Applied Soft Computing 12 (2012) 342–352

Contents lists available at SciVerse ScienceDirect

Applied Soft Computing

journal homepage: www.elsevier.com/locate/asoc

DisABC: A new artificial bee colony algorithm for binary optimization

Mina Husseinzadeh Kashan, Nasim Nahavandi*, Ali Husseinzadeh Kashan

Department of Industrial Engineering, Faculty of Engineering, Tarbiat Modares University, P.O. Box 14117-13116, Tehran, Iran

ARTICLE INFO

Article history:
Received 25 November 2010
Received in revised form 12 May 2011
Accepted 5 August 2011
Available online 22 August 2011

Keywords:
Binary optimization
Artificial bee colony algorithm
Swarm intelligence
Dissimilarity measure of binary structures
Uncapacitated facility location problem

ABSTRACT

Artificial bee colony (*ABC*) algorithm is one of the recently proposed swarm intelligence based algorithms for continuous optimization. Therefore it is not possible to use the original *ABC* algorithm directly to optimize binary structured problems. In this paper we introduce a new version of *ABC*, called *DisABC*, which is particularly designed for binary optimization. *DisABC* uses a new differential expression, which employs a measure of dissimilarity between binary vectors in place of the vector subtraction operator typically used in the original *ABC* algorithm. Such an expression helps to maintain the major characteristics of the original one and is respondent to the structure of binary optimization problems, too. Similar to original *ABC* algorithm, *DisABC*'s differential expression works in continuous space while its consequence is used in a two-phase heuristic to construct a complete solution in binary space. Effectiveness of *DisABC* algorithm is tested on solving the uncapacitated facility location problem (*UFLP*). A set of 15 benchmark test problem instances of *UFLP* are adopted from OR-Library and solved by the proposed algorithm. Results are compared with two other state of the art binary optimization algorithms, i.e., *binDE* and *PSO* algorithms, in terms of three quality indices. Comparisons indicate that *DisABC* performs very well and can be regarded as a promising method for solving wide class of binary optimization problems.

1. Introduction

Several heuristic algorithms have been developed for solving combinatorial optimization problems. These algorithms can be classified into different groups depending on the criteria being considered, such as population based, trajectory based, iterative based, stochastic, deterministic, etc. An algorithm working with a group of solutions and trying to improve them, is called population based [16]. One of the most recently population based methods is artificial bee colony (*ABC*) algorithm which belongs to a family of algorithms called swarm intelligence based algorithms. *ABC* was first introduced by Karaboga in 2005 for solving continuous optimization problems. To optimize over a continuous space, *ABC* simulates the intelligent foraging behaviour of honey bee swarm. In this algorithm a colony of artificial bees, including three groups of: employed bees, onlooker bees and scout bees, are considered. *ABC* performs based on sharing nectar information of food sources between two groups of bees, namely the employed bee and onlooker bee. When the nectar of a food source is abandoned by the bees, a new food source is randomly determined by a scout bee and is replaced with that abandoned one.

To model the intelligent behaviour of honey bee swarms several approaches have been proposed and implemented for solving various types of optimization problems. Karaboga and Basturk [17] compared the performance of artificial bee colony with differential evolution (*DE*), particle swam optimization (*PSO*) and evolutionary algorithm (*EA*) on multi-dimensional numerical optimization problems. Bao and Zeng [2] introduced three selection strategies, such as disruptive selection strategy, tournament selection strategy and rank selection strategy to improve the population diversity and avoid the premature convergence of *ABC*. Through improving the exploration capacity of *ABC*, Tsai et al., [23] proposed an Interactive Artificial Bee Colony (*IABC*) algorithm based on employing the Newtonian law of universal gravitation. Pan et al., [19] proposed a discrete artificial bee colony (DABC) algorithm to solve the lot-streaming flow shop scheduling problem with total weighted earliness and tardiness penalties criterion. Their algorithm uses a self adaptive strategy for generating neighbouring food sources based on insert and swap operators applied on a food source represented as a discrete job permutation. Zhang et al., [25] developed an artificial bee colony clustering algorithm to optimally partition *N* objects into *K* clusters. For this problem Karaboga and Ozturk [18] proposed an ABC algorithm and tested it with the results of a *PSO* based algorithm. Their results indicate that ABC can efficiently be used for multivariate data clustering.

As mentioned earlier, the original version of *ABC* algorithm is only able to optimize continuous problems. Therefore, we cannot use it directly for optimization in binary spaces. In

* Corresponding author.
E-mail addresses: n_nahavandi@modares.ac.ir, nasim_nahavandi@yahoo.com (N. Nahavandi).

1568-4946/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.asoc.2011.08.038

spite of the wide application of binary optimization problems in real word engineering and the ability of *ABC* algorithm in solving various problems, there is no effort put on developing the binary versions of *ABC* algorithm. To our knowledge, this paper presents the first effort on making the use of *ABC* algorithm applicable for solving binary optimization problems. Using dissimilarity measure of binary structures in place of the arithmetic subtraction operator, a differential expression is proposed which maintains the major characteristics of the original *ABC*'s expression. The main feature of this new operator is that it works in continuous space while the consequences are used in discrete space. One of the main benefits along with our approach is that it enables us to utilize the structural knowledge of the problem through using general or problem dependent heuristic procedures, during the construction of the new solution. This feature could be so crucial to an evolutionary search and may facilitate steering the search quickly toward the global optimum. Another advantage of our algorithm, which is called *DisABC* is that in contrary with transformation based methods or discretization methods [7], etc., which may cause some information to be lost (because for example we have to search a much larger solution space) or may fail to refine solutions (which is an indication of the lack of exploration ability), or may cause the lost of information on good substrings that should be remained together in a good solution, there is no information lost through enforcing *ABC*'s differential expression to work in continuous space, while using the consequences in binary space. Unlike other approaches, our approach does not map the problem space into another space and then transforming back the obtained solution into the original space in order to solve the problem.

To verify effectiveness of our algorithm, experiments are conducted on the benchmarked suites of the uncapacitated facility location problem (*UFLP*), collected from OR-Library. In *UFLP*, we have to select a set of facilities to be set up and a set of clients for each facility to service so as to minimize the total cost of setting up the facilities and servicing the clients [10]. *UFLP* forms the underlying model in several combinatorial problems like set covering, set partitioning, airline crew scheduling, etc., and is a sub-problem for various location analysis. Extensive computational experiments are carried out to find out the behaviour of *DisABC* algorithm, hybridized with a single local search module, under various setting of control parameters and also to measure how it competes with other state of the art binary optimization algorithms.

The paper is organized as follows. The following section reviews the basic preliminaries of *ABC*. Section 3 goes toward developing the new binary version of *ABC* algorithm called *DisABC* algorithm. In Section 4, the *UFLP* problem is introduced. Section 5 gives details on the effectiveness comparisons and suggestions on the parameter setting. Finally, Section 6 gives the concluding remarks.

2. Artificial bee colony (*ABC*) algorithm

The artificial bee colony algorithm is a population-based metaheuristic developed by Karaboga and Basturk [16,17] which is inspired by the intelligent foraging behaviour of honeybee swarm. The foraging bees are classified into three categories of employed, onlookers and scouts. All bees that are exploiting a food source are classified as "employed". The employed bees bring loads of nectar from the food source and share information with onlooker bees which are waiting in the hive for information to be shared by dance of employed bees about the food sources. The duration of a dance is proportional to the fruitfulness of the food source currently being exploited by the dancing bee. Onlooker bees tend to choose a food source according to the probability proportional to the quality of that food source. Therefore, good food sources attract more bees than the bad ones. Scout bees search for new food sources in the vicinity of the hive. Whenever a scout or onlooker bee finds a food source, it becomes employed. Whenever a food source is exploited fully, all the employed bees associated with it abandon it, and may again become scouts or onlookers. Scout bees perform the job of exploration, whereas employed and onlooker bees perform the job of exploitation.

In *ABC* algorithm, the position of a food source is a possible solution of the optimization problem and nectar amount of a food source corresponds to the fitness of associated solution. In *ABC*, the first half of the colony consists of employed bees and the other half includes onlookers [17]. The number of employed bees (*Nb*) is equal to the number of food sources (*SN*) because it is assumed that for every food source, there is only one employed bee [17]. Moreover, the number of employed or onlooker bees is equal to solutions in the population [16]. After generating a randomly distributed initial population of size *SN* of solutions, each of the employed and onlooker bees exerts a probabilistically modification on the solution (the position of a food source) for finding a new solution (new food source position) and tests the fitness (nectar) amount of this new solution (new food source). Suppose each solution consists of D parameters and let $X_i^t = (x_{i1}^t, x_{i2}^t, \ldots, x_{iD}^t)$ denotes to the ith solution generated in cycle t with parameter values $x_{i1}^t, x_{i2}^t, \ldots, x_{iD}^t$. In *ABC* algorithm, every employed bee produces a new solution $V_i^t = (v_{i1}^t, v_{i2}^t, \ldots, v_{iD}^t)$, in a D dimensional search space, from the old one (X_i^t) using a differential expression as follows:

$$v_{ij}^t = x_{ij}^t + \varphi_{ij}^t(x_{ij}^t - x_{kj}^t) \tag{1}$$

where $j \in \{1, 2, ..., D\}$, and k is selected randomly from $\{1, 2, ..., Nb\}$ such that $k \neq i$. φ_{ij}^t is a random variate scaling factor. If the fitness value of the new generated solution be better than the old one (the nectar amount of the new food source be higher than the old one) the bee forgets the old solution and memorizes the new one. Otherwise she keeps the position of the old solution.

When all employed bees have finished their searching process, they share the fitness (nectar) information of their solution (food sources) with the onlookers, each of whom selects a solution according to a probability proportional to the fitness value of that solution. Eq. (1) is employed again to generate a new solution by an onlooker bee based on the old solution in her memory and the selected one. If the fitness amount of the new solution is better than the old one, the bee memorizes the new position and forgets the old one. The probability value, p_i by which an onlooker bee chooses a food source is calculated as follows [16]:

$$p_i = \frac{\mathrm{fit}_i}{\sum_{j=1}^{\mathrm{SN}} \mathrm{fit}_j} \tag{2}$$

where fit_i is the fitness value of the solution i and SN is the number of food sources. To calculate a fitness value for a minimization problem, the following expression is employed:

$$\mathrm{fit}_i = \begin{cases} \dfrac{1}{1+f(X_i)} & \text{if } f(X_i) \geq 0 \\ 1 + \mathrm{abs}(f(X_i)) & \text{if } f(X_i) < 0 \end{cases} \tag{3}$$

where $f(X_i)$ is the cost value associated to X_i. The scout bees replace the food source whose nectar is abandoned by employed bees, with a new one. In *ABC* algorithm, if the quality of a solution cannot be improved after a predetermined number of cycles called "limit", the scout bee replaces the abandoned solution with a new random one. In such a condition, the new solution is constructed as follows [16]:

$$x_{ij} = x_{\min}^j + \mathrm{rand}(0, 1) \times (x_{\max}^j - x_{\min}^j) \tag{4}$$

where $x_{\min}^j$ and $x_{\max}^j$ are the lower and upper bounds on the value of the jth parameter, respectively.

3. A novel *ABC* algorithm for binary optimization (*DisABC*)

In this section, we propose a new approach to deal with binary optimization problems using *ABC*. In order to use *ABC* algorithm as a solution method for pure binary optimization problems, it is not possible to use the differential expression (Eq. (1)), directly. Because this expression works in the continuous space, while binary optimization problems are discrete. Therefore, we must adapt this equation such that it becomes applicable for binary search spaces.

To propose our binary version of *ABC*, called *DisABC* algorithm, we apply the concept of dissimilarity between binary vectors as a measure to quantify how far the two binary vectors are apart from each other. This measure can be used as an alternative to differential expression. Before giving the rationale of our approach, we need to have a brief introduction on the way of quantifying the degree of similarity/dissimilarity between binary structures.

3.1. Measuring similarity of binary structures

The binary feature vector is one of the most common ways of patterns representation, and measuring the similarity/dissimilarity of binary structures play a critical role in many problems such as clustering and classification [4]. Quantifying the degree of similarity between two objects, it can be found how similar they are to each other or how far apart they are from each other. There are several techniques for measuring similarity of objects with binary structures. A popular group of these measures applicable for binary data is known collectively as matching coefficients [6]. There are several types of matching coefficients, all of which take their goal as measuring similarity between any two structures composed of binary valued bits. The underlying logic is that two structures should be viewed as similar to the degree that they share a common pattern among their bits [9].

Let $X_i = (x_{i1}, x_{i2}, ..., x_{iD})$ and $X_j = (x_{j1}, x_{j2}, ..., x_{jD})$ represent two binary vectors for which we are interested to measure the degree of similarity between them (recall that x_{id} and x_{jd} $\forall d = 1, 2, ...D$, represent the dth bit in X_i and X_j, respectively and can take only 0 or 1 values).

To measure the similarity between X_i and X_j, their bit values should be compared. There are four possible cases:

$$x_{id} = x_{jd} = 1$$
$$x_{id} = 0, x_{jd} = 1$$
$$x_{id} = 1, x_{jd} = 0$$
$$x_{id} = x_{jd} = 0$$

Let define:

- M_{11} represents the total number of bits where both X_i and X_j have a value of 1 ($M_{11} = \sum_{d=1}^{D} I(x_{id} = x_{jd} = 1)$).
- M_{01} represents the total number of bits where the bit value for X_i is 0 and for X_j is 1 ($M_{01} = \sum_{d=1}^{D} I(x_{id} = 0, x_{jd} = 1)$).
- M_{10} represents the total number of bits where the bit value for X_i is 1 and for X_j is 0 ($M_{10} = \sum_{d=1}^{D} I(x_{id} = 1, x_{jd} = 0)$)
- M_{00} represents the total number of bits where both X_i and X_j have a value of 0 ($M_{00} = \sum_{d=1}^{D} I(x_{id} = 0, x_{jd} = 0)$)

Obviously we have $M_{11} + M_{01} + M_{10} + M_{00} = D$. One of the most commonly used measures to identify the degree of similarity between X_i and X_j is the Jaccard's coefficient of similarity defined as follows [21]:

$$Similarity(X_i, X_j) = \frac{M_{11}}{M_{01} + M_{10} + M_{11}} \quad (5)$$

The value of this measure belongs to interval [0,1]. Taking Jaccard's coefficient of similarity, an intuitive measure of dissimilarity between X_i and X_j, which quantifies how far apart X_i and X_j are from each other, can be defined as follows:

$$Dissimilarity(X_i, X_j) = 1 - Similarity(X_i, X_j) = 1 - \frac{M_{11}}{M_{01} + M_{10} + M_{11}} \quad (6)$$

Clearly $0 \le Dissimilarity(X_i, X_j) \le 1$. While there are a number of similarity metrics available for dichotomous variables [11], the Jaccard's coefficient of similarity is one of most widely used metric discussed in the literature. Therefore, without loss of generality, only this measure will be included in our study.

3.2. Generating a new solution in DisABC

As mentioned before, Eq. (1) in *ABC* algorithm has been designed for optimization in continuous space and cannot work with binary vectors. Therefore, to deal with pure binary optimization problems we should reconstruct (1) to obtain the one working with binary vectors instead of real valued vectors. To do this, some appropriate operators must be used in place of the arithmetic operators used in (1). Specially, we substitute "−" operator with a dissimilarity measure of binary vectors. Similar to "−" operator that quantifies the magnitude of difference between two scalars, a dissimilarity measure quantifies the magnitude of distance/dissimilarity between two binary vectors. One of the main characteristics of such a measure is that in spite of working in continuous space, its outcome can be used for construction of the new solution vector in binary space. We use "*Dissimilarity*" to address such measure.

Reshaping (1) in form of $V_i^t - X_i^t = \varphi(X_i^t - X_k^t)$ and replacing "−" by "*Dissimilarity*", the new differential expression in *DisABC* algorithm can be defined as follows:

$$Dissimilarity(V_i^t, X_i^t) \approx \varphi.Dissimilarity(X_i^t, X_k^t) \quad (7)$$

Recall that V_i^t, X_i^t and X_k^t are binary vectors and φ is a positive random scaling factor. In (7), "≈" implies "almost equal". We use this symbol, instead of "=", because it may not be possible to construct the new solution V_i^t, in such a way that the value of $Dissimilarity(V_i^t, X_i^t)$ becomes equal to the value of $\varphi.Dissimilarity(X_i^t, X_k^t)$, exactly. Let us define $A = \varphi.Dissimilarity(X_i^t, X_k^t)$, and assume that the value of A has been determined using (6).

Eq. (7) tells us that the construction of the new binary solution V_i^t is in such a way that its degree of dissimilarity with X_i^t is around the value of A. In other words, to produce the new binary solution V_i^t, the value of the following three variables must be determined:

- M_{11}: the number of bits with value 1 in both V_i^t and X_i^t
- M_{10}: the number of bits with value 1 in V_i^t and 0 in X_i^t
- M_{01}: the number of bits with value 0 in V_i^t and 1 in X_i^t

This work must be done in such a way that the value of $Dissimilarity(V_i^t, X_i^t)$ obtained by $1 - M_{11}/(M_{01} + M_{10} + M_{11})$ gets equal or the closest value to A. To determine the best possible value for M_{11}, M_{10} and M_{01}, the following integer programming model is used. As soon as these values are determined, the binary solution V_i^t can be constructed, accordingly. Let n_1 be the total number of bits with value 1 and n_0 be the total number of bits with value 0 in X_i^t, respectively.

$$\min \left| 1 - \frac{M_{11}}{M_{11} + M_{10} + M_{01}} - A \right| \quad (8)$$

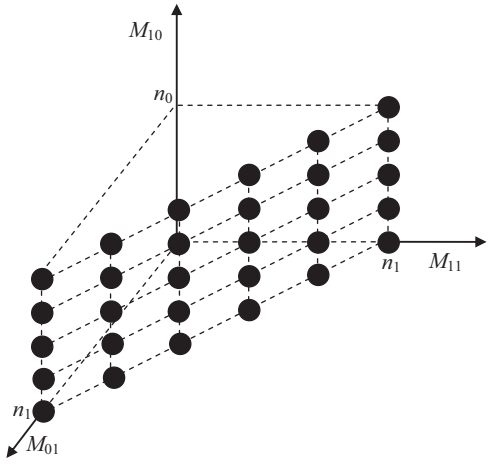

Fig. 1. The feasible space of the system (8)–(11).

st:

$$M_{11} + M_{01} = n_1 \tag{9}$$

$$M_{10} \le n_0 \tag{10}$$

$$M_{11}, M_{10}, M_{01} \ge 0 : \text{ and integer} \tag{11}$$

Objective (8) tries to minimize the gap between the value of $Dissimilarity(V_i^t, X_i^t)$ and A. In other words, it tries to keep (7) in balance. Since $M_{11} + M_{01}$ counts the number of bits where X_i^t has a value equal to 1, therefore constraint (9) enforces that this value must be equal to n_1. Constraint (10) ensures that the total number of bits where the bit value for V_i^t is 1 and for X_i^t is 0, is less than the total number of zeros in X_i^t, i.e., n_0. Solving the above mathematical model optimally, we can decide on the bit values of V_i^t. Therefore, a computationally advisable way should be devised to solve model (8)–(11) optimally. Fortunately, the structure of the above model allows us to solve it optimally through a total enumeration (*TE*) scheme by only $(n_1 + 1)(n_0 + 1)$ evaluations of the objective (8). This *TE* scheme requires at most $O(D^2)$ time.

With respect to constraint (9), the feasible set is $(M_{11}, M_{01}) = \{(i, n_1 - i) | i = 0, 1, \ldots, n_1\}$ with $n_1 + 1$ pairs. Taking each pair values (fixed values of M_{11} and M_{01}), the value of M_{10} can vary from 0 to n_0 (thus, there are $n_0 + 1$ different values for M_{10}). Therefore, the feasible state space of the system of (8)–(11) (the lattice in Fig. 1) is formed with $(n_1 + 1)(n_0 + 1)$ triplets as follows:

$$(M_{11}, M_{01}, M_{10}) = \{(i, n_1 - i, j) | i = 0, 1, \ldots, n_1, j = 0, 1, \ldots, n_0\}$$

Each triplet is represented by a black point in Fig. 1. Therefore, there would be required at most $(n_1 + 1)(n_0 + 1)$ evaluations of objective (8) to find the optimal solution of the model (8)–(11).

The new solution vector V_i^t can be simply constructed after getting the optimal value of M_{11}, M_{01} and M_{10} through *TE* scheme. Starting with a vector of size 1 by D of zeroes, the candidate solution V_i^t is obtained by: 1) choosing M_{11} number of zero bits for which the corresponding value in X_i^t takes 1 and changing their value in V_i^t from 0 to 1 and 2) choosing M_{10} number of zero bits within V_i^t for which the corresponding value in X_i^t takes 0 and changing their value in V_i^t from 0 to 1. Since V_i^t is initialized by zero vector, we just need to decide about the number of variables (bits) that their value should change to 1. The number of these bits is equal to $M_{10} + M_{11}$.

The complete vector V_i^t which is obtained based on the output of mathematical model (8)–(11) is the one whose dissimilarity value with X_i^t, in terms of Jaccard's coefficient of dissimilarity, is the closest possible value to A.

The following algorithm describes the required steps for generating the candidate solution V_i^t, via the parameter vectors X_i^t and X_k^t.

***Algorithm NBSG* (*new binary solution generator*)**

Step 1. Compute the value of A through $A = \varphi.Dissimilarity(X_i^t, X_k^t)$ *and use it in the mathematical programming model (8)–(11) with outputs* M_{11}, M_{01} *and* M_{10}. *Apply the total enumeration (TE) scheme to solve the mathematical programming problem optimally. Initialize* V_i^t *by a* $1 \times D$ *zero vector.*

Step 2-1 (*Inheritance phase*). *Based on any logic, select* M_{11} *number of zero bits from* V_i^t *which their corresponding value in* X_i^t *is* 1. *Change the value of the selected bits from* 0 *to* 1.

Step 2-2 (*Disinheritance phase*). *Based on any logic, select* M_{10} *number of zero bits from* V_i^t *which their corresponding value in* X_i^t *is* 0. *Change the value of the selected bits from* 0 *to* 1. *Then, report the new binary solution* V_i^t *as an output.*

At steps 2-1 and 2-2 of *NBSG*, general/problem-dependent heuristics can be employed. In step 2-1 of *NBSG*, one may do selection in a greedy fashion; for example based on the contribution that each bit (variable) might have in the cost function. Here is where we can use the structural knowledge of the problem to generate possibly better solutions. Indeed, the selected bits are one of the parts that the new solution V_i^t inherits from X_i^t. Usually *DROP* methods are applicable at step 2-1 of *NBSG* algorithm. Such methods would keep dropping the bit with value equal to 1 in X_i^t that gives the maximum decrease in the total cost, and would stop whenever the number of remained ones is equal to M_{11}. A similar situation is held for step 2-2 of *NBSG* algorithm, but here the selected elements are the parts that V_i^t disinherits. For this phase, *ADD* methods seem suitable. Such methods would keep adding the bit results in the maximum decrease in the total cost, and would stop whenever the number of added bits reaches M_{10}. The added bits will take 1 in V_i^t.

However, it should be remind that *DROP*/*ADD* methods perform on the basis of dropping/adding a variable after a variable and this may make the use of them computationally expensive. In our implementation of *DisABC*, to fulfil the inheritance and disinheritance steps in *NBSG* algorithm, we use one of random selection or selection based on the pattern observed in the best solution found so far (X_{Global}^t), in a probabilistic manner. Following the random selection logic, for the inheritance phase, M_{11} number of zero variables which their corresponding values in X_i^t is 1, are selected *randomly* from V_i^t (initialized by a $1 \times D$ zero vector) and their values are changed to 1. For the disinheritance phase, M_{10} number of zero variables which their corresponding values in X_i^t takes 0, are selected *randomly* from V_i^t and their values are changed to 1. To bias the search toward the components of X_{Global}^t, the second selection logic called *greedy* selection logic performs as follows: for the inheritance phase, M_{11} number of zero variables which their corresponding values in both X_i^t and X_{Global}^t are equal to 1, are selected from V_i^t (initialized by a $1 \times D$ zero vector) and their values are changed to 1. Let k be the number of variables with value 1 in both X_i^t and X_{Global}^t. If $M_{11} > k$ then the extra $M_{11} - k$ variables are selected based on the random selection logic described above. The disinheritance phase is done in a similar fashion but this time M_{10} number of zero variables which their corresponding values in both X_i^t and X_{Global}^t are equal to 0, are selected from V_i^t. The extra variables are selected based on the random selection logic, accordingly. To address both exploitation preserved by the greedy selection logic and exploration supported by the random selection logic, we employ them in a probabilistic manner. In this way, a random number in [0,1] is generated. If the value of random number be less than a predetermined value (p_s) the random selection logic is followed, otherwise the greedy logic is employed.

Example. To show how the candidate solution V_i^t is generated in *DisABC* algorithm, let us consider two parameter vectors $X_i^t = (1011010100)$ and $X_k^t = (1000101101)$. Let us also assume that $\varphi = 0.7$.

Step 1. Comparing X_i^t and X_k^t bit by bit, we find that $M_{11} = 2$, $M_{01} = 3$ and $M_{10} = 3$. Therefore we have:

$$A = \varphi.Dissimilarity(X_i^t, X_k^t) = 0.7\left(1 - \frac{2}{2+3+3}\right) = 0.525$$

Thus, V_i^t should be constructed in such a way that the value of $Dissimilarity(V_i^t, X_i^t)$ becomes around 0.525. The mathematical model of (8)–(11) can be given as follows. Recall that from X_i^t we have $n_1 = 5$ and $n_0 = 5$.

$$\begin{aligned} \min\ z = & \left| 1 - \frac{M_{11}}{M_{11}+M_{10}+M_{01}} - 0.525 \right| \\ & st: \\ & M_{11} + M_{01} = 5 \\ & M_{10} \le 5 \\ & M_{11}, M_{10}, M_{01} \ge 0: \text{ and integer} \end{aligned}$$

The optimal output of the model is as follows: $M_{11} = 3$, $M_{01} = 2$, $M_{10} = 1$ and $z = 0.025$. Initialize V_i^t by a 1×10 zero bit array.

Step 2-1. Let us define $P_{X_i^t} = \{d | x_{id}^t = 1\}$. We should select $M_{11} = 3$ bits among $P_{X_i^t} = \{1, 3, 4, 6, 8\}$. Assume that following the random selection, the first, the middle and the last member of $P_{X_i^t}$ are selected. Thus we have $V_i^t = (1001000100)$.
Step 2-2. Let us define $Q_{X_i^t} = \{d | x_{id}^t = 0\}$. We should select $M_{10} = 1$ variable among $Q_{X_i^t} = \{2, 5, 7, 9, 10\}$. Assume that following the random selection, the forth member of $Q_{X_i^t}$ is selected. Thus we set $v_{i9}^t = 1$ which yields the final solution as $V_i^t = (1001000110)$.

It is easy to check that $Dissimilarity(V_i^t, X_i^t) = 1 - (3/(3+1+2)) = 0.5$ which its difference with the value of A is the minimum possible value, i.e., 0.025.

3.3. Initialization in DisABC

The initial population in *DisABC* is generated randomly using a Bernoulli process. In this way, for each variable of an initial parameter vector, a random number within [0,1] is generated. If the value of this number is less than 0.5, the corresponding variable gets 0, otherwise it gets 1.

It is worth noting that the scot bee's solution is also generated randomly using the above procedure.

3.4. Hybridizing DisABC with a local search module

A local search is often a simple neighbourhood search method. It starts with an initial solution as the current solution and checks its neighbourhood for finding a better solution. If such solutions exist, then the local search designates the best solution found in the neighbourhood as the current solution and repeats the process [10].

A simple local search method using a neighbourhood structure based on swap moves is hybridized with *DisABC* algorithm and is employed after the onlooker phase. The swap move changes the value of a zero bit to 1 and simultaneously changes a bit with value of 1 to 0. This kind of move does not change the number of bits having value equal to 1 in the solution. The local search module has two input parameters, namely p_{local} and N_{local}. p_{local} controls the rate of recalling the local search module in *DisABC* algorithm. Also, N_{local} determines the number of generated and evaluated solutions in the local search module. The sketch of the local search module is as follows:

```
Local search method
Initialize input parameters
If r ≤ p_local % %r is a random number in [0,1]
  For i = 1 to N_local
  S ← X_i^t;
  Do a swap move on S;
  If f(S) < f(X_i^t)
    X_i^t ← S;
  End if
  End for
End if
```

The following algorithm describes the pseudo code of *DisABC* algorithm for minimizing a binary optimization problem with cost function f. We use $f(X)$ to denote the cost value associated with X.

```
DisABC Algorithm
Initialize the input parameters
t ← 1;
For i = 1 to SN
    Create a random solution X_i^t and evaluate it;
End for
While stopping criteria are not true
    For i = 1 to SN %% employed bee phase
      Generate a new solution V_i^t from X_i^t (and based on X_k^t (k ≠ i))
      via NBSG algorithm; Evaluate the new solution;
      If f(V_i^t) < f(X_i^t)
      X_i^t ← V_i^t
      Else
        Remember X_i^t;
      End if
    End for
    For i = 1 to SN %% onlooker bee phase
        Calculate the probability value p_i using (2);
        Produce a new solution V_i^t from X_i^t (and based on X_k^t
        (k ≠ i) selected depending on p_i) via NBSG
        algorithm; Evaluate the new solution;
          If f(V_i^t) < f(X_i^t)
            X_i^{t+1} ← V_i^t
          Else
              X_i^{t+1} ← X_i^t
          End if
      End for
      Apply the local search module;
      If necessary, select the abandoned solution and replace it
      with a new random solution generated in a manner similar
      to the procedure explained in Section 3.3; %% scot bee phase
      t ← t + 1;
End While
```

4. The uncapacitated facility location problem

This section gives a brief description on a concrete binary problem which is used to verify the effectiveness of *DisABC* algorithm. The problem is the uncapacitated facility location problem (*UFLP*). *UFLP* which has been extensively considered in the literature can be described as follows.

In *UFLP*, there is a set of customer locations with known demands and a set of candidate facility locations. If we select to locate a facility at a candidate location, a known fixed set up cost will be incurred. Moreover, there is a known shipment cost between each candidate facility location and each customer location. The problem is to find the optimal locations for setting up facilities and assignment of customers to located facilities in such a way that the sum of total opening costs and total shipment costs be minimize. It is assumed that the located facilities have sufficient capacity to meet all demands of the customer(s) connected to them [5]. Taking n and m to represent the potential number of facilities

and customers, respectively, *UFLP* can be mathematically stated as follows:

$$\min \ f = \sum_{i=1}^{n}\sum_{j=1}^{m} c_{ij}y_{ij} + \sum_{i=1}^{n} f_i x_i$$

$$st: \sum_{i=1}^{n} y_{ij} = 1, \quad j = 1, \ldots, m$$

$$y_{ij} \le x_i, \quad i = 1, \ldots, n, \quad j = 1, \ldots, m$$

$$y_{ij}, x_i = 0, 1, \quad i = 1, \ldots, n, \quad j = 1, \ldots, m$$

where c_{ij} is the shipment cost from facility location i to customer location j; f_i is the opening cost of a facility at location i;

$$y_{ij} = \begin{cases} 1 & \text{if customer } j \text{ is served by the facility} \\ & \text{opened at location } i \\ 0 & \text{otherwise} \end{cases}$$

and

$$x_i = \begin{cases} 1 & \text{if facility is opened at location } i \\ 0 & \text{otherwise} \end{cases}$$

Since the demand of each customer location is fulfilled by only one facility (i.e., no fractional fulfilment is allowed) the variable y_{ij} is of binary type. Variable x_i is also binary. $x_i = 1$, indicates the opening of a facility at candidate location i by incurring a fixed opening cost.

When the location of facilities to be opened is determined, the optimal allocation of customers will be obtained easily. Indeed, each customer j is fulfilled by the facility opened at location k whose shipment cost c_{kj} is minimal ($k = \arg \min_{i=1,\ldots,n} \{c_{ij}\}$). Then $y_{kj} = 1$ and $y_{ij} = 0, \forall i = 1, \ldots, n; i \neq k$. Therefore, this is the location decision that should be done optimally. In this regard, by a "solution" we address a vector $X = (x_1, x_2, \ldots, x_n)$ of n variables, where each variable (bit) x_i demonstrates that whether a facility is opened at location i or not. Such a binary vector plays the role of a parameter vector in *DisABC* algorithm.

UFLP is probably the most important *NP-hard* problem in location theory [5] and a rich literature has been devoted to it. Various solution methodologies have been emerged over the time for *UFLP*; among them the branch-and-bound method [12], linear programming and Lagrangian relaxation [3] and dual approach [8] are exact methods which ensure optimality. However, these methods may be computationally prohibited. Other methods are approximate methods which facilitate getting optimum or near optimum solutions in a reasonable time. Examples of these methods are tabu search [1,22], genetic algorithm [15], particle swarm optimization [20,24], neighbourhood search [10], etc.

To the best of our knowledge, this is the first effort investigating the application of an *ABC* type algorithm for solving *UFLP* test instances. Our main motivation to test the performance of *DisABC* on the *UFLP* test instances is due to the three facts: 1) *UFLP* is a pure binary optimization problem without having any continuous or integer decision variables, 2) any arbitrary binary vector X constitutes a feasible solution with respect to problem constraints. So, there is no need to consider penalties to eliminate infeasibilities, 3) several test problem instances of *UFLP* together with their known optimal solution are available.

5. Computational experiments

In this section vast computational experiments are conducted to test the performance of *DisABC* algorithm to solve the benchmarked instances of *UFLP*. One of the widely used set of test problem instances of *UFLP*, for which the optimal solutions are known, is the set of 15 test problems available in OR-Library [2]. Almost all methods developed for *UFLP* are tested first on these 15 test problem instances [22,20,24]. Among these test problems, 4 problems (Cap71–Cap74) are small size, 8 problems (Cap101–Cap104, Cap131–Cap134) are medium size and the other three problems (Cap*a*–Cap*c*) are large size problems. It is worth to mention that the problem titles are same as those originally used in OR-Library.

DisABC has a number of control parameters that affect its performance. These parameters are: φ, Nb, $t_{\max}$, p_s, limit, N_{local} and p_{local} which have been already introduced in different parts of the paper. It is common that the value of the "limit" parameter be controlled by the number of employed bees (Nb) and the search dimension [17]. In our experiments, the value of "limit" parameter is defined by $2.5 \times Nb \times n$, where n is the number of potential facility locations. Therefore, if the suitable size of the population of employed bees is determined, the proper value of the "limit" parameter will be obtained, accordingly.

The effect that different levels of control parameters have on the behaviour of *DisABC* algorithm, when solving 15 benchmark problems of *UFLP*, is investigated thoroughly during the next subsections. Results are summarized in Tables 1–6. In these tables, the effectiveness of *DisABC* algorithm under different settings for control parameters is measured by three indices. Columns captioned by "Gap (%)" report the average gap between the best cost values have been founded by *DisABC* (f^{DisABC}) and the optimal cost value ($f^{Optimal}$). The value of this index is calculated by:

$$GAP \ (\%) = AVG\left(\frac{f^{DisABC} - f^{Optimal}}{f^{Optimal}} \times 100\right) \tag{12}$$

where $AVG()$ is the simple averaging function. Each cell in the "Gap (%)" column reports the average percentage of the gap observed among 30 runs of the algorithm on the related instance.

Columns under the caption of "#EVL" report the average number of solutions generated and evaluated by *DisABC* algorithm, among 30 times execution. The algorithm stops when reaching the optimum cost ($f^{Optimal}$). Such a measure can gives an indication on the quality of *DisABC* convergence. The smaller the average number of solutions generated and evaluated by *DisABC* algorithm, indicates the faster convergence. Finally, columns under the caption of "#OPT" report the number of times that the optimal solution is reached by *DisABC* algorithm, among 30 replications. In the following subsections we try to find a proper value for the control parameters which led to a good performance of the algorithm.

5.1. On the choice of the value of φ

Our preliminary experiments show that choosing a dynamic setting for updating the value of φ such that it changes linearly throughout the search process is more appropriate than a fixed value. Hence, we allow the value of φ decreases linearly from an upper level ($\varphi_{\max}$) to a lower level ($\varphi_{\min}$) as follows:

$$\varphi^t = \varphi_{\max} - \left(\frac{\varphi_{\max} - \varphi_{\min}}{t_{\max}}\right) t \tag{13}$$

In which, t and $t_{\max}$ are the current cycle and the maximum number of cycles, respectively. Two levels are considered for $\varphi_{\max}$ (i.e., 0.5 and 0.9) and three levels are considered for $\varphi_{\min}$ (i.e., 0.1, 0.5 and 0.9). Thus, we have five different combinations of $\varphi_{\max}$ and $\varphi_{\min}$ such that $\varphi_{\max} \ge \varphi_{\min}$. These combinations are: $\varphi_{\max} = 0.5$ and $\varphi_{\min} = 0.1$; $\varphi_{\max} = 0.5$ and $\varphi_{\min} = 0.5$; $\varphi_{\max} = 0.9$ and $\varphi_{\min} = 0.1$; $\varphi_{\max} = 0.9$ and $\varphi_{\min} = 0.5$; $\varphi_{\max} = 0.9$ and $\varphi_{\min} = 0.9$.

Table 1 reports the results obtained by *DisABC* algorithm under above combinations of $\varphi_{\max}$ and $\varphi_{\min}$. Other control parameters are set as follows: $Nb = 10$; $t_{\max} = 2000$; $p_s = 0.5$; $p_{local} = 0.01$ and $N_{local} = 50$.

From the results of Table 1 it can be perceived that for small size problems, i.e., Cap71–Cap74, there is no significant difference between various combinations of $\varphi_{\max}$ and $\varphi_{\min}$, and

Table 1
Results obtained by *DisABC* under different levels of φ_{max} and φ_{min}.

Problem name	Problemsize	$\varphi_{max}=0.5$						$\varphi_{max}=0.9$					
		$\varphi_{min}=0.1$			$\varphi_{min}=0.5$			$\varphi_{min}=0.1$			$\varphi_{min}=0.5$			$\varphi_{min}=0.9$		
		GAP (%)	#EVL	#OPT	GAP (%)	#EVL	#OPT	GAP (%)	#EVL	#OPT	GAP (%)	#EVL	#OPT	GAP (%)	#EVL	#OPT
Cap71	16 × 50	0	672.6	30	0	1264.2	30	0	1601.7	30	0	3192.8	30	0	2538.8	30
Cap72	16 × 50	0	1020.7	30	0	1602.4	30	0	3250.0	30	0	2791.8	30	0	3260.7	30
Cap73	16 × 50	0	2927.1	30	0	1978.9	30	0	6591.4	30	0	7418.1	30	0	10062.0	30
Cap74	16 × 50	0	686.5	30	0	676.9	30	0	902.8	30	0	1037.8	30	0	1199.3	30
Cap101	21 × 50	0	8480	30	0	6752.6	30	0	6271.1	30	0	8843.9	30	3.60E−3	12848.6	29
Cap102	21 × 50	0	2155.6	30	0	4182.4	30	0	5995.9	30	0	8806.2	30	0	13601.1	30
Cap103	21 × 50	5.49E−3	15461.1	28	0	9649.2	30	0	6852.5	30	0	7492.5	30	0	4991.8	30
Cap104	21 × 50	0	1110.5	30	3.73E−3	6300.4	28	3.72E−3	6289.1	28	7.45E−3	8271.0	26	3.73E−3	7191.4	28
Cap131	50 × 50	6.20E−2	31991.3	13	3.70E−2	20861.9	21	1.44E−2	18764.0	26	7.22E−3	16004.5	28	6.38E−2	35874.2	16
Cap132	50 × 50	0	6016.6	30	0	4860.2	30	8.43E−3	11051.7	28	4.21E−3	14013.2	29	1.26E−2	23143.8	23
Cap133	50 × 50	7.35E−2	34625.2	10	1.12E−1	39446.2	7	4.48E−2	31351.2	13	3.27E−2	25086.0	18	5.32E−2	34579.9	12
Cap134	100 × 1000	0	2187	30	1.86E−3	4480.1	29	0	3584.3	30	1.86E−3	4478.9	29	2.64E−2	15469.9	23
Cap*a*	100 × 1000	0	6266.6	30	0	11756.8	30	0	6301.5	30	0	9866.8	30	0	10504.8	30
Cap*b*	100 × 1000	6.11E−1	32221.7	13	4.52E−1	32133.6	13	3.51E−1	37437.5	14	3.67E−1	35537.7	17	9.92E−1	48772.1	1
Cap*c*	100 × 1000	2.73E−1	43584.4	5	2.18E−1	48734.3	2	2.36E−1	45065.1	6	1.49E−1	46158.4	4	3.53E−1	48681.7	3
Average		6.83E−2	12627.1	24.6	5.50E−2	12978.7	24.7	4.39E−2	12754.0	25.7	3.79E−2	13266.7	26.1	1.01E−1	18181.3	23

Fig. 2. Plot of the mean of gap values evolved by *DisABC* under different combinations for upper and lower bound of φ.

DisABC can reach the optimum in all of 30 runs. However, as the value of φ_{max} and φ_{min} becomes larger, the required number of solutions generated to reach the optimum (#EVL) almost becomes large.

Analyzing the results of solving medium size benchmark instances (Cap101–Cap104), we can see that in general there is not a bold difference between different combinations. Since each combination fails to ensures optimality for at least one run. On Cap131–Cap134 the average pattern indicates that as the value of φ_{max} and φ_{min} becomes larger (except for $\varphi_{max}=0.9$ and $\varphi_{min}=0.9$), The average #EVL and #OPT values improves. However, $\varphi_{max}=0.5$ and $\varphi_{min}=0.1$ is the only combination which yields optimum in all runs for both Cap132 and Cap134. For these problems, $\varphi_{max}=0.9$ and $\varphi_{min}=0.9$ provides the worst, and the combination of $\varphi_{max}=0.9$ and $\varphi_{min}=0.5$ provides the best average results. On Cap*a*–Cap*c*, the algorithm is more effective under combination of $\varphi_{max}=0.9$ and $\varphi_{min}=0.1$ and $\varphi_{max}=0.9$ and $\varphi_{min}=0.5$.

In conclusion, from the "Average" row in Table 1 it can be inferred that, the greatest rate of getting optimal solutions (26.1 out of 30) and the lowest gap values (0.0388%) are achieved by *DisABC* under $\varphi_{max}=0.9$ and $\varphi_{min}=0.5$ combination. Considering the above illustrations, the combination of $\varphi_{max}=0.9$ and $\varphi_{min}=0.5$ is chosen for the rest of computations.

To visualize how *DisABC* converges to the global optimum, the evolution of the mean of gap values during the searches is depicted in Fig. 2 for Cap131 under different combinations of φ_{max} and φ_{min} values.

5.2. On the choice of the value of Nb and t_{max}

In this section the effect of the size of population of employed bees and/or the maximum number of cycles (t_{max}) on the quality of solutions obtained by *DisABC* algorithm is investigated.

Three levels of 10, 20 and 30 are considered for *Nb* (consequently, the size of population is 2×10, 2×20 and 2×30). Initial investigations show that *DisABC* algorithm is not very sensitive to different values of t_{max}. Therefore, instead of considering different values, only one reasonable level is considered for t_{max}, i.e., 2000. The other control parameters are set as follows: $\varphi_{max}=0.9$ $\varphi_{min}=0.5$; $p_s=0.5$; $N_{local}=50$ and $p_{local}=0.01$. Results are summarized in Table 2.

As expected, increasing the number of individuals results in increasing the chance of getting the optimal solution by the algorithm. But this is at the expense of increasing the number of evaluations. Almost for all problems, choosing a greater number

Table 2
Results obtained by *DisABC* under different levels of *Nb*.

Problem name	Problem size	*Nb* = 10			*Nb* = 20			*Nb* = 30		
		GAP (%)	#EVL	#OPT	GAP (%)	#EVL	#OPT	GAP (%)	#EVL	#OPT
Cap71	16 × 50	0	3192.8	30	0	2816	30	0	1791.3	30
Cap72	16 × 50	0	2791.8	30	0	5353.2	30	0	6621.9	30
Cap73	16 × 50	0	7418.1	30	0	12604.7	30	0	19311.9	30
Cap74	16 × 50	0	1037.8	30	0	695.2	30	0	848.3	30
Cap101	21 × 50	0	8843.9	30	0	15421.4	30	0	22723.1	30
Cap102	21 × 50	0	8806.2	30	0	16576.3	30	0	17784.2	30
Cap103	21 × 50	0	7492.5	30	0	9309.0	30	0	5503.7	30
Cap104	21 × 50	7.45E−3	8271.0	26	0	1529.2	30	0	1487.3	30
Cap131	50 × 50	7.22E−3	16004.5	28	0	23628.3	30	0	34544.9	30
Cap132	50 × 50	4.21E−3	14013.2	29	0	14169.1	30	0	21930.5	30
Cap133	50 × 50	3.27E−2	25086.0	18	3.22E−2	51670.6	17	1.84E−2	53659.9	23
Cap134	50 × 50	1.86E−3	4478.9	29	0	4963.3	30	0	3183.4	30
Cap*a*	100 × 1000	0	9866.8	30	0	11471.5	30	0	12295.3	30
Cap*b*	100 × 1000	3.67E−1	35537.7	17	2.98E−1	72900.8	16	2.51E−1	84505.4	19
Cap*c*	100 × 1000	1.49E−1	46158.4	4	1.01E−1	87067.6	7	5.95E−2	139156.7	3
Average		3.79E−2	13266.7	26.1	2.87E−2	22011.7	26.7	2.19E−2	28356.5	27

Table 3
Results obtained by *DisABC* under different levels of p_s.

Problem name	Problem size	p_s = 0.5			p_s = 1		
		GAP (%)	#EVL	#OPT	GAP (%)	#EVL	#OPT
Cap71	16 × 50	0	1791.3	30	0	9201.3	30
Cap72	16 × 50	0	6621.9	30	0	3476	30
Cap73	16 × 50	0	19311.9	30	0	3476.1	30
Cap74	16 × 50	0	848.3	30	0	1149.4	30
Cap101	21 × 50	0	22723.1	30	0	17246.2	30
Cap102	21 × 50	0	17784.2	30	0	11794.7	30
Cap103	21 × 50	0	5503.7	30	0	8655.2	30
Cap104	21 × 50	0	1487.3	30	0	3526.0	30
Cap131	50 × 50	0	34544.9	30	0	56667.9	30
Cap132	50 × 50	0	21930.5	30	0	44876.9	30
Cap133	50 × 50	1.84E−2	53659.9	23	0	39470.4	30
Cap134	50 × 50	0	3183.4	30	0	8175.1	30
Cap*a*	100 × 1000	0	12295.3	30	0	34148.9	30
Cap*b*	100 × 1000	2.51E−1	84505.4	19	3.54E−1	137507.4	9
Cap*c*	100 × 1000	5.95E−2	139156.7	3	7.40E−2	140530.4	11
Average		2.19E−2	28356.5	27	2.85E−2	34660.1	27.3

of employed bees, results in improving the quality of solutions obtained by *DisABC*. This issue can be indicated by the GAP (%) and #OPT values. To guarantee a reasonable quality under a reasonable amount of searches, we use *Nb* = 30 in rest of our experiments. Fig. 3 depicts the evolution of the mean of gap values under different values of *Nb* for Cap133.

5.3. On the choice of the value of p_s

To preserve both exploration and exploitation, random and greedy selection logics are used in the inheritance and disinheritance steps of *NBSG* algorithm. To make a balance between these strategies, a tunable parameter p_s, which affects the rate of

Table 4
Results obtained by *DisABC* under different levels of p_{local} and N_{local}.

Problem name	Problem size	p_{local} = 0.01; N_{local} = 50			p_{local} = 0.02; N_{local} = 100		
		GAP (%)	#EVL	#OPT	GAP (%)	#EVL	#OPT
Cap71	16 × 50	0	9201.3	30	0	2254.9	30
Cap72	16 × 50	0	3476	30	0	1458.8	30
Cap73	16 × 50	0	3476.1	30	0	3113.4	30
Cap74	16 × 50	0	1149.4	30	0	1255.3	30
Cap101	21 × 50	0	17246.2	30	0	11274.1	30
Cap102	21 × 50	0	11794.7	30	0	7159.9	30
Cap103	21 × 50	0	8655.2	30	0	5678.2	30
Cap104	21 × 50	0	3526.0	30	0	3240.4	30
Cap131	50 × 50	0	56667.9	30	0	45257.4	30
Cap132	50 × 50	0	44876.9	30	0	30261.6	30
Cap133	50 × 50	0	39470.4	30	0	23778.2	30
Cap134	50 × 50	0	8175.1	30	0	7195.8	30
Cap*a*	100 × 1000	0	34148.9	30	0	21340.3	30
Cap*b*	100 × 1000	3.54E−1	137507.4	9	0	87574.3	30
Cap*c*	100 × 1000	7.40E−2	140530.4	11	1.86E−2	185759.5	13
Average		2.85E−2	34660.1	27.3	1.24E−3	29106.8	28.9

Table 5
Results obtained by *DisABC* in comparison with *binDE*.

Problem name	Problem size	*DisABC*				*binDE*			
		GAP (%)	#EVL	# OPT	Time (s)	GAP (%)	# EVL	# OPT	Time (s)
Cap71	16 × 50	0	2254.9	30	3.1	0	2379.1	30	2.5
Cap72	16 × 50	0	1458.8	30	1.8	0	2046	30	2.2
Cap73	16 × 50	0	3113.4	30	3.6	0	2500.0	30	2.7
Cap74	16 × 50	0	1255.3	30	1.3	0	2737.3	30	2.9
Cap101	21 × 50	0	11274.1	30	17.7	0	8298.1	30	9.5
Cap102	21 × 50	0	7159.9	30	9.7	0	8192.5	30	9.2
Cap103	21 × 50	0	5678.2	30	7.2	0	8170.9	30	9.3
Cap104	21 × 50	0	3240.4	30	4.0	0	5910.7	30	6.6
Cap131	50 × 50	0	45257.4	30	73.6	0	37290.7	30	48.0
Cap132	50 × 50	0	30261.6	30	42.3	0	37727.5	30	47.5
Cap133	50 × 50	0	23778.2	30	30.5	0	44593.7	30	54.2
Cap134	50 × 50	0	7195.8	30	9.4	0	19002.7	30	23.5
Cap*a*	100 × 1000	0	21340.3	30	86.8	7.32E−01	100784.4	18	402.1
Cap*b*	100 × 1000	0	87574.3	30	378.3	1.77	120120	0	524.0
Cap*c*	100 × 1000	1.86E−2	185759.5	13	886.4	1.71	120120	0	555.9
Average		1.24E−3	29106.8	28.9	103.7	0.2	34658.2	25.2	113.3

Table 6
Results obtained by *DisABC* in comparison with *PSO*.

Problem name	Problem size	*DisABC*		*PSO*		*PSO + Local search*	
		GAP (%)	#OPT	GAP (%)	#OPT	GAP (%)	#OPT
Cap71	16 × 50	0	30	0.05	26	0	30
Cap72	16 × 50	0	30	0.07	24	0	30
Cap73	16 × 50	0	30	0.06	19	0	30
Cap74	16 × 50	0	30	0.07	22	0	30
Cap101	21 × 50	0	30	0.14	16	0	30
Cap102	21 × 50	0	30	0.15	12	0	30
Cap103	21 × 50	0	30	0.16	6	0	30
Cap104	21 × 50	0	30	0.18	21	0	30
Cap131	50 × 50	0	30	0.75	2	0	30
Cap132	50 × 50	0	30	0.78	0	0	30
Cap133	50 × 50	0	30	0.73	0	0	30
Cap134	50 × 50	0	30	0.89	3	0	30
Capa	100 × 1000	0	30	22.01	0	0	30
Capb	100 × 1000	0	30	10.75	0	0	30
Capc	100 × 1000	1.86E−2	13	9.72	0	0.02	15
Average		1.24E−3	28.9	3.1	10.0	1.33E−03	29

applying the two selection strategies, is introduced. In *DisABC* algorithm, two levels for value of p_s, namely 0.5 and 1 are considered. $p_s = 0.5$ puts equal weight on the greedy (exploitation) and random (exploration) selection strategy, while $p_s = 1$ puts entire weight on the random selection strategy (exploration). Other control parameters are set as follows: $Nb = 30$; $t_{max} = 2000$; $\varphi_{max} = 0.9$; $\varphi_{min} = 0.5$; $N_{local} = 50$ and $p_{local} = 0.01$. Results are presented in Table 3.

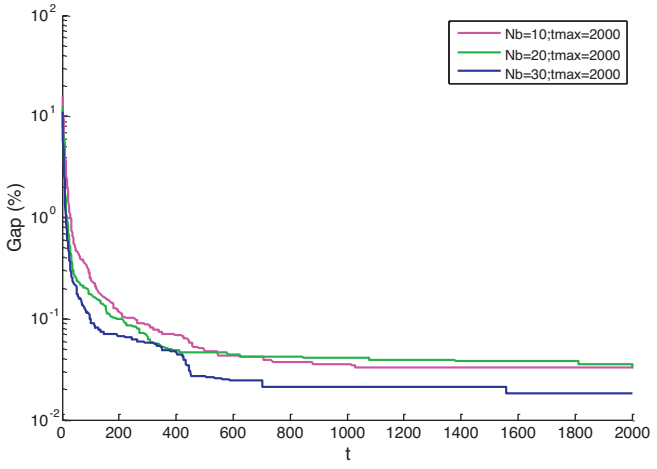

Fig. 3. Plot of the mean of gap values evolved by *DisABC* under different values for *Nb*.

From the results of Table 3 it can be observed that, $p_s = 1$ ensures optimality for 13 out of 15 (87%) problems, while this record for $p_s = 0.5$ is 12 out of 15 (80%). However there is no direct surpassing relation between these settings on large problems. Effectiveness of algorithm on Cap*b*, under $p_s = 0.5$ seems better in comparison with $p_s = 1$. But on Cap*c*, choosing $p_s = 1$ provides more fruitful results. In the absence of any significant evidence of superiority between the two considered schemes for p_s, we use $p_s = 1$ for the reminder of computations. Fig. 4 shows the evolution of the mean of gap values under different levels of p_s for Cap74. As it is expected, due to the nature of greedy selection which utilizes the information along with the best solution found so far, the convergence under $p_s = 0.5$ is faster than the pure random selection preserved by $p_s = 1$. The lower value of #EVL also indicates this conclusion.

5.4. On the choice of the value of N_{local} and p_{local}

In this section we will investigate the effect of setting different levels for the input parameters of local search module on the performance of *DisABC* algorithm. Two schemes are considered for N_{local} and p_{local} namely, $p_{local} = 0.01$ and $N_{local} = 50$ (the first scheme); and $p_{local} = 0.02$ and $N_{local} = 100$ (the second scheme). In the second scheme, both of the recalling rate of local search module

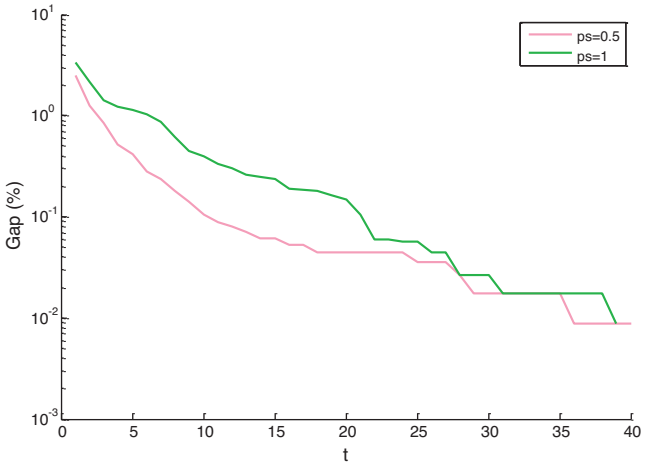

Fig. 4. Plot of the mean of gap values evolved by *DisABC* under different values for p_s.

(p_{local}) and the number of generated and evaluated neighbourhood solutions (N_{local}) are twice to bold the role of local search. Other control parameters are set as follows: $Nb = 30$, $t_{\max} = 2000$, $\varphi_{\max} = 0.9$, $\varphi_{\min} = 0.5$ and $p_s = 1$. Results are tabulated in Table 4.

From Table 4 it can be observed that the benefits along with more highlighting the role of the local search module in *DisABC* algorithm, in terms of reducing the Gaps, increasing the success rate and speeding up the convergence (see Fig. 5 for an example) are meaningful. On Cap*b*, the percentage of success in reaching the optimum under first scheme is less than 30% (9 out of 30). However, under second scheme the success rate hits 100% (30 out of 30). For Cap*c*, both of Gap (%) and #OPT indices are improved under second scheme. However there is an increment in the total number of solutions evaluated by the algorithm. In general, effectiveness of the algorithm in solving large scale problems will be better supplied under second scheme.

Fig. 5 visualizes the evolution of the mean of gap values under two settings of local search input parameters for Cap133.

5.5. Comparing DisABC with binDE

The *binDE* algorithm [7] uses the floating-point component of a solution vector to determine a probability for each component. These probabilities are then used to generate a bit-string solution from the floating-point vector. This bit string is used by the fitness function to determine its quality. The resulting fitness is then associated with floating point representation of the individuals [7].

To have a fair comparison between *DisABC* and *binDE* algorithms, the size of population and maximum number of cycles are considered equal. Moreover, we use a decreasing strategy to update the value of F (the scaled factor used in *DE*) same as that of used for updating the value of φ in *DisABC*. The value of CR (the crossover rate used in *DE*) is also set at 0.1 as it is typically suggested in *DE*. However, we have tried other levels of CR (e.g., 0.05 and 0.15) to possibly obtain better results. From the preliminary computations we found out that $CR = 0.1$ provides the best results.

The performance of *DisABC* and *binDE* algorithms, in solving 15 benchmark test instances of the *UFLP*, are compared in Table 5.

Comparing the results obtained by *DisABC* and *binDE* it can be inferred that both of them are quite successful in reaching the optimum of small and medium size problems (Cap71–Cap134). In most cases (except for Cap73, Cap 101 and Cap131) the convergence rate of *DisABC* is faster than *binDE*. This issue can be evidenced by the value of #EVL index obtained by the algorithms. On Cap*a*, the number of optimum solutions obtained by *binDE* is 18 out of 30 (60%) while *DisABC* reaches the optimum in all of 30 runs. On Cap*b* and Cap*c*, the difference between performance of *DisABC* and *binDE* is very egregious. Percentage of success of *DisABC* in getting the optimum of these problems is 100 and 43 respectively, while *binDE* could never reach the optimum. In conclusion, superiority of *DisABC* over *binDE* in solving large size test problem instances of *UFLP* is quite tangible. This issue can be confirmed by the value of all indices in "Average" row of Table 5.

Due to the representational restriction, we cannot use our local search module in the body of *binDE* to improve its performance. In order to improve a floating-point vector generated in an iteration of *binDE*, first the vector should be transformed to its corresponding bit-string vector and then the local search should be applied on it. Now the improved bit-string should be encoded into the floating-point vector corresponding to it (because *binDE* applies the evolutionary operators on the floating-point vectors). The restriction is due to the fact that we cannot determine the corresponding real vector of a given bit-string vector.

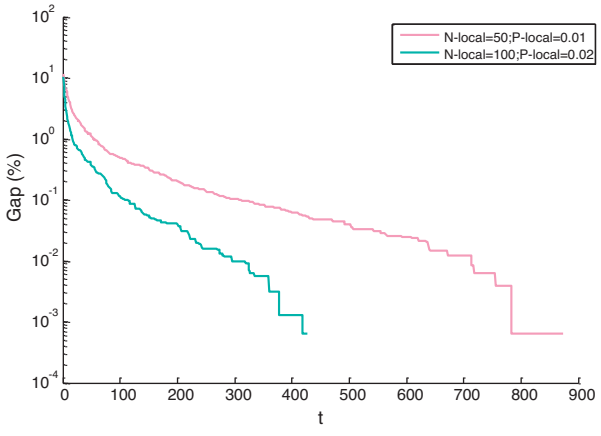

Fig. 5. Plot of the mean of gap values evolved by *DisABC* under different values for N_{local} and P_{local}.

5.6. Comparing DisABC with PSO

Similar to *binDE*, the *PSO* algorithm [20] uses the floating point vector representation. The only difference between *binDE* and *PSO* is in conversion of a continuous component to a binary one. Unlike *binDE*, *PSO* uses each floating-point component of a solution vector to identify the opening or closing a facility in a deterministic manner. In this algorithm each continuous component x is converted to binary value y using the formula $y = \lfloor |x \bmod 2| \rfloor$.

We adopt the results of *PSO* algorithm and its hybridized version (*PSO + Local search*) directly from [20] and reported them in Table 6. Results obtained by *DisABC* algorithm have been also restated in the same table.

From Table 6 we can infer that *PSO* produces premature results and does not offer satisfactory performance. The gap value for *PSO* on Cap*a*, Cap*b* and Cap*c* is very high and none of its attempts yields the optimum. The average gap value for *PSO* is 3.1% while for *DisABC* this value is 0.001%. Besides, the average #OPT values tell us that *DisABC* is more dependable than *PSO* in hitting the optimum. The performance of *PSO + Local search* algorithm looks very impressive compared to *PSO* with respect to the two indices of solution quality. However, almost for all problems the performance of *PSO + Local search* and *DisABC* algorithms are comparable.

6. Conclusion

Many real world engineering problems are stated mathematically as a binary optimization problem. Therefore, devising a suitable solution method for these problems seems noteworthy. We introduced a new binary version of Artificial Bee Colony (*ABC*) algorithm, which is called *DisABC* algorithm and uses a measure of dissimilarity between binary structures in place of the arithmetic vector subtraction operation, typically used in the continuous version of *ABC*.

One of the most important characteristics of our algorithm is that it works in continuous space, while the consequences are used to construct the new solution in the binary space. The effectiveness of the proposed approach was tested on benchmark test problem instances of the uncapacitated facility location problem (*UFLP*), and compared with two binary optimization algorithms, e.g., *binDE* and *PSO*, where the results demonstrate that our approach is competitive.

One of the main advantages of *DisABC* algorithm is that, unlike transformation based method, it does not map the problem space into another space and therefore, there is no lost of information. Ease of implementation and preserving the main characteristics of *ABC* algorithm are the other advantages of *DisABC* algorithm.

For future research, the effectiveness of our approach could be examined on other binary problems, e.g., knapsack problem, bin packing problem, etc. Developing the binary version of other well-known algorithms, which use differential operation, e.g., particle swarm optimization algorithm and league championship algorithm [13,14] is particularly encouraged.

References

[1] K.S. Al-Sultan, M.A. Al-Fawzan, A tabu search approach to the uncapacitated facility location problem, Annals of Operations Research 86 (1999) 91–103.
[2] L. Bao, J-c. Zeng, Comparison and analysis of the selection mechanism in the artificial bee colony Algorithm, Ninth International Conference on Hybrid Intelligent Systems (2009) 411–416.
[3] J. Barcelo, A. Hallefjord, E. Fernandez, K. Jrnsten, Lagrangian relaxation and constraint generation procedures for capacitated plant location problems with single sourcing, OR Spectrum 12 (1990) 78–79.
[4] S.-S. Choi, S.-H. Cha, C. Tappert, A survey of binary similarity and distance measures, Journal of Systematics, Cybernetics and Informatics 8 (1) (2010) 43–48.
[5] M.S. Daskin, L.V. Snyder, R.T. Berger, Facility Location in Supply Chain Design, Lehigh University, 2003, working paper No. 03-010.
[6] W.R. Dillon, M. Goldstein, Multivariate Analysis: Methods and Applications, John Wiley and Sons, 1984.
[7] A.P. Engelbrecht, G. Pampara, Binary differential evolution Strategies, IEEE Congress on Evolutionary Computation (2007) 1942–1947.
[8] D. Erlenkotter, A dual-based procedure for uncapacitated facility location, Operations Research 26 (6) (1978) 992–1009.
[9] H. Finch, Comparison of distance measures in cluster analysis with dichotomous data, Journal of Data Science 3 (1) (2005) 85–100.
[10] D. Ghosh, Neighborhood search heuristics for the uncapacitated facility location problem, European Journal of Operational Research 150 (1) (2003) 150–162.
[11] S. Hands, B. Everitt, A Monte Carlo study of the recovery of cluster structure in binary data by hierarchical clustering techniques, Multivariate Behavioral Research 22 (2) (1987) 235–243.
[12] K. Holmberg, Exact solution methods for uncapacitated location problems with convex transportation costs, European Journal of Operational Research 114 (1) (1999) 127–140.
[13] A. Husseinzadeh Kashan, League Championship Algorithm: a new algorithm for numerical function optimization, in: IEEE International Conference of Soft Computing and Pattern Recognition, SoCPaR 2009, 2009, pp. 43–48.
[14] A. Husseinzadeh Kashan, An efficient algorithm for constrained global optimization and application to mechanical engineering design: League championship algorithm (LCA), Computer-Aided Design (2011), doi:10.1016/j.cad.2011.07.003.
[15] J.H. Jaramillo, J. Bhadury, R. Batta, On the use of genetic algorithms to solve location problems, Computers & Operations Research 29 (6) (2002) 761–779.
[16] D. Karaboga, B. Basturk, A powerful and efficient algorithm for numerical function optimization: artificial bee colony (ABC) algorithm, Journal of Global Optimization 39 (3) (2007) 459–471.
[17] D. Karaboga, B. Basturk, On the performance of artificial bee colony (ABC) algorithm, Applied Soft computing 8 (1) (2008) 687–697.
[18] D. Karaboga, C. Ozturk, A novel clustering approach: artificial bee colony (ABC) algorithm, Applied Soft Computing 11 (1) (2011) 652–657.
[19] Q.K. Pan, M. Fatih Tasgetiren, P.N. Suganthan, T.J. Chua, A discrete artificial bee colony algorithm for the lot-streaming flow shop scheduling problem, Information Sciences 181 (12) (2011) 2455–2468.
[20] M. Sevkli, A.R. Guner, A continuous particle swarm optimization algorithm for uncapacitated facility location problem, ANTS 2006 (2006) 316–323.
[21] P.H.A. Sneath, Some thoughts on bacterial classification, Journal of General Microbiology 17 (1) (1957) 184–200.
[22] M. Sun, Solving the uncapacitated facility location problem using tabu search, Computers & Operations Research 33 (9) (2006) 2563–2589.
[23] P.-W. Tsai, J.-S. Pan, B.-Y. Liao, S.-C. Chu, Enhanced artificial bee colony optimization, International Journal of Innovative 5 (12B) (2009) 5081–5092.
[24] D. Wang, C.H. Wu, A. Ip, D. Wang, Y. Yan, Parallel multi-population particle swarm optimization algorithm for the uncapacitated facility location problem using openMP, IEEE Congress on Evolutionary Computation (2008) 1214–1218.
[25] C. Zhang, D. Ouyang, J. Ning, An artificial bee colony approach for clustering, Expert Systems with Applications 37 (7) (2010) 4761–4767.