, 1)$
        $NH(k).solution(j) \leftarrow (BW \odot \epsilon) \otimes NH(k).solution(j) \oplus \Delta$
      end if
    end for

    %Mutation
    $NH(k).solution \leftarrow Mutate(NH(k).solution)$

    %Evaluation
    Evaluate $NH(k).solution$ *% compute* $f_1()$ *and* $f_2()$ *for each solution*
    if $f_1(NH(k).solution) < f_1(BestSol)$ && $f_2(NH(k).solution, PS) < f_2(BestSol, PS)$ **then**
      $BestSol \leftarrow NH(K).solution$
    end if
  end for

  %Selection
  $HM \leftarrow HM \bigcup NH$ *% Merge the current population and new harmonies*
  $HM \leftarrow Select(HM)$ *%select the first HMS solutions*
end for

When the recruited bees finished their search, they share the information about the location of their food sources and their nectar amounts with the onlooker bees by performing special dance in the dance area in their colony. Onlooker bees then assess the quality of the food sources based on the information that was passed to them

by the recruited bees and then with a probability related to the quality of the food sources choose food source sites [25]. For probabilistic selection, different functions can be considered; the most well-known of them are Roulette wheels, ranking-based and tournament selections. In this work, we use Roulette wheel selection which is expressed based on the fitness of the food sources (solutions) as follows:

$$p_i = \frac{f_i}{\sum_{j=1}^{NS} f_j} \tag{16}$$

f_i is the fitness value of food source i.

Based on the evaluated $p_i$and the information taken from the recruited bees, an onlooker bee selects its food source. The onlooker bee explores its neighboring areas using Eq. 2 in order to find a better solution. It then will replace its solution with a better discovered solution.

When a food source X_i cannot be further improved through a defined number of trials, the corresponding recruited bee abandons the food source. This bee is now known as scout and following Eq. (17) randomly searches for another food source.

$$x_{i,j} = x_{min,j} + Rand(0,1)(x_{max,j} - x_{min,j}) \tag{17}$$

8.2 Discrete ABC for CSPs

To convert the continuous ABC algorithm to a discrete one, all the features of the ABC algorithm including all the equations must be redefined, so they suit the definition of the discrete problems like CSPs.

The initial population of bees is generated by assigning the random values from variables' domain to the variables of the solutions. The solutions here are represented as chromosome of variables. After the generated solutions were assigned to recruited bees, they try to locally improve the solutions by searching neighboring areas of their solutions (food sources). To do so, we need to redefine the V_i as follows:

$$v_{i,j} = w \otimes (x_{i,j} \ominus (\Upsilon_i)) \tag{18}$$

where

$$\Upsilon_{i,j} = \emptyset_{i,j} \otimes (x_{i,j} \ominus x_{k,j}) \tag{19}$$

Here, we need to define our new operators $\otimes$ and $\ominus$. The idea is that the algorithm randomly selects a food source site K for each recruited bee in their neighborhood. Then, the recruited bees need to move toward that new site. In discrete problem, spaces moving from one solution to another means to share more identical values with that solution. Operator $\ominus$ identifies the variables of the first solution that have

X_i	2	1	3	5	5	4	1	2	3	4
X_K	3	1	4	3	5	4	4	5	1	3
$\emptyset_i$	0.1	0.4	0.1	0.6	0.3	0.3	0.6	0.5	0.7	0.4
γ_i	2	1	3	3	5	4	4	5	1	4

Fig. 2 An example of calculating $\gamma_i = \emptyset_{i,j} \bigotimes (x_{i,j} \ominus x_{k,j})$

different values from the one the bee is going to move toward. After identifying the variables with different values, their values will be replaced by the corresponding variables' values of solution X_K with the probability $\emptyset_{i,j}$ (a value in (0,1)). The application of $\otimes$ is to replace the variables' values of X_i with X_K considering the replacement probability $\emptyset_i$. Figure 2, with an example, shows the procedure of calculating $\gamma_i = \emptyset_{i,j} \bigotimes (x_{i,j} \ominus x_{k,j})$. And then $\ominus$ compares $x_{i,j}$ and γ_i to identify the different variables. Then, $\ominus$ compares $x_{i,j}$ and γ_i to identify the different variables.

When the differences have been identified, those variables of $x_{i,j}$ will be replaced by corresponding variables' values of γ_i with probability w. Figure 3 shows the procedure for $w = 0.5$.

The quality of the solutions is evaluated based on the number of violated constraints. The less constraints a solution violates, the more quality that solution is. V_i then will be assessed using defined fitness function. If it has higher quality than X_i, it will replace it. The improvement in developing every food site is monitored using parameter *C*. If for defined number of trials, the expected improvement has not been achieved, the algorithm replaces that site with a randomly generated site (solution).

8.3 Discrete ABC (DABC) for DCSPs

As discussed earlier, the exploitation and exploration features of the ABC algorithm are controlled by ∅. From previous discussion, we know that the high value of ∅ encourages the exploration and the low value encourages the exploitation of the algorithm [27]. Although ∅ is a random parameter which cannot be set manually, we can still control it by determining the right intervals to meet the problem requirements. By initializing this parameter appropriately, exploitation will be emphasized first, followed by exploration, as per our solving methodology.

Algorithm 7 presents the pseudocode of DABC.

X_i	2	1	3	5	5	4	1	2	3	4
γ_i	2	1	3	3	5	4	4	5	1	4
$w = 0.5$										
$w \bigotimes (x_{i,j} \ominus x_{k,j})$	2	1	3	3	5	4	4	2	1	4

Fig. 3 An example of calculating $w \bigotimes (x_{i,j} \ominus x_{k,j})$

Algorithm 7 DABC

$f_1 \leftarrow$ *calculate the number of violated constraints*
$f_2 \leftarrow$ *distance from previous CSP's solution (Hamming distance* d_H*)*
$PS \leftarrow$ *Previous Solution*
$L \leftarrow$ *Abondonment limit parameter*
$C \leftarrow 0$
$MaxIt \leftarrow$ *Maximum iteration number*
$nPop \leftarrow$ *Population size*
$nOnlooker \leftarrow nPop\%$ *number of Olooker bees*
$BestSol.Cost \leftarrow Inf$
$w \leftarrow$ *inertia weight*
Initialize the population
for $It \leftarrow 1$ *to* $MaxIt$ **do**
    % Recruited bees
    for $i \leftarrow 1$ *to* $nPop$ **do**
        $K \leftarrow$ *choose randomly k from* $[1 : i-1 \;\; i+1 : nPop]$
        $newbee \leftarrow newsite(bee(i), bee(k), w)\%$ *calculate* V_i
        evaluate the newbee% according tothefitnessfunction
        if $f_1(newbee) < f_1(bee(i))\&f_2(newbee) < f_2(bee(i))$ **then**
            $bee(i) \leftarrow newbee$
        else
            $C(i) \leftarrow C(i) + 1$
        end if
    end for
    for $i \leftarrow 1$ *to* $nPop$ **do**

        $F(i) = \exp \leftarrow (-bee(i).cost/mean(bee(i).cost))$
    end for
    $p = F/sum(F)$
    % Onlooker bees

    for $m \leftarrow 1$ *to* $nOnlooker$ **do**
        $i \leftarrow RouletteWheelSelection(P)$
        $K \leftarrow$ *choose randomly k from* $[1 : i-1 \;\; i+1 : nPop]$
        $newbee \leftarrow newsite(bee(i), bee(k), w)$
        evaluate the newbee
        if $f_1(newbee) < f_1(bee(i))\&f_2(newbee, Ps) < f_2(bee(i), PS)$ **then**
            $bee(i) \leftarrow newbee$
        else
            $C(i) \leftarrow C(i) + 1$
        end if
    end for
    %Scout bees
    for $i \leftarrow 1$ *to* $nPop$ **do**
        if $C(i) = L\%$ *checks to see if approperiate progress has been achieved* **then**
            $bee(i) \leftarrow$ *randomly generate a solution*
            evaluate the $bee(i)$
            $C(i) \leftarrow 0$
        end if
    end for%Update the best solution
    for $i \leftarrow 1$ *to* $nPop$ **do**
        if $f_1(bee(i)) < f_1(BestSol)\&f_2(bee(i)) < f_2(BestSol)$ **then**
            $BestSol \leftarrow bee(i)$
        end if
    end for
end for
function $newbee =$ ***newsite***$(beei, beek, w)$
$newbee \leftarrow beei$
$\phi \leftarrow Rand\ (0, 1)$
for ***for*** $i \leftarrow 1$ *to* $nVar$ *%* $nVar$ *is number of variables* **do**
    $rnd \leftarrow Rand(0, 1)$
    if $rnd < \phi$ **then**
        $beei(i) \leftarrow beek(i)$
    end if
    for $i \leftarrow 1$ *to* $nVar$ **do**
        $rnd \leftarrow Rand(0, 1)$
        if $rnd < w$ **then**
            $newbee(i) \leftarrow beei(i)$
        end if
    end for
end for

9 Discrete Mushroom Reproduction Optimization (DMRO)

In [28], we introduced a new nature-inspired optimization algorithm, namely mushroom reproduction optimization (MRO) inspired and motivated by the reproduction and growth mechanisms of mushrooms in nature. MRO follows the process of discovering rich areas (containing good living conditions) by spores to grow and develop their own colonies. In fact, this algorithm mimics the way mushrooms reproduce and migrate to rich areas with adequate nourishment, moisture and light. The reproduction model is twofold: a) producing spores and b) distributing them randomly in the environment. This reproduction mechanism leads to bigger mushroom colonies located in different regions with suitable living conditions. The searching agents are parent mushrooms and spores. The transporter mechanism is the artificial wind that stochastically distributes spores in different locations of the problem space. The probability of developing colonies in rich areas is higher than poor areas. In fact, rich areas are those containing high-quality solutions and probably the optimal one. In the initial phase, a mature mushroom (the parent), located in the problem space, distributes spores throughout the problem space. If no wind is blowing and the living conditions in the neighboring area are good, then upon landing, spores germinate and grow (local search).

$$X_{i,j} = X_i^{parent} + Rand(0, 1) \tag{20}$$

Equation (20) calculates the new location of spore j of colony (parent mushroom) i ($X_{i,j}$). Here, X_i^{parent} is the location of the parent i and Rand(0,1) generates a random number in (0,1). However, if the wind is blowing, spores are moved to different parts of the problem space and land in new locations. Next, spores grow and become mature mushrooms. Eqs. (21) and (22) present the movement of the spores induced by wind.

$$X_{i,j} = X_i^{parent} + Move_j^{wind} \tag{21}$$

$$Move_j^{wind} = (X_i^* - X_k^*) \times \left(\frac{Ave(i)}{T_{ave}}\right)^m \times Rand(-\delta, \delta) \times rs + Rand(-r, r) \tag{22}$$

Here, X_i^* and X_k^* are the parent solutions of the colonies i and k, $Ave(i)$ is the average of solutions quality of colony i, T_{ave} is the total average of all colonies, $Rand(-\delta, \delta)$ is a vector that determines direction movement of the wind, $rs \in (0, 1)$ is the size of random step and $Rand(-r, r)$ is the random movement of the spores to their neighboring areas with radius r.

This searching process of production and distribution is repeated, and some spores will ultimately find the richest area. In the final phase, the main operation, consequently the colony expands locally and quickly. This will result in finding the optimal solution.

The initial version of MRO, described above, was introduced to tackle continues optimization problems [28]. To apply it to discrete optimization problems, such as

CSPs, the features of the algorithm must be converted to discrete features to be able to deal with a discrete problem space which is the space of all possible combinations of variables values. The steps that must be taken to convert continuous MRO to discrete MRO are discussed below.

The initial population of the MRO is generated randomly by assigning randomly chosen values from a given CSP variables' domain to variables of the initial solutions. Two main features of the MRO are local movement which its role is to locally extend the colonies in order to make a local improvement in the solutions of a colony and the second feature is the global movement of the spores (migration of the spores) induced by wind.

According to the mushrooms' life cycle studies, mushroom colonies are more likely to be found in the regions that have good living condition for mushrooms. Once a spore found such region, it then begins to locally extend its colony by distributing its spores in the neighboring areas of its colony. Local extension of a mushroom colony was discussed in [28]; however, to fit the definition of CSPs, a redefined version of the local movement is required. In a CSP, the search space consists of all combination of values in the form of chromosomes (potential solutions). A local movement of a solution (spore in MRO) corresponds to making a slight change in a variable value. This basically corresponds to moving toward the surrounding areas of a solution with radius r, as shown in Algorithm 8.

Algorithm 8 Local Movement of spores in DMRO

```
for K ← 1 to nSpore do% nSpores is number of the spores
    Mush(K).Position ← ParentMushroom.Position
    for l ← 1 to nVar do
        rnd ← Rand(0,1)
        if rnd < Threshold1 then
            Mush(K).Position(l) ← Mush(K).Position(l) + Rand(-r , r)% r is the local movement radius
        end if
    end for
    % keep the solutions within boundaries of the problem space
    Mush(K).Position ← max(Mush(K).Position, varmin))
    Mush(K).Position← min(Mush(K).Position,varmax)
    Mush(K).weight ← evaluate(Mush(K))
end for
avg ← mean(Mush.cost)
Mush ← bestMush
```

The second main feature is the global movement induced by wind. Mature mushrooms generate spores and distribute them by wind through the environment. The wind has stochastic and unpredictable features like direction and

speed. Therefore, spores will be distributed stochastically throughout the problem space. Many spores are distributed through the environment, but those regions with better living conditions have more chance to host the bigger mushroom colonies. In fact, spores move everywhere and those reaching regions with good living conditions begin to germinate and extend their colonies. The quality of a region is represented by the average of the quality of its solutions (mushrooms). The global movement is defined as shown in Algorithm 9.

As described in Algorithm 9, the position of a new spore is determined based on the location of its parent, the quality of the other regions, as well as the random movement of spores (given that they are light, spores can move unpredictably towards different directions). The parameter *rs* (size of random step) controls the ratio of influencing a solution by other solutions. A higher value of *rs* together with a lower value of *RV* (number of random variables) will allow a solution to move rapidly toward other solutions and converge to them, while a lower value of *rs* together with a higher value of *RV* will increase the chance of discovering new regions. In other words, this process corresponds to exploitation versus exploration strategies, controlled by *rs* and *RV*. A high value of *rs* and low value of *RV* encourage exploitation, while the opposite will encourage exploration. When solving a DCSP, our DMRO applies exploitation and exploration, as described in our general methodology.

Algorithm 9 Global movement induced by wind

for $\ell \leftarrow 1$ to nVar **do**
  rnd ← Rand(0,1)
  if rnd < Threshold2 **then**
    Mush(i).Position(l) ← Rand(0,Domain)
  end if
end for
% Movement towards better regions
for l ← 1 to nVar **do**
  rnd ← Rand(0,1)
  if rnd < $(\frac{ave(i)}{Tave}) \times rs$ **then**
    Mush(i).Position(l) ← Mush(k).Position(l)
  end if
end for
% Random Movement
RV ← Rand(1,nVar) % Generate a random number
RVS=Randi([1,n],1,RV) % generate an array of RV randomly chosen variables
for l ← 1 to RV **do**
  Mush(i).Position(RVS(l)) ← Rand(0,Domain)
end for
Return Sol_i

Algorithm 10 DMRO for DCSPs

```
nSpore ← Number of the permitted spores for each parent mushroom
nPop ← Number of the parent mushrooms
FW ← 0.001∗ (VarMax - Varmin) % Fret Width
MaxIt ← Maximum number of iterations
f() ← evaluate the weight of the solution based on (13)
Mush ← Create initial populatiuon
Evaluate the solutions
Calculate Ave_i
Calculate Tave
BestSol % save the best solution in each iteration
for It ← 1 to MaxIt do
    Create New Harmonies
    for i ← 1 to nPop do
        if Ave_i < Tave then
            for k ←1 to nPop do
                if f(Mush(i)) < f_1(Mush(k)) then
                    Dave ← ave_i / Tave % distribute nSpore Spores in the problem space by wind
                    Mush(i) ← WindMove(Mush(i),Mush(k),ave,Tave)
                    % Local distribution of spores
                    (Mush(i),ave(i)) ← LocalMovement(Mush(i),nSpore)
                else% Local distribution of spores
                    (Mush(i),ave(i)) ← LocalMovement(Mush(i),nSpore)
                    if f(Mush(i)) < f(BestSol) then
                        BestSol ← Mush(i)
                    end if
                end if
                Tave ← mean(ave)
            end for
        end if
    end for
end for
```

Algorithm 10 presents the discrete MRO for tackling DCSPs. At each iteration, the global best is updated according to these two objectives.

10 Experimentation

10.1 Experimentation Environment and Problem Instances

In this section, we report on the experiments we conducted in order to assess the performance of the nature-inspired techniques we have presented, for solving DCSP instances, randomly generated using a variant of the RB model [10]. The RB model has the ability to generate hard-to-solve instances (those near the phase transition). As discussed before, a DCSP can be seen as a sequence of static CSPs, each obtained from the previous one by adding a set of new constraints. In this regard, we generate two instances for each random DCSP. The first one is simply a static CSP instance. The second one is obtained by adding a set of new constraints to the initial instance, such that we will end up with *nv* constraint violations of the solution to the

initial CSP. We also make sure that the distance between the new solution and the old one is equal to *nv*. Note that *nv* is set to 8 in our experiments. In this regard, if *nv* constraints need to be added in a dynamic way to produce *nv* violations, we proceed as follows. Starting from the initial randomly generated CSP, we remove *nv* constraints in order to produce the static CSP instance. These removed constraints will be added incrementally to the second instance. This will guarantee the consistency of the second instance.

Each CSP instance is generated as follows using the parameters n, p, α and r where n is the number of variables, $p(0 < p < 1)$ is the constraint tightness (the number of incompatible tuples over the Cartesian product of the two involved variables domains), and r and α $(0 < r, \alpha < 1)$ are two positive constants used by the model RB [10].

1. Select $t = r \times n \times \ln n$ distinct random constraints. Each random constraint is formed by selecting 2 of n variables (without repetition).
2. For each constraint, we uniformly select $q = p \times d^2$ distinct incompatible pairs of values, where $d = n^{\alpha}$ is the domain size of each variable.
3. All the variables have the same domain corresponding to the first d natural numbers $(0, \ldots, d-1)$.

According to [10], the phase transition P_{cr} is calculated as follows: $P_{cr} = 1 - e^{-}\alpha/r$. In theory, solvable problems are those with a tightness $p < P_{cr}$. Given that the phase transition is 0.7, we generate CSPs (having 100 and 200 variables) with a tightness between 0.3 and 0.7. The consistency of each generated instance is confirmed in practice, using an exact method (backtrack search algorithm).

All methods, used in these comparative experiments, have been implemented using MATLAB R2013b. In addition to the nature-inspired techniques we described in this paper, we have also implemented a dynamic variant (following our general methodology) of the genetic algorithm we proposed in [29] for solving CSPs. The population size of each method is fixed to 50. For ABC, the number of employed bees and onlooker beers is 50. The other controlling parameters of the algorithms are tuned to their best.

All experiments have been performed on a PC with Intel Core i7-6700K 4.00 GHz processor and 32GB of RAM.

The results of the experiments are reported in terms of running time (RT) in seconds, number of violated constraints (NVC), success rate (SR) and number of constraint checks (NCC). Each experiment is conducted five times, and the average result (one of the above parameters) is returned.

10.2 Results Regarding the First Instance

Figure 4 indicates the convergence trends of the proposed methods to the solutions of the instances. The left and right charts correspond to CSPs with 100 and 200 variables, respectively. The tightness is set to 0.6 which corresponds to the hardest problems to solve. As we see in all the experiments, the proposed algorithms were

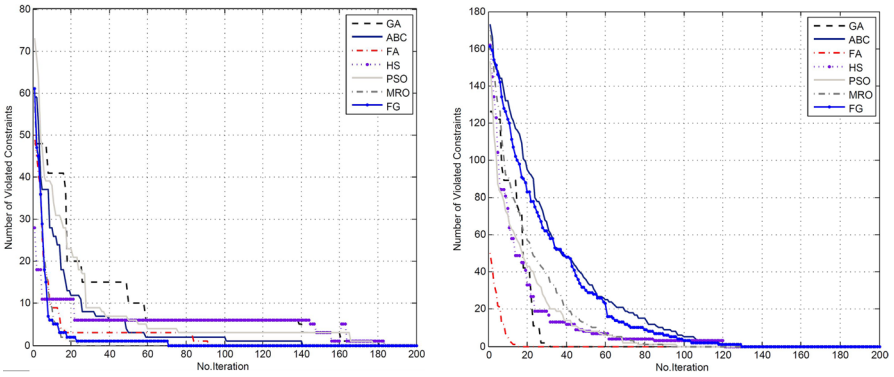

Fig. 4 Convergence trend in number of violated constraints for 100 vars (left chart) and 200 vars (right chart)

successful in finding the solution; however, different methods reveal different behaviors in terms of the required number of iterations to reach the solution. In the case of 100 variables, DFGOA, SADFA and DMRO were able to converge toward the optimal solution after 20 iterations, where it took a bit longer for the other methods. For 200 variables, SADFA is the winner, followed with GAs.

10.2.1 Results regarding the second instance

Figure 5 shows the convergence trends in terms of minimum distance (left charts) and number of violated constraints (right charts) when solving CSP instances with 200 variables, and with tightness 0.4 (top charts), 0.55 (middle charts) and 0.6 (bottom charts). In all charts, we can easily see how each nature-inspired technique successfully minimizes both objectives. The number of constraints violations actually reaches 0 by all the methods, as we can see in the left charts (which corresponds to completely solving the corresponding instance). As for the quality of the solution returned (distance to the optimal), all methods were able to return a near-to-optimal distance of 10 (optimal is 8).

Note that the tests reported in Fig. 5 only consider the number of iterations as a comparative parameter, for reaching the optimal solution. In order to see how does this number translate into the actual running time needed by each method to return the solution, we report in Table 9 the detailed results in terms of running time (RT), number of violated constraints (NVC), number of constraint checks (NCC) and quality of the returned solution. The latter (similarity or Sim.) is measured as the percentage of similar values to the optimal solution (100% corresponds to the optimal solution). CSP instances have 200 variables, with a tightness value ranging between 0.3 and 0.6. As noticed in the table, all the methods were successful again here to return a consistent solution (corresponding to 0 violated constraints). A

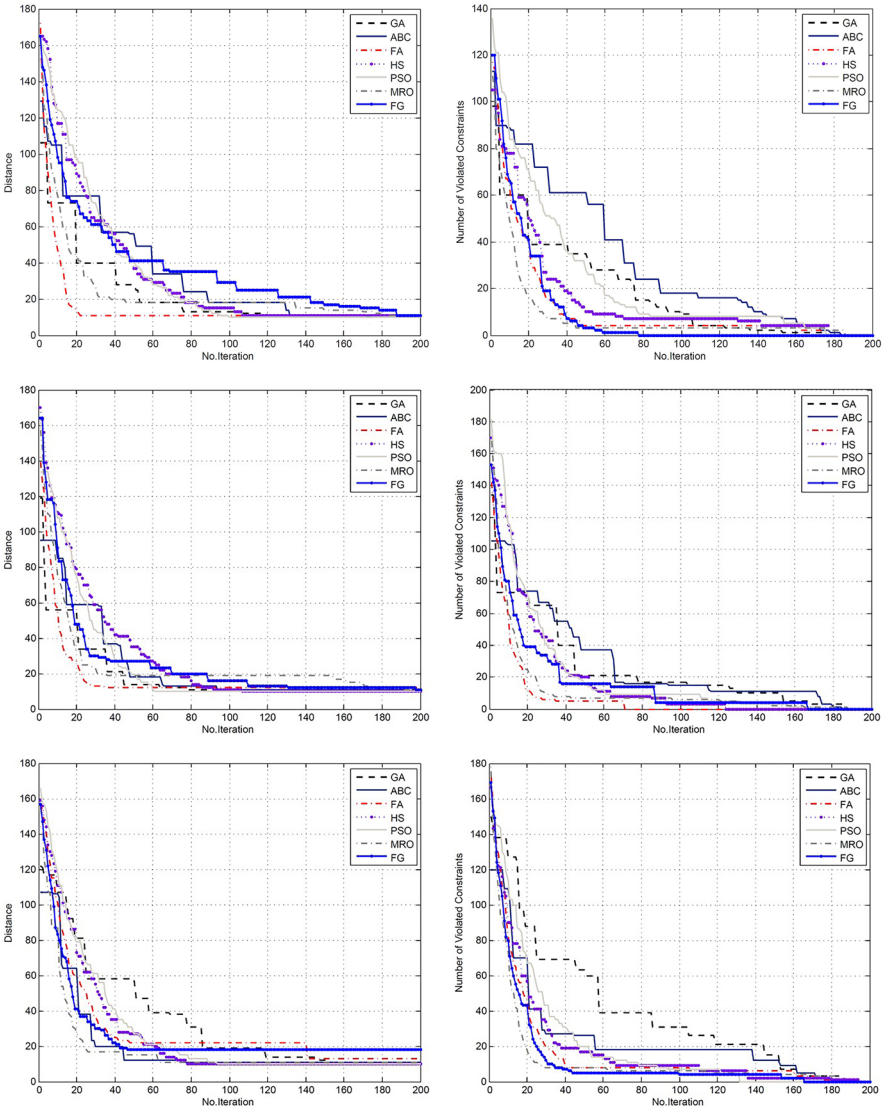

Fig. 5 Convergence trend in NVC (left chart) and distance (right chart)

near-to-optimal solution (88% to 90%) is also obtained by all the techniques. There is, however, a noticeable difference in terms of running time, especially when the tightness is equal to 0.6 (which corresponds to the hardest problems to solve). In this regard, FGOA and ABC are the best methods in both running time and quality of the solution returned (similarity to the previous one), with HS coming next. We conducted further experiments with 300 variables, and the results are listed in Table 10. Here again, FGOA and ABC show the best results, followed by HS and DPSO.

Table 9 Comparative test results for CSP instances with 200 variables

Tightness		0.3	0.35	0.4	0.45	0.5	0.55	0.6
MRO	**RT**	125.71	217.40	223.14	308.49	305.11	373.07	439.29
	NVC	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	NCC	8.94E+07	1.53E+08	1.54E+08	2.27E+08	2.11E+08	2.66E+08	3.23E+08
	Sim.	0.88	0.89	0.89	0.89	0.88	0.89	0.89
FG	**RT**	224.99	226.31	232.22	247.64	257.87	247.55	274.07
	NVC	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	NCC	1.47E+08	1.42E+08	1.48E+08	1.55E+08	1.77E+08	1.57E+08	1.92E+08
	Sim.	0.90	0.90	0.89	0.90	0.88	0.89	0.89
GA	**RT**	210.43	205.45	236.04	236.94	257.11	266.11	322.65
	NVC	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	NCC	1.46E+08	1.52E+08	1.75E+08	1.67E+08	1.71E+08	1.93E+08	2.37E+08
	Sim.	0.89	0.90	0.89	0.89	0.90	0.89	0.89
HS	**RT**	198.55	218.65	225.78	244.03	253.20	253.40	289.74
	NVC	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	NCC	1.07E+08	1.19E+08	1.23E+08	1.28E+08	1.29E+08	1.36E+08	1.61E+08
	Sim.	0.90	0.88	0.89	0.90	0.88	0.90	0.90
FA	**RT**	206.66	207.15	221.53	210.60	220.04	298.45	349.01
	NVC	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	NCC	1.55E+08	1.55E+08	1.56E+08	1.49E+08	1.52E+08	2.20E+08	2.57E+08
	Sim.	0.89	0.89	0.89	0.89	0.89	0.89	0.87
PSO	**RT**	208.67	234.60	235.22	295.14	298.49	297.79	312.74
	NVC	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	NCC	1.07E+08	1.22E+08	1.22E+08	1.58E+08	1.61E+08	1.62E+08	1.70E+08
	Sim.	0.90	0.90	0.90	0.90	0.90	0.90	0.90
ABC	**RT**	211.25	217.01	218.42	221.52	228.47	232.72	278.00
	NVC	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	NCC	1.49E+0.08	1.50E+08	1.57E+08	1.52E+08	1.59E+08	1.61E+08	1.86E+08
	Sim.	0.90	0.90	0.89	0.89	0.88	0.90	0.89

Mouhoub [7]

Note that we have implemented an exact method based on backtrack search and constraint propagation, as reported in [7]. This exact method has then been used to conduct the experiments reported in this section. Overall, the exact method performed poorly (especially for high tightness values) in comparison with the results reported in Table 9. In the case of 300 variables (reported in Table 10), the exact method was not even able to find a consistent solution within the time allocated for the nature-inspired techniques.

Table 10 Comparative test results for CSP instances with 300 variables

Tightness		0.3	0.35	0.4	0.45	0.5	0.55	0.6
MRO	**RT**	1542.42	2458.06	2711.04	3633.94	3727.23	4501.51	4972.79
	Sim.	0.88	0.86	0.86	0.85	0.80	0.80	0.80
FG	**RT**	2733.86	2704.94	2596.73	2919.88	2951.20	3030.47	3118.97
	Sim.	0.92	0.90	0.90	0.88	0.88	0.88	0.89
GA	**RT**	2383.28	2321.07	2874.88	2822.50	3104.32	3032.29	3769.14
	Sim.	0.90	0.90	0.90	0.85	0.85	0.85	0.85
HS	**RT**	2363.04	2599.47	2572.48	2781.73	3005.58	3029.23	3515.85
	Sim.	0.90	0.90	0.90	0.88	0.88	0.90	0.90
FA	**RT**	2437.12	2419.19	2606.38	2417.10	2648.49	3442.44	4087.62
	Sim.	0.88	0.90	0.85	0.85	0.85	0.80	0.80
PSO	**RT**	2388.21	2877.00	2717.72	3351.06	3664.17	3415.45	3593.69
	Sim.	0.90	0.90	0.90	0.90	0.90	0.90	0.90
ABC	**RT**	2419.18	2460.61	2684.67	2680.60	2589.92	2630.89	3260.63
	Sim.	0.90	0.90	0.88	0.88	0.88	0.89	0.89

11 Conclusion and Future Work

We investigated the applicability of nature-inspired techniques for solving DCSPs. Our contribution has a significant socioeconomic impact given the highly dynamic behavior that most real-world problems exhibit. In this regard, our work is very relevant for those applications under dynamic constraints, and where any change in requirements has to be handled in a short amount of time. More precisely, a new consistent scenario (satisfying the old and new constraints) has to be found as early as possible and should not be that different from the old one. This latter objective is particularly important in constraint optimization problems such as scheduling and planning, where the new schedule should be as close as possible to the old one. Using our general solving methodology, we show that nature-inspired techniques can be used to tackle the objectives we listed above. This claim has been validated through several experiments conducted on randomly generated DCSPs. The results of the experiments are very promising and encouraging. However, real-world scenarios are not random and a further investigation is needed in this regard. This has motivated us to consider, in the near future, further experiments on real-world scenarios, especially in the case of healthcare staff scheduling [30]. This latter application can be seen as preference-based multi-objective constraint optimization problem, where the goal is to come up with a set of Pareto-optimal solutions that satisfy a set of requirements and optimizing some objectives, including personnel preferences. The challenge here is to maintain the Pareto-optimal set anytime there is a change in requirements. We also plan to tackle configuration applications, where the user interacts with the system by adding or removing constraints and see the corresponding changes in an incremental manner [2]. Finally, we will consider dynamic vehicle routing

[5], where a new optimal path is needed when unpredictable changes (expressed through constraints) such as accidents occur.

Data Availability As clearly indicated in the Experimentation section of the manuscript, the data used to perform the experiments have been randomly generated using the RB model [10]. The Experimentation section provides the description on how these data were randomly generated.

Declarations

Conflicts of Interest On behalf of all authors, the corresponding author states that there is no conflict of interest.

References

1. Dechter R (2003) Constraint processing. Morgan Kaufmann
2. Alanazi E, Mouhoub M (2014) Configuring the webpage content through conditional constraints and preferences. In: Modern Advances in Applied Intelligence - 27th International Conference on Industrial Engineering and Other Applications of Applied Intelligent Systems, IEA/AIE 2014, Kaohsiung, Taiwan, June 3-6, 2014, Proceedings, Part II. pp 436–445
3. Amilhastre J, Fargier H, Marquis P (2002) Consistency restoration and explanations in dynamic CSPS–application to configuration. Artif Intell 135(1–2):199–234
4. Mouhoub M, Sukpan A (2012) Managing dynamic CSPS with preferences. Appl Intell 37(3):446–462
5. Alanazi E, Mouhoub M, Halfawy M (2017) A new system for the dynamic shortest route problem. In: Benferhat S, Tabia K, Ali M (eds) Advances in Artificial Intelligence: From Theory to Practice - 30th International Conference on Industrial Engineering and Other Applications of Applied Intelligent Systems, IEA/AIE 2017, Arras, France, June 27-30, 2017, Proceedings, Part I, Lecture Notes in Computer Science, vol 10350. Springer, pp 51–60
6. Roos N, Ran Y, Van Den Herik J (2000) Combining local search and constraint propagation to find a minimal change solution for a dynamic CSP. In: International Conference on Artificial Intelligence: Methodology, Systems, and Applications. Springer, pp 272–282
7. Mouhoub M (2004) Systematic versus non systematic techniques for solving temporal constraints in a dynamic environment. AI Commun 17(4):201–211
8. Verfaillie G, Schiex T (1994) Solution reuse in dynamic constraint satisfaction problems. In: The Twelfth National Conference On Artificial Intelligence, vol 94. AAAI, pp 307–312
9. Talbi EG (2009) Metaheuristics: from design to implementation, vol 74. John Wiley & Sons
10. Xu K, Li W (2000) Exact phase transitions in random constraint satisfaction problems. J Artif Intell Res 12:93–103
11. Bidar M, Mouhoub M (2019a) Discrete particle swarm optimization algorithm for dynamic constraint satisfaction with minimal perturbation. In: 2019 IEEE International Conference on Systems, Man and Cybernetics, SMC 2019, Bari, Italy, October 6-9, 2019. IEEE, pp 4353–4360
12. Bidar M, Mouhoub M (2019b) Self-adaptive discrete firefly algorithm for minimal perturbation in dynamic constraint satisfaction problems. In: IEEE Congress on Evolutionary Computation, CEC 2019, Wellington, New Zealand, June 10-13, 2019. IEEE, pp 2620–2627
13. Schiex T, Verfaillie G (1994) Nogood recording for static and dynamic constraint satisfaction problems. Int J Artif Intell Tools 3(2):187–208
14. Zivan R, Grubshtein A, Meisels A (2011) Hybrid search for minimal perturbation in dynamic CSPS. Constraints 16(3):228–249
15. Yang XS (2008) Nature-Inspired Metaheuristic Algorithms. Luniver Press
16. Bidar M, Mouhoub M, Sadaoui S (2018) Discrete firefly algorithm: A new metaheuristic approach for solving constraint satisfaction problems. In: 2018 IEEE Congress on Evolutionary Computation (CEC). pp 1–8

17. Bidar M, Mouhoub M (2019c) Self-adaptive discrete firefly algorithm for minimal perturbation in dynamic constraint satisfaction problems. In: 2019 IEEE Congress on Evolutionary Computation (CEC). pp 11–19
18. Kennedy J, Eberhart R (1995) Particle swarm optimization. In: Proceedings of ICNN'95-International Conference on Neural Networks. IEEE, vol 4. pp 1942–1948
19. Poli R, Kennedy J, Blackwell T (2007) Particle swarm optimization. Swarm Intell 1(1):33–57
20. Nickabadi A, Ebadzadeh MM, Safabakhsh R (2011) A novel particle swarm optimization algorithm with adaptive inertia weight. Appl Soft Comput 11(4):3658–3670
21. Fattahi E, Bidar M, Kanan HR (2018) Focus group: An optimization algorithm inspired by human behavior. Int J Comput Intell Appl 17:1–27
22. Bidar M, Mouhoub M, Sadaoui S (2020) Discrete focus group optimization algorithm for solving constraint satisfaction problems. In: Rocha AP, Steels L, van den Herik HJ (eds) Proceedings of the 12th International Conference on Agents and Artificial Intelligence, ICAART 2020, Volume 2, Valletta, Malta, February 22-24, 2020. SCITEPRESS, pp 322–330
23. Geem ZW, Kim JH, Loganathan G (2001) A new heuristic optimization algorithm: Harmony search. Simulation 76(2):60–68
24. Degertekin SO (2008) Optimum design of steel frames using harmony search algorithm. Struct Multidiscip Optim 36(4):393–401
25. Karaboga D (2005) An idea based on honey bee swarm for numerical optimization. Technical Report-tr06 Erciyes University, Computer Engineering Department 200:1–10
26. Wf Gao, Sy Liu (2012) A modified artificial bee colony algorithm. Comput Oper Res 39(3):687–697
27. Gothania B, Mathur G, Yadav R (2014) Accelerated artificial bee colony algorithm for parameter estimation of frequency-modulated sound waves. International Journal of Electronics and Communication Engineering 7:63–74
28. Bidar M, Kanan HR, Mouhoub M, Sadaoui S (2018) Mushroom reproduction optimization (MRO): A novel nature-inspired evolutionary algorithm. In: 2018 IEEE Congress on Evolutionary Computation, CEC 2018, Rio de Janeiro, Brazil, July 8-13, 2018. IEEE, pp 1–10
29. Abbasian R, Mouhoub M (2016) A new parallel ga-based method for constraint satisfaction problems. Int J Comput Intell Appl 15(03):1650017
30. Topaloglu S (2006) A multi-objective programming model for scheduling emergency medicine residents. Comput Ind Eng 51(3):375–388

Authors and Affiliations

Mahdi Bidar[1] · Malek Mouhoub[2]

✉ Malek Mouhoub
malek.mouhoub@uregina.ca

Mahdi Bidar
Mahdi.Bidar@auroramj.com

[1] Aurora Cannabis Enterprises Inc., 510 Seymour Street, 9th floor, Vancouver V6B1V5, BC, Canada

[2] Department of Computer Science, University of Regina, Regina, SK, Canada