Operational Research (2020) 20:2555–2581
https://doi.org/10.1007/s12351-018-0427-9

ORIGINAL PAPER

Weighted superposition attraction algorithm for binary optimization problems

Adil Baykasoğlu[1] · **Fehmi Burcin Ozsoydan**[1] · **M. Emre Senol**[1]

Received: 27 March 2018 / Revised: 27 August 2018 / Accepted: 10 September 2018 /

Abstract

Weighted superposition attraction algorithm (WSA) is a new generation population-based metaheuristic algorithm, which has been recently proposed to solve various optimization problems. Inspired by the superposition of particles principle in physics, individuals of WSA generate a superposition, which leads other agents (solution vectors). Alternatively, based on the quality of the generated superposition, individuals occasionally tend to perform random walks. Although WSA is proven to be successful in both real-valued and some dynamic optimization problems, the performance of this new algorithm needs to be examined also in stationary binary optimization problems, which is the main motivation of the present study. Accordingly, WSA is first designed for stationary binary spaces. In this modification, WSA does not require any transfer functions to convert real numbers to binary, whereas such functions are commonly used in numerous approximation algorithms. Moreover, a step sizing function, which encourages population diversity at earlier iterations while intensifying the search towards the end, is adopted in the proposed WSA. Thus, premature convergence and local optima problems are attempted to be avoided. In this context, the contribution of the present study is twofold: first, WSA is modified for stationary binary optimization problems, secondarily, it is further enhanced by the proposed step sizing function. The performance of the modified WSA is examined by using three well-known binary optimization problems, including uncapacitated facility location problem, 0–1 knapsack problem and a natural extension of it, the set union knapsack problem. As demonstrated by the comprehensive experimental study, results point out the efficiency of the proposed WSA modification in binary optimization problems.

Keywords Weighted superposition attraction algorithm · Binary optimization · Uncapacitated facility location problem · 0–1 Knapsack problem · Set union knapsack problem

---

✉ Adil Baykasoğlu
adil.baykasoglu@deu.edu.tr; baykasoglu@gmail.com
http://web.deu.edu.tr/baykasoglu

Extended author information available on the last page of the article

1 Introduction

As recently introduced by Baykasoğlu and Akpinar (2015, 2017), Weighted Superposition Attraction algorithm (WSA) is a novel swarm intelligence-based metaheuristic algorithm, proposed to solve real-valued constrained and unconstrained optimization problems. WSA draws inspiration from the superposition of particles principle in physics. Solution vectors in WSA generate a superposition, which is followed by some of the agents (solution vectors), depending on the quality of the generated superposition.

As mentioned by Baykasoğlu and Akpinar (2015, 2017), WSA differs from other well-known methods including Genetic Algorithm (GA) (Holland 1975), Differential Evolution Algorithm (DE) (Storn and Price 1997), Ant Colony Optimization (ACO) (Dorigo et al. 1991), Particle Swarm Optimization (PSO) (Kennedy and Eberhart 1995) and new generation metaheuristics such as Firefly Algorithm (FA) (Yang 2008), Cuckoo Optimization Algorithm (Rajabioun 2011) and Bat Algorithm (BA) (Yang 2010) in several ways. One of the most striking differences of WSA is the utilization of anonymous intelligence of the agents. As to be clarified later, WSA generates a target point (superposition) by making use of the current population information. This target point is referred to as the superposition, which directs the agents towards its coordinates based on the quality (fitness) of the generated superposition. That is to say, superposition is an anonymous composition of the currently discovered points so far. Once the superposition is determined, agents explore the search space either by moving towards that point or by performing random walks.

The related literature includes some studies of WSA for a variety of optimization problems. In one of them, Baykasoğlu and Akpinar (2015) proposed making use of this new generation metaheuristic in constrained design optimization problems. In another work of the authors, Baykasoğlu and Akpinar (2017) employed WSA to solve real-valued unconstrained optimization problems including well-known mathematical functions. Resource constrained scheduling problem, which is a challenging combinatorial optimization problem, is solved by Baykasoğlu and Şenol (2016a) using WSA. A similar approach is applied to the Travelling Salesman Problem, which is another well-known challenging problem (Baykasoğlu and Şenol 2016b). The authors reported promising results, which are compared to the results of several state-of-the-art algorithms. Özbakır and Turna (2017) examined the performance of WSA in clustering problems. In the same study, the authors also introduced two new modifications of WSA, where in the former one, contributions of lower quality agents are eliminated and in the latter one, superposition is directly used as the best individual of the population. According to the reported results, both modifications of WSA and its canonical version show notable performance particularly in continuous and categorical data. Finally, Baykasoğlu and Ozsoydan (2018) tested WSA in dynamic optimization problems. According to the reported results, WSA is found as a competitive and a promising algorithm also in dynamically changing problems.

Although WSA has been shown to be successful in various optimization problems (Baykasoğlu and Akpinar 2015, 2017; Baykasoğlu and Şenol, 2016a, b;

Özbakır and Turna 2017; Baykasoğlu and Ozsoydan 2018), the performance of WSA needs to be examined also in fundamental binary optimization problems, which is the main motivation of the present work.

In this respect, WSA is modified for binary spaces first. In the proposed modification, WSA does not require any transfer functions to convert real numbers to binary, whereas such functions are commonly used in approximation algorithms that are adapted to binary problems. Moreover, by making use of this approach, any type of variables can also be used along with binary variables. Additionally, a discrete step sizing function, which encourages population diversity at earlier iterations while intensifying the search towards the end, is adopted in the proposed WSA modification. Thus, premature convergence and local optima problems are attempted to be avoided.

Finally, the performance of the proposed binary WSA (bWSA) is examined by using some well-known binary optimization problems, including uncapacitated facility location problem (UFLP), 0–1 knapsack problem (0–1 KP) and the set union knapsack problem (SUKP). The obtained results are compared to the previously published results taken from the related literature. Comprehensive experimental study points out the superiority of bWSA in binary optimization problems.

The rest of the paper is organized as follows: benchmarking problems are formally defined in Sect. 2, detailed explanations of WSA and bWSA are introduced in Sect. 3. Finally, experimental study and concluding remarks are presented in Sects. 4 and 5, respectively.

2 Benchmarking problems

2.1 Uncapacitated facility location problem

Location problems have attracted notable attention from researchers because they have numerous applications in real world (Laporte et al. 2015). UFLP is a well-known type of location problems, where each customer is served by exactly one facility and facilities are assumed to have unlimited capacities (Cornuejols et al. 1990). Given a possible set of sites for establishing facilities and a set of customers at demand points, the aim here is to find the optimal locations for facilities to meet all demand such that the sum of facility opening costs and occurred shipment costs is minimized. UFLP can formally be formulated as given in the following (Eqs. 1–5).

$$minimize \sum_{k\in K}\sum_{l\in L} c_{kl}z_{kl} + \sum_{k\in K} b_k y_k \tag{1}$$

$$\text{s.t.} \sum_{k\in K} z_{kl} = 1, \quad l \in L \tag{2}$$

$$z_{kl} \le y_k, \quad k \in K \text{ and } l \in L. \tag{3}$$

$$z_{kl} \in \{0, 1\}, \quad k \in K \text{ and } l \in L \tag{4}$$

$$y_k \in \{0, 1\}, \quad k \in K \tag{5}$$

where $K = \{1, 2, \ldots, n\}$ is the possible locations for opening facilities, $L = \{1, 2, \ldots, m\}$ is the set of customers (demand points), b_k is the cost of establishing a facility at the kth location,c_{kl} is the shipment cost between the facility opened at the kth location and the lth customer, z_{kl} is a binary variable denoting whether the demand of the lth customer is fulfilled by the kth facility location ($z_{kl} = 1$ if the lth customer is served by the facility opened at the kth location, $z_{kl} = 0$ otherwise) and y_k is another binary variable representing the status of a facility location ($y_k = 1$ if a facility is opened at the kth location, $y_k = 0$ otherwise). In this context, Eq. 1 represents the objective function to minimize the total cost. Equation 2 satisfies that each customer can exactly be served by only one facility. Equation 3 provides that customers can only be served by the opened facilities. Finally, Eqs. 4–5 impose restrictions on decision variables. An illustration for this problem is presented in Fig. 1.

There are several exact solution techniques proposed for UFLP including a dual-based procedure (Erlenkotter 1978), a Lagrange relaxation (Barcelo et al. 1990), and a branch-and-bound method (Holmberg 1999). However, UFLP is shown to be an NP-hard problem and hence the solution time grows exponentially with the problem size. Therefore, making use of such exact solution approaches is limited only for small sized instances. In this regard, although exact solution methods guarantee optimality, they unfortunately have such limitations. On the other hand, metaheuristic algorithms do not guarantee optimality, however, they are easy to implement to any size of problem.

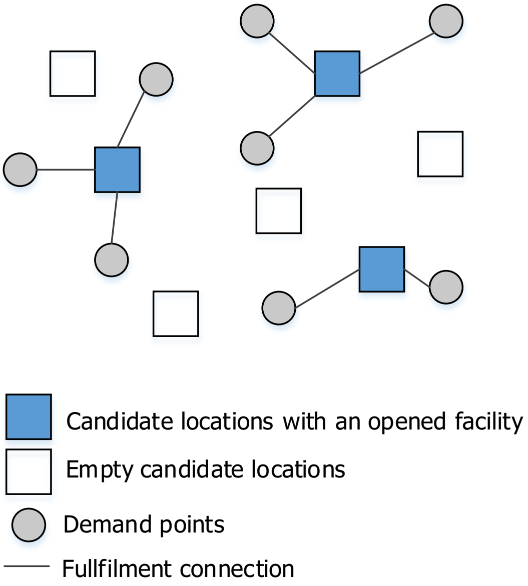

Fig. 1 An illustration for UFLP

In the related literature, there are numerous publications focusing on location problems. Reporting them all is out of the scope of the present paper. Therefore, only the closely related ones and some state-of-the-art are mentioned here. In one of them, Jaramillo et al. (2002) employed GAs on several location problems. Tabu Search is another well-known method for solving UFLP (Al-Sultan and Al-Fawzan 1999; Sun 2006). Wang et al. (2008) introduced a multi-population-based parallel PSO that divides whole population into sub-populations for UFLP. Literature also includes other PSO implementations for this problem (Sevkli and Guner 2006; Guner and Sevkli 2008). In a more recent work, an Artificial Bee Colony (ABC) optimization algorithm was proposed by Kiran (2015). Tsuya et al. (2017) employed FA to solve UFLP. de Armas et al. (2017) and Della Croce et al. (2017a, b) reported deterministic and stochastic versions of UFLP. Hale and Moberg (2003) and Şahin and Süral (2007) presented comprehensive surveys for the related problem.

2.2 0–1 Knapsack problem (0–1 KP)

0–1 KP (Lin 2008) is the second benchmark problem used in the present study. 0–1 KP has many practical real world applications such as cargo loading, project funding selection, budget management, cutting stock, etc., (Kellerer et al. 2004). The aim in 0–1 KP is to maximize the knapsack profit such that the capacity of the knapsack is not exceeded. 0–1 KP can be formulated as in Eqs. 6–8.

$$maximize \sum_{r \in R} p_r x_r \tag{6}$$

$$\text{s.t.} \quad \sum_{r \in R} w_r x_r \le C \tag{7}$$

$$x_r \in \{0, 1\}, \quad r \in R \tag{8}$$

where $R = \{1, 2, \ldots, z\}$ is the set of items, p_r represents the profit the rth item, w_r is the resource consumption of the rth item in the knapsack, C is the capacity of the knapsack, x_r is a binary variable denoting whether the rth item is assigned to the knapsack ($x_r = 1$ if the rth item is assigned, $x_r = 0$, otherwise). In this regard, Eq. 6 is the objective function to maximize the profit of the knapsack. Equation 7 represents the capacity constraint. Finally, Eq. 8 imposes restrictions on the decision variables.

0–1 KP is shown to be an NP-hard problem (Kellerer et al. 2004). Although there exist some exact solution methodologies for 0–1 KP (Della Croce et al. 2017a, b), heuristic approaches are more commonly preferred due to similar reasons with those of UFLP's. Shi (2006) proposed an improved modification of ACO for 0–1 KP. Shah-Hosseini (2008) employed an intelligent water drops algorithm to solve and extension of the 0–1 KP. Drake et al. (2014) used a genetic programming based on a hyper-heuristic for the same problem. A schema-guiding evolutionary algorithm was proposed by Liu and Liu (2009). A binary modification of PSO was employed by Li and Li (2009). Several bio-inspired algorithms were proposed to solve 0–1 KP (Bhattacharjee and Sarmah 2014; Feng et al. 2017). Zhou et al. (2016) used a complex-valued encoding in a metaheuristic algorithm. The same procedure was also adopted by Zhou et al. (2017) within another approximation algorithm, namely, Wind Driven Optimization algorithm. Bhattacharjee and Sarmah (2017) reported a survey focusing on swarm-based algorithms for knapsack problems.

2.3 Set union knapsack problem (SUKP)

SUKP is an extension of 0–1 KP (Goldschmidt et al. 1994; Kellerer et al. 2004; Arulselvan 2014). It comprises of a set of elements $U=\{1, \ldots, n\}$ and a set of items $\aleph = \{1, \ldots, m\}$, where each item in set $\aleph$ $(i=1, \ldots, m)$ corresponds to a subset of elements, represented by S_i, with a non-negative profit denoted by $p : \aleph \rightarrow \mathbb{R}^+$. Each of the elements has non-negative weight $w : U \rightarrow \mathbb{R}^+$ in SUKP. For an arbitrary subset $A \subseteq \aleph$, total weight of the union of subset A is defined as $W(A) = \sum_{e\in \cup_{i\in A} S_i} w_e$ and profit of A is denoted by $P(A) = \sum_{i\in A} p_i$. The aim is to find a subset of items $\aleph^* \subseteq \aleph$ such that the total profit $P(\aleph^*)$ is maximized and $W(\aleph^*)$ does not exceed knapsack capacity. The problem is formally given by Eqs. 9–10, where w_e is the weight of the eth element in the union set of the selected items, p_i is the profit of ith item in the subset A and C is the knapsack capacity.

$$maximize P(A) = \sum_{i\in A} p_i \tag{9}$$

s.t.

$$W(A) = \sum_{e\in \cup_{i\in A} S_i} w_e \leq C, A \subseteq \aleph \tag{10}$$

An illustration for the SUKP is depicted in Fig. 2. As one can see from this figure, there exist three items each with three elements. The item-1 contains the element-1 and the element-3, while the item-2 contains only the element-2. Finally, the item-3 contains the element-2 and the element-3. If the item-1 and the item-2 are included in the knapsack, the weights of all elements should be taken into account

Fig. 2 An illustration for set union knapsack problem

while evaluating the capacity constraint. Similarly, if the item-2 and the item-3 are assigned to the knapsack, only the element-2 and the element-3 should be considered, because, their union set is comprised of only these elements.

SKUP has many practical applications including financial decision-making, database partitioning, data stream compression etc. (He et al. 2018). Although there are some SKUP-related publications in the literature, this problem deserves further research. Goldschmidt et al. (1994) proposed a dynamic programming procedure for SUKP. Lister et al. (2010) addressed dynamic character caching as SKUP. An important note, which is related to boundaries and a special case of SUKP, was presented by Arulselvan (2014). Riondato and Vandin (2014) solved SUKP in order to obtain an upper bound for frequent itemsets, which is closely related to data mining issues. Several approximation schemes for SUKP and related problems were presented by Taylor (2016). Su and Zhou (2016) presented a reduced modification of this problem. Diallo et al. (2017) modeled virtual machine selection problem in cloud computing systems as a SUKP. Finally, He et al. (2018) presented detailed discussions about this problem.

3 Weighted superposition attraction algorithm

Before introducing the proposed bWSA, canonical WSA (Baykasoğlu and Akpinar 2015, 2017) is first summarized.

3.1 Canonical WSA

WSA is initialized with a predefined number of solutions. These solution vectors are referred to as artificial agents in WSA. Each artificial agent has its own position (coordinates) and an objective function value, representing the quality of the corresponding solution vector.

3.1.1 Initialization of WSA

This step consists of determining the values of the parameters used in WSA. The symbols of these parameters and related definitions are given below in Table 1. This nomenclature is used in the rest of the paper.

3.1.2 Generating neighborhood solutions in WSA

The artificial agents in WSA move according to the guidance of the superposition vector. Therefore, the first step in neighborhood solution generation is to generate a superposition by making use of the current population information. Accordingly, WSA algorithm calculates the coordinates of the superposition as the weighted sum of the agents' positions. In this respect, first, artificial agents are sorted according to their fitness values. The first rank is the fittest agent here. Next, *weights* of each agent are evaluated based on these ranks. Finally, weighted sum for each dimension is calculated as given

Table 1 Nomenclature for WSA

maxiter	Allowed maximum number of iterations (termination criterion)
t	Iteration index
AA	Number of artificial agents (population size)
D	Number of dimensions
τ	User-supplied parameter
λ	User-supplied parameter
φ	User-supplied parameter
UB	Upper bound of the dimensions
LB	Lower bound of the dimensions
$f(i)$	Fitness of agent i
$f(tar)$	Fitness of the target point (superposition)
weight	Weight of the current position of an agent
$\vec{x}$	Current position vector of an agent
$\overrightarrow{tar}$	Coordinates of the superposition (target point)
$\overrightarrow{gap}$	Vector combining an agent to target point
$\overrightarrow{direct}$	Search direction vector of an agent
sign()	Signum function
sl	Step length
rand()	Random uniform number $\in [0, 1]$

in Algorithm 1. In this procedure, effects of the sorted agents on the generated superposition are exponentially decreased as their ranks increase. While the fittest individual has the greatest effect, lower quality solutions have minor effects. It should be noted here that effects the artificial agents on the generated superposition can be controlled by the user-supplied parameter τ.

Subsequent to generating the target vector (superposition), its fitness is evaluated and search directions of agents are determined as presented in Algorithm 2. It should be mentioned that this procedure is given for a minimization problem. According to this procedure, agents of which the fitness values are worse than that of the superposition's, get attracted by the superposition vector. Additionally, if an agent is better than the superposition, then that agent might also move towards the coordinates of the supposition by a probability defined by $e^{f(i)-f\left(\overrightarrow{tar}\right)}$. Otherwise, that agent performs a random walk. Finally, the positions of the agents for the next iteration are determined in regard to the formulation given in Eq. 11, where $\left|\vec{x}(i,j)^t\right|$ represents the absolute value of the position vector of the ith artificial agent on dimension j at iteration t, and $\overrightarrow{direct}(i,j)^t \in \{-1, 0, 1\}$. The mentioned procedure is formally given in Algorithm 3, where the parameter sl is also adaptively tuned. Updating of the sl for the next iteration is formulated in Eq. 12.

Algorithm 1. Generating a target point.

```
sort the agents
$weight = zeros\ (1,AA)$
$\overrightarrow{tar} = zeros\ (1,D)$
for ($i$=1 to $AA$) do
        $weight\ (1,i) = i^{-\tau}$
        for ($j$=1 to $D$) do
                $\overrightarrow{tar}(1,j) = \vec{x}(i,j) \times weight\ (1,i)$
        end for
end for
```

Algorithm 2. Determining search directions.

```
$\overrightarrow{gap} = zeros\ (AA,D)$
$\overrightarrow{direct} = zeros\ (AA,D)$
for ($i$=1 to $AA$) do
        if $f(i) \geq f(\overrightarrow{tar})$
                for ($j$=1 to $D$) do
                        $\overrightarrow{gap}(i,j) = \overrightarrow{tar}(1,j) - \vec{x}(i,j)$
                end for
                for ($j$=1 to $D$) do
                        $\overrightarrow{direct}(i,j) = sign(\overrightarrow{gap}(i,j))$
                end for
        else if $f(i) < f(\overrightarrow{tar})$
                if rand() $< e^{f(i) - f(\overrightarrow{tar})}$
                        for ($j$=1 to $D$) do
                                $\overrightarrow{gap}(i,j) = \overrightarrow{tar}(1,j) - \vec{x}(i,j)$
                        end for
                        for ($j$=1 to $D$) do
                                $\overrightarrow{direct}(i,j) = sign(\overrightarrow{gap}(i,j))$
                        end for
                else
                        for ($j$=1 to $D$) do
                                $\overrightarrow{direct}(i,j) = sign(-1 + 2 \times \boldsymbol{rand}())$
                        end for
                end if
        end if
end for
```

Algorithm 3. Updating artificial agents and step length.

```
if rand() $\leq \lambda$
        $sl = sl - e^{-t/(t+1)} \times \varphi \times sl$
else
        $sl = sl + e^{-t/(t+1)} \times \varphi \times sl$
end if
for ($i$=1 to $AA$) do
        for ($j$=1 to $D$) do
                $\vec{x}(i,j) = \vec{x}(i,j) + sl \times \overrightarrow{direct}(i,j) \times \|\vec{x}(i,j)\|$
                if $\vec{x}(i,j) < LL$
                        $\vec{x}(i,j) = LL$
                else if $\vec{x}(i,j) > UL$
                        $\vec{x}(i,j) = UL$
                end if
        end for
end for
```

$$\vec{x}(i,j)^{t+1} = \vec{x}(i,j)^{t} + sl^{t} \times \overrightarrow{direct}(i,j)^{t} \times \left|\vec{x}(i,j)^{t}\right| \tag{11}$$

$$sl^{t+1} = \begin{cases} sl^{t} - e^{-t/(t+1)} \times \varphi \times sl^{t} & if \quad rand() \leq \lambda \\ sl^{t} + e^{-t/(t+1)} \times \varphi \times sl^{t} & if \quad rand() > \lambda \end{cases} \tag{12}$$

Putting things together, subsequent to initialization of WSA algorithm, an iteration here is comprised of first, determination of the superposition and next, movement of all artificial agents in guidance of this generated superposition vector. Finally, at the end of an iteration, the best-found solution is updated if necessary. It is worth stressing that, although an adaptive step length is proposed in the canonical version of WSA, it not necessarily required in implementation. One can simply use a fixed step length that remains stationary throughout the search or any other step length controlling method can optionally replace this method.

3.2 The binary WSA

As mentioned above, a superposition in canonical WSA is generated by the weighted sum of the solution vectors for each dimension. However, in a binary space, this procedure is not practical. Therefore, in the present work, a recent method (Baykasoğlu and Şenol 2016a; Baykasoglu and Ozsoydan 2018), which is clarified in the following subsection, is adopted.

3.2.1 Generating binary superposition

In bWSA, agents are first sorted in regard to their fitness values (1st rank is the fittest) and *weights* are evaluated as in canonical WSA (Algorithm 1). Next, uniform random numbers *rand*() ∊ [0,1] are generated for each dimension. These random numbers are used as threshold values to determine the candidate agents that undergo roulette wheel selection. Thus, agents, whose *weights* are equal to or greater than these thresholds undergo roulette wheel procedure. Finally, the bit of the winner agent is chosen as the corresponding bit of the superposition vector. This procedure is followed for each of the dimensions. For a better understanding, an example with different *weights* for different τ values is presented in Table 2.

Let's assume that τ is fixed to 0.7 and generated random numbers for each of the corresponding dimensions are 0.6, 0.3, 0.35, 0.4 and 0.9, respectively. Thus, for the first dimension, the first two agents undergo a roulette wheel, because, their weights are greater than or equal to 0.6. The bit of the winner agent (underlined and highlighted in bold) of this roulette wheel is transferred to the superposition vector (which is currently empty). The same procedure is performed for the remaining dimensions except for the last dimension. Since at least two candidates are required to perform a roulette wheel, the bit of the first agent for the last dimension is directly transferred to the superposition vector. Finally, the generated superposition is given in the last row of Table 2.

It should be stressed that the first agent is always a candidate for the roulette wheel procedure. It is also clear that transfer functions are not required here. Thus, one can directly use a binary population. Moreover, in parallel with WSA, behavior of bWSA can also be controlled via different values of τ. Lower values of τ encourages diversity, on the contrary, greater values of τ yield to a more elitist behavior.

3.2.2 Solution representation and generating neighbors in bWSA

A solution is represented by binary variables in bWSA. It is clear that the 1s in a solution string represent the opened facilities at the corresponding locations in UFLP. Therefore, the length of a solution string for UFLP is equal to the number of possible locations at which a facility can be established. Similarly, length of a solution string for 0–1 KP and SUKP is equal to the number of items that can be assigned to the knapsack. In parallel, 1s in a solution string for 0–1 KP and SUKP represent the included items in the knapsack.

Table 2 A sample population for bWSA

						weight $(1,i)=i^{-\tau}$				
ranks	*dimension*					$\tau = 0.1$	$\tau = 0.3$	$\tau = 0.5$	$\tau = 0.7$	$\tau = 0.9$
1	0	1	**1**	0	**1**	1.000	1.000	1.000	1.000	1.000
2	**1**	1	0	**0**	1	0.933	0.812	0.707	0.615	0.535
3	0	**1**	0	1	1	0.895	0.719	0.577	0.463	0.372
4	1	1	1	0	0	0.870	0.659	0.500	0.378	0.287
5	1	0	0	1	1	0.851	0.617	0.447	0.324	0.234
6	0	0	1	1	0	0.835	0.584	0.408	0.285	0.199
7	1	0	1	0	1	0.823	0.557	0.377	0.256	0.173
8	1	1	0	1	0	0.812	0.535	0.353	0.233	0.153
random numbers	0.6	0.3	0.35	0.4	0.9					
superposition	1	1	1	0	1					

Gray shades are used for determininig the candidates that will undergo roulette wheel procedure

In canonical WSA, agents move via $\overrightarrow{direct}$ and $\overrightarrow{gap}$ vectors. The parameters sl and φ control the speed of movements. However, since these vectors and parameters are not usable in binary spaces with their current forms, movements of agents are modified by making use of uniform-based crossover and bit-flipping procedures.

It should be recalled that artificial agents either move towards the coordinates of the superposition or they perform random walks. Let us first consider the random walks. In order to detect promising regions quickly, it is clear that a more diversified search is required at the beginning of the search. Additionally, it is known that an approximation algorithm should intensify to achieve better results. Therefore, the exponentially decreasing step length (Baykasoglu 2012; Baykasoğlu and Ozsoydan 2015) given in Eq. 12 is modified here for binary spaces. In this extension, the parameter $sl^t \in [0, 1]$ is decreased for the next iteration by using the formula given in Eq. 13. Next, it is multiplied by the length of the solution string (dimension) and the obtained value is rounded up to obtain the total number of bits to flip, which shown by int_sl^{t+1} (Eq. 14). Finally, randomly selected int_sl^{t+1} bits of an agent are flipped.

$$sl^{t+1} = sl^t - e^{-t/(t+1)} * \varphi * sl^t \tag{13}$$

$$int\_sl^{t+1} = \lceil sl^{t+1} \times D \rceil \tag{14}$$

Differently from canonical WSA, it is assumed here that initial step length is bounded by $sl^1 \in [0, 1]$. For example, if sl^1 is set to 0.50, an agent performing a random walk flips 50% of its bits at the first iteration. An illustration for $sl^1 = 1.00$ and for $D = 50$ with different φ values is presented in Fig. 3. As one can see from Fig. 3.a, greater values of φ give rise to steeper decrements in the continuous step length. Accordingly, randomly walking agents flip only a few bits throughout the second half of the search. On the other hand, $\varphi = 0.01$ yields to a late intensification. Therefore, a precise balance should be established to use this procedure efficiently.

While moving an agent towards the superposition, a uniform-based crossover is employed in bWSA. In this procedure, uniform-based crossover rate (p_{uxo}) is defined first. This parameter represents the probability of taking a gene from the superposition vector. Accordingly, random numbers $rand() \in [0, 1]$ are generated

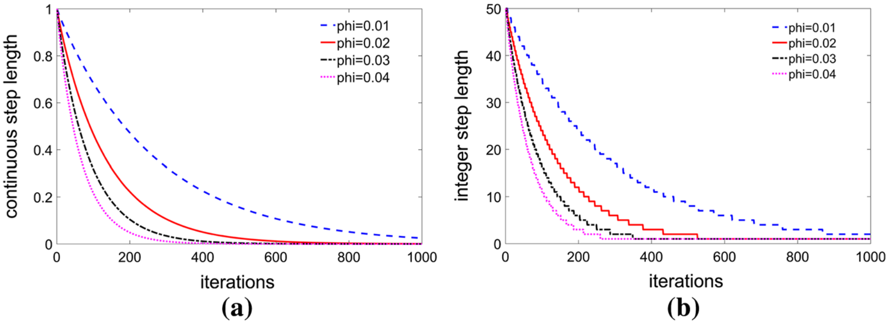

Fig. 3 Exponentially decreasing **a** continuous step length and **b** corresponding integer step length for different values of φ

for each dimension separately. If the generated random number for the corresponding bit is less than or equal to p_{uxo}, that dimension takes the value of the superposition's. Otherwise, it is not changed. Thus, greater values of p_{uxo} induce faster movements towards the superposition. In the present study, p_{uxo} is assumed to remain stationary until the end of the run. The artificial agent movements in bWSA with respect to a minimization problem are presented in Algorithm 4. Finally, putting things together, bWSA is provided in Algorithm 5.

Algorithm 4. Movement of a population in bWSA.

```
for (i=1 to AA) do
    if $f(\overrightarrow{tar}) \leq f(i)$
        for (j=1 to D) do
            if rand() $\leq p_{uxo}$
                $\vec{x}(i,j) = \overrightarrow{tar}(1,j)$
            end if
        end for
    else if $f(i) < f(\overrightarrow{tar})$
        if rand() $< e^{f(i)-f(\overrightarrow{tar})}$
            for (j=1 to D) do
                if rand() $\leq p_{uxo}$
                    $\vec{x}(i,j) = \overrightarrow{tar}(1,j)$
                end if
            end for
        else
            for (j=1 to $int\_sl^t$) do
                select a random locus $\omega$
                flip the value of $\vec{x}(i,\omega)$
            end for
        end if
    end if
end for
```

3.2.3 Handling infeasibilities

The first binary problem solved by bWSA is indeed obtained by excluding the capacity constraints from facility location problems. In other words, demands of all customers can be fulfilled by only a single opened facility in UFLP. It is also mentioned in the previous subsection that the number of the opened facilities is controlled by solution representations in bWSA. Therefore, this problem can be considered as an unconstrained binary optimization problem. Additionally, as mentioned by Kiran (2015), demand of a customer is always entirely fulfilled by the nearest opened facility.

Algorithm 5. A pseudo code for bWSA.

```
set the input parameters of WSA
generate an initial population & evaluate fitness values
for ($i$=1; $i \le maxiter$; $i$++) do
    rank the solutions according to their fitness values
    determine the weights and a superposition (Table 2)
    evaluate fitness of the binary superposition
    move artificial agents (Algorithm 4)
    evaluate fitness of the agents
    update the best found solution
    modify the $i$th step length (Eqs. 13-14)
end for
print the best solution
```

On the other hand, 0–1 KP has a knapsack capacity constraint. In this regard, the penalizing function of Zou et al. (2011) (Eq. 15) is adopted for 0–1 KP. In this method, first, the maximization function of 0–1 KP (Eq. 6) is transformed into a minimization function by multiplying it by -1. Next, the penalty term $\omega \times \max(0, g)$ is appended to the objective function, where $g = \sum_{r \in R} w_r x_r - C$ and ω is a scalar that is fixed to 10^{20} in Eq. 15.

$$minimize\left(-\sum_{r \in R} p_r x_r + \omega \times \max(0, g)\right) \tag{15}$$

Finally, in SUKP, the repairing method presented by He et al. (2018) is employed. In this method, first, a quick sorting algorithm is used to sort the items in regard to their profit/weight ratios, taking the frequencies of elements separately for each item into account. Next, violating item(s) is (are) found by using this sorted list. After discarding them, a posterior procedure is peformed to detect any fitting item(s) to the remaining capacity. This procedure is presented in detail by He et al. (2018).

4 Experimental study

4.1 Fine tuning of the parameters in bWSA

bWSA has several parameters to tune. Allowed maximum number of iterations (*maxiter*) and population size (*AA*), which determine the number of consumed function evaluations (FEs), are two of these parameters. In order to perform fair comparisons with the literature and to keep FEs at an acceptable level, a balance between the parameters *maxiter* and *AA* is established. In this respect, performed preliminary trial tests show that higher quality solutions can be obtained, when *maxiter* and *AA* are fixed to 1000 and 20, respectively.

Another parameter that is denoted by $\tau \in [0,1]$ defines the search characteristic of bWSA. Therefore, a set of different values for τ ({0.1, 0.2, 0.3, …, 0.9}) is tested in preliminary work to observe the effects of this parameter. The preliminary empirical tests that have been performed based on medium-scaled instances show that bWSA achieves better results, when τ is fixed to 0.80.

The parameters φ and $sl^t \in [0,1]$ have critical effects on random walks. It is further clear that the value of sl^1 affects the succeeding step lengths. Therefore, one needs to define the initial step length sl^1 and the speed of decrement (φ) to control the random walks in bWSA. According to the results of the preliminary tests, sl^1 and φ are fixed to 0.4 and 0.008, respectively. Thus, while the agents flip 40% of their bits in the first generation, they flip only a single bit towards the end of the run.

The final parameter to tune in bWSA is the crossover rate ($p_{uxo} \in [0,1]$) that is used in uniform-based crossover while moving an agent towards the coordinates of the superposition. A set of different levels for p_{uxo} ({0.1, 0.2, 0.3, …, 0.9}) is tested. According to the same preliminary tests, p_{uxo} is fixed to 0.8.

4.2 Computational results

4.2.1 Results for UFLP

The performance of the bWSA in UFLP is tested in benchmarking instances taken from OR-Lib (http://people.brunel.ac.uk/~mastjjb/jeb/orlib/files/), for which the optimal solution is known. The attributes of these problems are presented in Table 3. Obtained results for UFLP are presented in Table 4. Columns of this table (*best*, *worst*, *std. dev.*, *hit*) represent the best-found solution, the worst-found solution, standard deviance of the found solutions over 30 runs and the number of runs for which the optimum is found, respectively.

Table 3 Attributes of the used UFLP instances

problem ID	Size (facilities × customers)	*Optimum solution value*
cap71	16×50	932,615.750
cap72	16×50	977,799.400
cap73	16×50	1,010,641.450
cap74	16×50	1,034,976.975
cap101	25×50	796,648.437
cap102	25×50	854,704.200
cap103	25×50	893,782.112
cap104	25×50	928,941.750
cap131	50×50	793,439.562
cap132	50×50	851,495.325
cap133	50×50	893,076.712
cap134	50×50	928,941.750

Table 4 Computational results for UFLP

problem ID	CPSO (Sevkli and Guner 2006)				ABC_{bin} (Kiran 2015)				bWSA			
	*best**	*worst**	*std. dev**	*hit**	*best*	*worst*	*std. dev*	*hit*	*best*	*worst*	*std. dev*	*hit*
cap71	932,615.75	934,199.14	562.23	25	932,615.75	932,615.75	0.00	30	**932,615.750**	**932,615.750**	**0.000**	**30**
cap72	977,799.40	983,713.81	1324.30	25	977,799.40	977,799.40	0.00	30	**977,799.400**	**977,799.400**	**0.000**	**30**
cap73	1,010,641.45	1,012,643.69	702.13	22	1,010,641.45	1,010,641.45	0.00	30	**1,010,641.450**	**1,010,641.450**	**0.000**	**30**
cap74	1,034,976.98	1,045,342.23	2124.54	0	1,034,976.98	1,034,976.98	0.00	30	**1,034,976.975**	**1,034,976.975**	**0.000**	**30**
cap101	796,648.44	802,457.23	1480.72	0	796,648.44	796,648.44	0.00	30	796,648.437	797,508.725	380.434	22
cap102	854,704.20	857,380.85	1015.64	10	854,704.20	854,704.20	0.00	30	**854,704.200**	**854,704.200**	**0.000**	**30**
cap103	893,782.11	899,424.91	1695.79	0	893,782.11	894,008.14	85.67	25	893,782.112	895,027.188	470.951	10
cap104	928,941.75	944,394.83	3842.64	18	928,941.75	928,941.75	0.00	30	**928,941.750**	**928,941.750**	**0.000**	**30**
cap131	795,291.86	804,549.64	2429.54	0	793,439.56	794,910.64	1065.73	6	**793,439.562**	**798,449.038**	**1025.786**	**6**
cap132	851,495.33	865,667.16	4297.07	0	851,495.33	851,636.70	213.28	14	**851,495.325**	**852,257.975**	**251.654**	**23**
cap133	893,076.71	909,908.70	4210.93	0	893,076.71	895,407.93	561.34	5	**893,076.712**	**894,801.163**	**501.912**	**7**
cap134	928,941.75	951,803.25	6619.05	7	928,941.75	928,941.75	0.00	30	928,941.750	934,586.975	1016.144	26

*Values in these columns are adopted from Kiran (2015)

Results of bWSA are compared to the previously published results of the algorithms based on PSO and ABC. As one can see from Table 4, bWSA is capable of finding optimum solutions in all instances. Additionally, the highlighted results in Table 4 represent the instances, for which the *hit* numbers are greater than or equal to the *hit* numbers of other algorithms. In this respect, bWSA clearly outperforms CPSO of Sevkli and Guner (2006). Moreover, it can be concluded that the proposed algorithm is competitive considering the results of ABC, recently reported by Kiran (2015).

Finally, convergence graphs of bWSA for different instances are presented in Fig. 4. Since each problem has different optimum, plotting them on the same graph with an identical scale yields to disproportionality. Therefore, percentage gap values (Eq. 16) for the corresponding iteration are plotted in these figures. Each data point is the average over 30 runs.

$$gap\% = \left({}^{(z^* - opt)}/_{opt} \right) \times 100 \tag{16}$$

It is clear from Fig. 4 that bWSA starts with greater gaps in larger-scaled instances. This is an expected result due to the adopted step size function. However, it is further clear from Fig. 4 that bWSA is capable of finding approximately same gap % values after performing only 50–100 iterations regardless of the difficulty

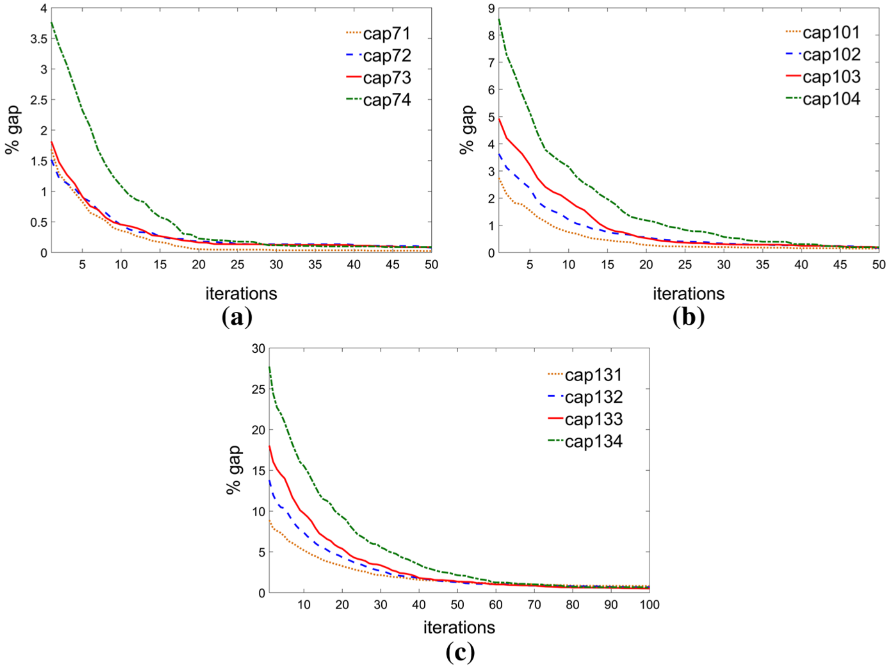

Fig. 4 Gap % convergence graph of bWSA for instances **a** cap71–74, **b** 101–104 and **c** cap131–134

of the problem. This demonstrates the convergence capability of the proposed approach.

4.2.2 Results for 0–1 KP

The performance of bWSA in 0–1 KP is tested first by using the standard benchmarks instances, which are denoted by $f_i(i = 1, \dots, 10)$ in Table 5. Next, the proposed approach is tested in randomly generated larger-scaled instances. All printed results are evaluated over 30 independent runs. The results of bWSA for the 10 commonly used benchmarks are presented in Table 5 of which the columns (*f*, *opt.*, *best*, *mean*, *worst*, *std. dev.*, gap %, *hit*) denote the instance index, optimum value, the best found solution, mean over 30 runs, the worst found solution, standard deviance, percentage gap (Eq. 16) and the number of runs, for which the optimum is found, respectively.

As one can see from Table 5, bWSA is capable of finding optimum solution for each of the instances $f_{1\text{-}10}$. Moreover, except for the instance f_7, bWSA finds optimum solutions in all runs. However, it is apparent that the proposed approach finds sub-optimal results in a few runs of instance f_7.

Secondarily, bWSA is tested in large-scaled 0–1 KP instances, which are generated according to the guidelines reported in Zou et al. (2011). Briefly, in this technique, profit and resource consumption parameters of the items are randomly generated as integer numbers according to some restrictions and the knapsack capacity is assumed to be fixed. It is clear that, although the optimum solutions of the instances $f_{1\text{-}10}$ are known, an upper bound technique for the rest of the instances $f_{11\text{-}18}$ is required. In this respect, *Dantzig upper bound* (Pisinger and Saidi 2017) is employed in the present study. In *Dantzig upper bound*, the mathematical model is first relaxed. Thus, the LP-relaxed (fractional) 0–1 KP, where $0 \le x_r \le 1$ for all $r = 1, 2, \dots, n$. can be solved to optimality. Accordingly, items are sorted in regard to non-increasing order of *profit/weight* $\left({}^{p_r}/_{w_r}\right)$ ratios. Next, sorted items are assigned to the knapsack until an item s does not fit into the

Table 5 Standard instances for 0–1 KP (Zou et al. 2011; Bhattacharjee and Sarmah 2014)

f	*opt.*	*best*	*mean*	*worst*	*std. dev.*	*gap %*	*hit*
f_1	295	295	295	295	0.000	0.000	30
f_2	1024	1024	1024	1024	0.000	0.000	30
f_3.	35	35	35	35	0.000	0.000	30
f_4	23	23	23	23	0.000	0.000	30
f_5.	481.0694	481.0694	481.0694	481.0694	0.000	0.000	30
f_6.	52	52	52	52	0.000	0.000	30
f_7.	107	107	106.53	105	0.846	0.000	23
f_8.	9767	9767	9767	9767	0.000	0.000	30
f_9.	130	130	1	130	0.000	0.000	30
f_{10}.	1025	1025	1025	1025	0.000	0.000	30

knapsack. Finally, the optimum value of the LP-relaxed 0–1 KP can be found via Eq. 17. It is further known that if all profits are integers (as in the present case), this value can be floored and the *Dantzig upper bound* can be obtained as $\lfloor z^{*}_{LP\_relaxed} \rfloor$.

$$z^{*}_{LP\_relaxed} = \sum_{r=1}^{s-1} p_r + \left(C - \sum_{r=1}^{s-1} w_r \right) \times \frac{p_r}{w_r} \tag{17}$$

The results for the rest of the problems f_{11-18} are presented in Table 6, of which the columns (*f*, *number of items*, *Dantzig upper bound*, *best*, *mean*, *worst*, *std. dev.*, *gap %*) represent instance index, problem size, upper bound, the best found solution by the algorithm, mean over 30 runs, the worst found solution, standard deviance and the percentage gap (Eq. 16), respectively.

It is clear in this case that $\lfloor z^{*}_{LP\_relaxed} \rfloor$ replaces the *opt* in Eq. 16. As one can see from Table 6, obtained percentage gap values vary between 0.32% and 7.25% except for the instance f_{14}. Particularly for extremely large-scaled instances with 1000–1500 items ($f_{16\text{-}18}$), average gap appears to be approximately 4.09%, which can be considered as a promising performance. On the other hand, bWSA obtains lower quality solutions in f_{14}. It is thought that this circumstance arises from the structure of the generated data, which has severe effects on the tightness of the instance. In such cases, occasionally the gap between the theoretical upper bound the best-found solution might dramatically increase.

Convergence graphs for *gap %* values of the used instances (except for f_{14}) are presented in Fig. 5, where all plotted data points are the averages over 30 run. It is clear from this figure that bWSA can still converge to 0.00% gap if the algorithm is allowed to continue to search. This is due to the exponentially decreasing step size procedure. However, more challenging conditions are employed here and the algorithm is terminated when a fixed number of iterations is achieved. This criterion is defined regardless of the problem size. It is apparent from the curves in Fig. 5a (more horizontal looking) and Fig. 5b (more vertical looking).

Table 6 Results for randomly generated larger-scaled instances for 0–1 KP

f	*number of items*	*Dantzig upper bound*	*best*	*mean*	*worst*	*std. dev.*	*gap %*
f_{11}	100	6976	6954	6919.267	6874	17.699	0.315
f_{12}	200	10,997	10,769	10,700.500	10,627	43.279	2.073
f_{13}	300	14,042	13,606	13,429.667	13,296	71.275	3.105
f_{14}	500	18,444	13,042	12,517.067	11,396	304.224	29.289
f_{15}	800	38,908	36,088	35,711.233	35,210	219.376	7.248
f_{16}	1000	66,641	63,395	62,899.600	62,172	273.846	4.871
f_{17}	1200	87,308	85,346	84,437.433	83,193	388.231	2.247
f_{18}	1500	10,2309	97,025	96,330.567	95,854	277.809	5.165

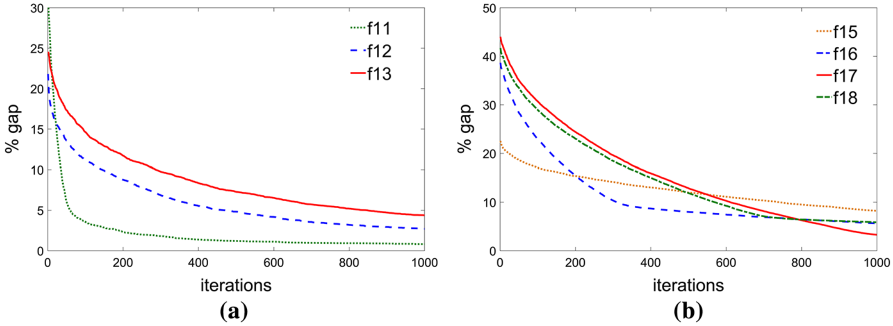

Fig. 5 Gap % convergence graph of bWSA for instances **a** f_{11-13} and **b** f_{14-18}

4.2.3 Results for SUKP

In order to test the performance of bWSA in SUKP, all instances presented by He et al. (2018) are used in the present work. The authors name those instances in the form of $m\_n\_\alpha\_\beta$, where m is the number of items, n is the number of elements, α is the density of an element and β is a parameter to define the knapsack capacity. Three groups of instances are provided by He et al. (2018), where it is assumed that $m > n$, $m = n$, $m < n$ for the first, for the second and for the third group of instances, respectively. The rest of the parameters are used exactly the same. All results, which are presented in Tables 7, 8, 9, are evaluated over 100 independent runs.

In Tables 7, 8, 9, the *Instance* columns represent the attributes of the instance. The columns *A-SUKP* give the results obtained by the method proposed by Arulselvan (2014). Another column *BABC* is devoted to the results of the binary *ABC* of He et al. (2018), which was inspired by the study of Ozturk et al. (2015). A similar algorithm is denoted by ABC_{bin} that is adopted from Kiran (2015). The other columns, *GA* and *binDE* are devoted to the results of GA and a binary modification of DE. All of these results are adopted from He et al. (2018). Finally, the columns *bWSA* are devoted to the results of the proposed approach.

As one can see from Tables 7, 8, 9, bWSA outperforms the rest of the algorithms in most of the instances. It is apparent that there are some ties particularly in smaller-scaled problems. However, it is further clear that the performance of bWSA is better than the compared algorithms particularly in large-scaled problems. This indeed shows that the proposed approach mostly obtains better solutions in SUKP regardless of the problem size. Additionally, it can be put forward that although similar *mean values* are obtained by each of the compared algorithms, generally speaking, *standard deviance* performance of bWSA is worse than the other algorithms'. This can be considered as a drawback for the proposed approach.

Putting things together, all conducted tests in the present work points out the efficiency of the proposed modification of bWSA in binary optimization problems.

Table 7 Computational results of the first type of instances for SUKP

Instance	*Results*	*A-SUKP*	*GA*	*BABC*	ABC_{bin}	*binDE*	*bWSA*
100_85_0.10_0.75	*best*	12,459	**13,044**	13,251	13,044	13,044	**13,044**
	mean	12,459	12,956.4	13,028.5	12,818.5	12,991	12,915.67
	std. dev.	0.00	130.66	92.63	153.06	75.95	185.45
	worst	–	–	–	–	–	12,244
100_85_0.15_0.85	*best*	11,119	12,066	12,238	12,238	**12,274**	12,238
	mean	11,119	11,546	12,155	12,049.3	12,123.9	11,527.41
	std. dev.	0.00	214.94	53.29	96.11	67.61	332.27
	worst	–	–	–	–	–	10,408
200_185_0.10_0.75	*best*	11,292	13,064	13,241	12,946	13,241	**13,250**
	mean	11,292	12,492.5	13,064.4	11,861.5	12,940.7	12,657.65
	std. dev.	0.00	320.03	99.57	324.65	205.70	319.58
	worst	–	–	–	–	–	11,951
200_185_0.15_0.85	*best*	12,262	13,671	13,829	13,671	13,671	**13,858**
	mean	12,262	12,802.9	13,359.2	12,537	13,110	12,585.35
	std. dev.	0.00	291.66	234.99	289.53	269.69	302.66
	worst	–	–	–	–	–	11,836
300_285_0.10_0.75	*best*	8941	10,553	10,428	9751	10,420	**10,991**
	mean	8941	9980.87	9994.76	9339.3	9899.24	10,366.21
	std. dev.	0.00	142.97	154.03	158.15	153.18	257.10
	worst	–	–	–	–	–	9802
300_285_0.15_0.85	*best*	9432	11,016	12012	10913	11,661	**12,093**
	mean	9432	10,349.8	10,902.9	9957.85	10,499.4	10,901.59
	std. dev.	0.00	215.13	449.45	276.90	403.95	508.79
	worst	–	–	–	–	–	9912
400_385_0.10_0.75	*best*	9076	10,083	10,766	9674	10,576	**11,321**
	mean	9076	9641.85	10,065.2	9187.76	9681.46	10,785.74
	std. dev.	0.00	168.94	241.45	167.08	275.05	361.45
	worst	–	–	–	–	–	9798
400_385_0.15_0.85	*best*	8514	9831	9649	8978	9649	**10,435**
	mean	8514	9326.77	9135.98	8539.95	9020.87	9587.72
	std. dev.	0.00	192.20	151.90	161.83	150.99	360.29
	worst	–	–	–	–	–	8695
500_485_0.10_0.75	*best*	9864	11031	10 784	10340	10,586	**11,540**
	mean	9864	10,567.9	10,452.2	9910.32	10,363.8	10,921.58
	std. dev.	0.00	123.15	114.35	120.82	93.39	351.69
	worst	–	–	–	–	–	10,293
500_485_0.15_0.85	*best*	8299	9472	9090	8759	9191	**9681**
	mean	8299	8692.67	8857.89	8365.04	8783.99	9013.09
	std. dev.	0.00	180.12	94.55	114.10	131.05	204.85
	worst	–	–	–	–	–	8479

Table 8 Computational results of the second type of instances for SUKP

Instance	*Results*	*A-SUKP*	*GA*	*BABC*	ABC_{bin}	*binDE*	*bWSA*
100_100_0.10_0.75	*best*	13,634	**14,044**	13,860	13,860	13,814	**14,044**
	mean	13634	13,806	13,734.9	13,547.2	13,675.9	13,492.71
	std. dev.	0.00	144.91	70.76	119.11	119.53	325.34
	worst	–	–	–	–	–	12,625
100_100_0.15_0.85	*best*	11,325	13,145	**13,508**	13,498	13,407	13,407
	mean	11,325	12,234.8	13,352.4	13,103.1	13,212.8	12,487.88
	std. dev.	0.00	388.66	155.14	343.46	287.45	718.23
	worst	–	–	–	–	–	10987
200_200_0.10_0.75	*best*	10,328	11,656	11,846	11191	11,535	**12,271**
	mean	10,328	10,888.7	11,194.3	10,424.1	10,969.4	11,430.23
	std. dev.	0.00	237.85	249.58	197.88	302.52	403.33
	worst	–	–	–	–	–	10,622
200_200_0.15_0.85	*best*	9784	11,792	11,521	11,287	11,469	**11,804**
	mean	9784	10,827.5	10,945	10,345.9	10,717.1	11,062.06
	std. dev.	0.00	334.43	255.14	273.47	341.08	423.90
	worst	–	–	–	–	–	10,042
300_300_0.10_0.75	*best*	10208	12,055	12,186	11,494	12,304	**12,644**
	mean	10208	11,755.1	11,945.8	10,922.3	11,864.4	12,227.56
	std. dev.	0.00	144.45	127.80	182.63	160.42	308.11
	worst	–	–	–	–	–	11,365
300_300_0.15_0.85	*best*	9183	10,666	10,382	9633	10,382	**11,113**
	mean	9183	10,099.2	9859.69	9186.87	9710.37	10,216.71
	std. dev.	0.00	337.42	177.02	147.78	208.48	351.12
	worst	–	–	–	–	–	9520
400_400_0.10_0.75	*best*	9751	10570	10,626	10,160	10,462	**11,199**
	mean	9751	10,112.4	10101.1	9549.04	9975.8	10,624.79
	std. dev.	0.00	157.89	196.99	141.27	185.57	266.46
	worst	–	–	–	–	–	9818
400_400_0.15_0.85	*best*	8497	9235	9541	9033	9388	**10,915**
	mean	8497	8793.76	9032.95	8365.62	8768.42	9580.64
	std. dev.	0.00	169.52	194.18	153.40	212.24	411.83
	worst	–	–	–	–	–	8717
500_500_0.10_0.75	*best*	9615	10,460	10,755	10,071	10,546	**10,827**
	mean	9615	10,185.4	10,328.5	9738.17	10,227.7	10,482.80
	std. dev.	0.00	114.19	91.615	111.63	103.32	165.62
	worst	–	–	–	–	–	10147
500_500_0.15_0.85	*best*	7883	9496	9318	9262	9312	**10,082**
	mean	7883	8882.88	9180.74	8617.91	9096.13	9478.71
	std. dev.	0.00	158.21	84.91	141.32	145.45	262.44
	worst	–	–	–	–	–	8705

Table 9 Computational results of the third type of instances for SUKP

Instance	*Results*	*A-SUKP*	*GA*	*BABC*	*ABC_{bin}*	*binDE*	*bWSA*
85_100_0.10_0.75	*best*	10,231	11,454	11,664	11,206	11,352	**11,947**
	mean	10,231	11,092.7	11,182.7	10,879.5	11,075	11,233.16
	std. dev.	0.00	171.22	183.57	163.62	119.42	216.67
	worst	–	–	–	–	–	10,627
85_100_0.15_0.85	*best*	10,483	12,124	**12,369**	12,006	**12,369**	**12,369**
	mean	10,483	11,326.3	12,081.6	11,485.3	11,875.9	11,342.70
	std. dev.	0.00	417.00	193.79	248.33	336.94	474.76
	worst	–	–	–	–	–	9774
185_200_0.10_0.75	*best*	11,508	12,841	13,047	12,308	13,024	**13,505**
	mean	11,508	12,236.6	12,522.8	11,667.9	12,277.5	12,689.09
	std. dev.	0.00	198.18	201.35	177.14	234.24	336.51
	worst	–	–	–	–	–	11,820
185_200_0.15_0.85	*best*	8621	**10,920**	10,602	10,376	10,547	10,831
	mean	8621	10,351.5	10,150.6	9684.33	10,085.4	10,228.07
	std. dev.	0.00	208.08	152.91	184.84	160.60	286.92
	worst	–	–	–	–	–	9467
285_300_0.10_0.75	*best*	9961	10,994	11,158	10,269	11,152	**11,538**
	mean	9961	10,640.1	10,775.9	9957.09	10,661.3	11,105.09
	std. dev.	0.00	126.84	116.80	141.48	149.84	197.78
	worst	–	–	–	–	–	10,600
285_300_0.15_0.85	*best*	9618	11,093	10,528	10,051	10,528	**11,377**
	mean	9618	10,190.3	9897.92	9424.15	9832.32	10,452.03
	std. dev.	0.00	249.76	186.53	197.14	232.72	416.76
	worst	–	–	–	–	–	9519
385_400_0.10_0.75	*best*	8672	9799	10,085	9235	9883	**10,414**
	mean	8672	9432.82	9537.5	8904.94	9314.57	9778.03
	std. dev.	0.00	163.84	184.62	111.85	191.59	221.49
	worst	–	–	–	–	–	9378
385_400_0.15_0.85	*best*	8064	9173	9456	8932	9352	**10,077**
	mean	8064	8703.66	9090.03	8407.06	8846.99	9203.52
	std. dev.	0.00	154.15	156.69	148.52	210.91	303.12
	worst	–	–	–	–	–	8600
485_500_0.10_0.75	*best*	9559	10,311	10,823	10,357	10,728	**10,835**
	mean	9559	9993.16	10,483.4	9615.37	10,159.4	10,607.21
	std. dev.	0.00	117.73	228.34	151.41	198.49	191.86
	worst	–	–	–	–	–	10,024
485_500_0.15_0.85	*best*	8157	9329	9333	8799	9218	**9603**
	mean	8157	8849.46	9085.57	8347.82	8919.64	9141.94
	std. dev.	0.00	141.84	115.62	122.65	168.90	180.42
	worst	–	–	–	–	–	8562

5 Concluding remarks

The present study examines the performance of a new generation metaheuristic referred to as WSA in binary optimization problems. In this respect, WSA, which is already shown to be efficient in a variety of optimization problems, is first modified for stationary binary spaces. A specialized roulette wheel procedure is used for this modification. Additionally, uniform-based crossover and random bit flips are employed while generating a new solution. Thus, any transfer functions for converting real values to binary are not required in the proposed bWSA. Moreover, a systematically controlled step sizing procedure to put control on the search speed is adopted in bWSA. Step size, which is indeed the number of bits to be flipped, is exponentially decreased throughout generations. Thus, while bWSA performs a more diversified search at the initialization stage, it is encouraged for intensification around the found promising regions towards the end of a run.

The performance of bWSA is examined in some well-known binary optimization problems, including uncapacitated facility location problem, 0–1 knapsack problem and set union knapsack problem, which have numerous applications in real-life. Promising results point out the efficiency of the proposed bWSA. Extending the proposed modification for other combinatorial problems is scheduled as future work.

References

Al-Sultan KS, Al-Fawzan MA (1999) A tabu search approach to the uncapacitated facility location problem. Ann Oper Res 86:91–103. https://doi.org/10.1023/A:1018956213524

Arulselvan A (2014) A note on the set union knapsack problem. Discrete Appl Math 169:214–218. https://doi.org/10.1016/j.dam.2013.12.015

Barcelo J, Hallefjord Å, Fernandez E, Jörnsten K (1990) Lagrangean relaxation and constraint generation procedures for capacitated plant location problems with single sourcing. OR Spectrum 12(2):79–88. https://doi.org/10.1007/BF01784983

Baykasoğlu A (2012) Design optimization with chaos embedded great deluge algorithm. Appl Soft Comput 12(3):1055–1067. https://doi.org/10.1016/j.asoc.2011.11.018

Baykasoğlu A, Akpinar Ş (2015) Weighted superposition attraction (WSA): a swarm intelligence algorithm for optimization problems—part 2: constrained optimization. Appl Soft Comput 37:396–415. https://doi.org/10.1016/j.asoc.2015.08.052

Baykasoğlu A, Akpinar Ş (2017) Weighted Superposition Attraction (WSA): a swarm intelligence algorithm for optimization problems—part 1: unconstrained optimization. Appl Soft Comput 56:520–540. https://doi.org/10.1016/j.asoc.2015.10.036

Baykasoğlu A, Ozsoydan FB (2015) Adaptive firefly algorithm with chaos for mechanical design optimization problems. Appl Soft Comput 36:152–164. https://doi.org/10.1016/j.asoc.2015.06.056

Baykasoğlu A, Ozsoydan FB (2018) Dynamic optimization in binary search spaces via weighted superposition attraction algorithm. Expert Syst Appl 96:157–174. https://doi.org/10.1016/j.eswa.2017.11.048

Baykasoğlu A, Şenol ME (2016a) Combinatorial optimization via weighted superposition attraction. In: International conference on operations research of the GOR, OR 2016, Hamburg

Baykasoğlu A, Şenol ME (2016b) Oppositon-based weighted superposition attraction algorithm for travelling salesman problems. In: Lm-Scm 2016 Xiv. International Logistics and Supply Chain Congress, Izmir, Turkey

Bhattacharjee KK, Sarmah SP (2014) Shuffled frog leaping algorithm and its application to 0/1 knapsack problem. Appl Soft Comput 19:252–263. https://doi.org/10.1016/j.asoc.2014.02.010

Bhattacharjee KK, Sarmah SP (2017) Modified swarm intelligence based techniques for the knapsack problem. Appl Intell 46(1):158–179. https://doi.org/10.1007/s10489-016-0822-y

Cornuejols ML, Nemhauser GL, Wolsey LA (1990) The uncapacitated facility location problem. In: Francis RL, Mirchandani P (eds) Discrete location theory. Wiley Interscience, New York

de Armas J, Juan AA, Marquès JM, Pedroso JP (2017) Solving the deterministic and stochastic uncapacitated facility location problem: from a heuristic to a simheuristic. J Oper Res Soc. https://doi.org/10.1057/s41274-016-0155-6

Della Croce F, Salassa F, Scatamacchia R (2017a) An exact approach for the 0–1 knapsack problem with setups. Comput Oper Res 80:61–67. https://doi.org/10.1016/j.cor.2016.11.015

Della Croce F, Salassa F, Scatamacchia R (2017b) A new exact approach for the 0–1 collapsing knapsack problem. Eur J Oper Res 260(1):56–69. https://doi.org/10.1016/j.ejor.2016.12.009

Diallo MH, August M, Hallman R, Kline M, Slayback SM, Graves C (2017) AutoMigrate: a framework for developing intelligent, self-managing cloud services with maximum availability. Clust Comput 20(3):1995–2012. https://doi.org/10.1007/s10586-017-0900-x

Dorigo M, Maniezzo V, Colorni A (1991) Positive feedback as a search strategy. Politecnico di Milano, Italy, (Technical Report No: 91-016)

Drake JH, Hyde M, Ibrahim K, Ozcan E (2014) A genetic programming hyper-heuristic for the multidimensional knapsack problem. Kybernetes 43(9/10):1500–1511. https://doi.org/10.1108/K-09-2013-0201

Erlenkotter D (1978) A dual-based procedure for uncapacitated facility location. Oper Res 26(6):992–1009. https://doi.org/10.1287/opre.26.6.992

Feng Y, Wang GG, Deb S, Lu M, Zhao XJ (2017) Solving 0–1 knapsack problem by a novel binary monarch butterfly optimization. Neural Comput Appl 28(7):1619–1634. https://doi.org/10.1007/s00521-015-2135-1

Goldschmidt O, Nehme D, Yu G (1994) Note: on the set-union knapsack problem. Naval Res Log 41(6):833–842. https://doi.org/10.1002/1520-6750(199410)41:6%3c833:AID-NAV3220410611%3e3.0.CO;2-Q

Guner AR, Sevkli M (2008) A discrete particle swarm optimization algorithm for uncapacitated facility location problem. J Artif Evol Appl. https://doi.org/10.1155/2008/861512

Hale TS, Moberg CR (2003) Location science research: a review. Ann Oper Res 123(1):21–35. https://doi.org/10.1023/A:1026110926707

He Y, Xie H, Wong TL, Wang X (2018) A novel binary artificial bee colony algorithm for the set-union knapsack problem. Future Gener Comput Syst 78:77–86. https://doi.org/10.1016/j.future.2017.05.044

Holland JH (1975) Adaptation in natural and artificial systems. The University of Michigan Press, Ann Arbor

Holmberg K (1999) Exact solution methods for uncapacitated location problems with convex transportation costs. Eur J Oper Res 114(1):127–140. https://doi.org/10.1016/S0377-2217(98)00039-3

Jaramillo JH, Bhadury J, Batta R (2002) On the use of genetic algorithms to solve location problems. Comput Oper Res 29(6):761–779. https://doi.org/10.1016/S0305-0548(01)00021-1

Kellerer H, Pferschy U, Pisinger D (2004) Knapsack problems. Springer, Berlin

Kennedy J, Eberhart R (1995) Particle swarm optimization. In: Proceedings of IEEE international conference on neural networks, 4, 1942–1948, Perth, WA, Australia. https://doi.org/10.1109/icnn.1995.488968

Kiran MS (2015) The continuous artificial bee colony algorithm for binary optimization. Appl Soft Comput 33:15–23. https://doi.org/10.1016/j.asoc.2015.04.007

Laporte G, Nickel S, da Gama FS (2015) Location science. Springer, Berlin

Li ZK, Li N (2009) A novel multi-mutation binary particle swarm optimization for 0/1 knapsack problem. In: Proceedings of IEEE control and decision conference, 3042–3047, Guilin, China. https://doi.org/10.1109/ccdc.2009.5192838

Lin FT (2008) Solving the knapsack problem with imprecise weight coefficients using genetic algorithms. Eur J Oper Res 185(1):133–145. https://doi.org/10.1016/j.ejor.2006.12.046

Lister W, Laycock RG, Day AM (2010) A dynamic cache for real-time crowd rendering. In: Computer graphics forum

Liu Y, Liu C (2009) A schema-guiding evolutionary algorithm for 0–1 knapsack problem. In: Proceedings of IEEE computer science and information technology—spring conference, 160–164, Singapore, Singapore. https://doi.org/10.1109/iacsit-sc.2009.31

Özbakır L, Turna F (2017) Clustering performance comparison of new generation meta-heuristic algorithms. Knowl-Based Syst 130(2017):1–16. https://doi.org/10.1016/j.knosys.2017.05.023

Ozturk C, Hancer E, Karaboga D (2015) A novel binary artificial bee colony algorithm based on genetic operators. Inf Sci 297:154–170. https://doi.org/10.1016/j.ins.2014.10.060

Pisinger D, Saidi A (2017) Tolerance analysis for 0–1 knapsack problems. Eur J Oper Res 258(3):866–876. https://doi.org/10.1016/j.ejor.2016.10.054

Rajabioun R (2011) Cuckoo optimization algorithm. Appl Soft Comput 11(8):5508–5518. https://doi.org/10.1016/j.asoc.2011.05.008

Riondato M, Vandin F (2014) Finding the true frequent itemsets. In: Proceedings of the 2014 SIAM international conference on data mining (pp 497–505). Society for Industrial and Applied Mathematics. https://doi.org/10.1137/1.9781611973440.57

Şahin G, Süral H (2007) A review of hierarchical facility location models. Comput Oper Res 34(8):2310–2331. https://doi.org/10.1016/j.cor.2005.09.005

Sevkli M, Guner A (2006) A continuous particle swarm optimization algorithm for uncapacitated facility location problem. In: Ant colony optimization and swarm intelligence, 316–323. https://doi.org/10.1007/11839088_28

Shah-Hosseini H (2008) Intelligent water drops algorithm: a new optimization method for solving the multiple knapsack problem. Int J Intell Comput Cybern 1(2):193–212. https://doi.org/10.1108/17563780810874717

Shi H (2006). Solution to 0/1 knapsack problem based on improved ant colony algorithm. In: Proceedings of IEEE international conference on information acquisition, 1062–1066, Weihai, China. https://doi.org/10.1109/icia.2006.305887

Storn R, Price K (1997) Differential evolution: a simple and efficient heuristic for global optimization over continuous spaces. J Glob Optim 11(4):341–359. https://doi.org/10.1023/A:1008202821328

Su L, Zhou Y (2016) Tolerating correlated failures in massively parallel stream processing engines. In: Proceedings of IEEE conference in data engineering (ICDE), 517–528, Helsinki, Finland. https://doi.org/10.1109/icde.2016.7498267

Sun M (2006) Solving the uncapacitated facility location problem using tabu search. Comput Oper Res 33(9):2563–2589. https://doi.org/10.1016/j.cor.2005.07.014

Taylor R (2016) Approximations of the densest k-subhypergraph and set union knapsack problems. arXiv:1610.04935

Tsuya K, Takaya M, Yamamura A (2017) Application of the firefly algorithm to the uncapacitated facility location problem. J Intell Fuzzy Syst 32(4):3201–3208. https://doi.org/10.3233/JIFS-169263

Wang D, Wu CH, Ip A, Wang D, Yan Y (2008) Parallel multi-population particle swarm optimization algorithm for the uncapacitated facility location problem using openMP. In: Proceedings of IEEE world congress on computational intelligence, 1214–1218, Hong Kong, China. https://doi.org/10.1109/cec.2008.4630951

Yang XS (2008) Nature-inspired metaheuristic algorithms. Luniver Press

Yang XS (2010) A new metaheuristic bat-inspired algorithm. In: González J, Pelta D, Cruz C, Terrazas G, Krasnogor N (eds) Nature inspired cooperative strategies for optimization. Springer, Berlin. pp 65–74. https://doi.org/10.1007/978-3-642-12538-6_6

Zhou Y, Li L, Ma M (2016) A complex-valued encoding bat algorithm for solving 0–1 knapsack problem. Neural Process Lett 44(2):407–430. https://doi.org/10.1007/s11063-015-9465-y

Zhou Y, Bao Z, Luo Q, Zhang S (2017) A complex-valued encoding wind driven optimization for the 0–1 knapsack problem. Appl Intell 46(3):684–702. https://doi.org/10.1007/s10489-016-0855-2

Zou D, Gao L, Li S, Wu J (2011) Solving 0–1 knapsack problem by a novel global harmony search algorithm. Appl Soft Comput 11(2):1556–1564. https://doi.org/10.1016/j.asoc.2010.07.019

Affiliations

Adil Baykasoğlu[1] · **Fehmi Burcin Ozsoydan**[1] · **M. Emre Senol**[1]

Fehmi Burcin Ozsoydan
burcin.ozsoydan@deu.edu.tr

M. Emre Senol
emre.senol@deu.edu.tr

[1] Department of Industrial Engineering, Faculty of Engineering, Dokuz Eylül University, Tinaztepe Campus, 35397 Buca, Izmir, Turkey