International Journal of Applied Metaheuristic Computing
Volume 15 • Issue 1 • January-December 2024

An Artificial Bee Colony Algorithm for the Multidimensional Knapsack Problem:
Using Design of Experiments for Parameter Tuning

Niusha Yaghini
Iran University of Science and Technology, Iran

Mir Yasin Seyed Valizadeh
Iran University of Science and Technology, Iran

ABSTRACT

The Multidimensional Knapsack Problem (MDKP) stands as a prominent challenge in combinatorial optimization, with diverse applications across various domains. The Artificial Bee Colony (ABC) algorithm is a swarm intelligence optimization algorithm inspired by the foraging behavior of bees. The aim of this paper is to develop an ABC with the goal of improving the solution quality in comparison to previous studies for the MDKP. In the proposed ABC algorithm, a heuristic method is presented to make employed bees. The roulette wheel and k-tournament methods are investigated for selecting employed bees by onlooker bees. For crossing over, two methods including one-point and uniform are studied. To tune the parameters, the Design of Experiment (DOE) method has been applied. The well-known benchmark test problems have been used to evaluate the proposed algorithm. The results show the absolute superiority of the solutions generated by the proposed algorithm in compared with the previous studies.

KEYWORDS

The Multidimensional Knapsack Problem, The Artificial Bee Colony Algorithm, The Design of Experiments, Metaheuristics

1. INTRODUCTION

The Multidimensional Knapsack Problem (MDKP) is an extension of the classic knapsack problem, where instead of a single constraint, there are multiple constraints to consider. Each item has multiple attributes or dimensions, and the goal is to maximize the total value of the items selected while staying within the constraints for each dimension. The MDKP is recognized as an NP-Hard integer programming problem (1). The problem can be mathematically defined as equations (1)-(3).

$$\max z = \sum_{j=1}^{n} c_j x_j, \tag{1}$$

$$\sum_{j=1}^{n} a_{ij} x_j \leq b_i, \quad i \in M = \{1,2,\ldots,m\}, \tag{2}$$

DOI: 10.4018/IJAMC.350225

This article published as an Open Access article distributed under the terms of the Creative Commons Attribution License (http://creativecommons.org/licenses/by/4.0/) which permits unrestricted use, distribution, and production in any medium, provided the author of the original work and original publication source are properly credited.

$$x_j \in \{0,1\}, \quad j \in N = \{1,2,\ldots,n\}$$

. (3)

In Kellerer et al. (2004) study, a set of items n with profits $c_j \geq 0$ needs to be packed into a knapsack with m dimensions, each having capacities $b_i \geq 0$. Each item j consumes $a_{ij} \geq 0$ from each dimension i and binary variables x_j determine the selection of items to maximize overall profit while adhering to knapsack constraints.

The MDKP is significant because it can represent a variety of real-world applications including resource allocation, intelligent transportation systems, logistics, Quality of Service (QoS), web service composition, energy-efficient offloading in mobile edge computing, medicine, budgeting problems, hardware design, and cloud computing. As a highly complex multi-constraint Combinatorial Optimization Problem (COP) with binary decision variables, and extensive research has been dedicated to the MDKP (Mkaouar et al., 2020).

Solving the MDKP is generally more complex than the classic knapsack problem due to the additional dimensionality. The most effective exact algorithms primarily utilize the branch-and-bound method. However, as the size of the MDKP increases, the time required for the branch-and-bound method grows exponentially, making it inefficient for large-scale MDKP instances. This inefficiency is a common drawback of exact algorithms. Various algorithms, such as dynamic programming, greedy algorithms, and metaheuristics can be adapted or extended to address the multidimensional version. Solving the MDKP is considered as a challenge in the field of optimization discussions (Chu & Beasley, 1998).

The ABC algorithm is a swarm intelligence optimization algorithm inspired by the foraging behavior of bees. ABC is widely used for solving optimization problems. ABC has been widely applied in various fields such as engineering design, data mining, and machine learning, due to its simplicity and ability to find high-quality solutions efficiently. Its versatility and effectiveness make it a popular choice for addressing diverse optimization challenges (Karaboga & Basturk, 2007).

The ABC algorithm is widely used due to its simplicity and high efficiency in various fields such as transportation, communications, engineering design, data mining, and machine learning. In transportation, ABC optimizes transportation routes and reduces associated costs (Karaboga et al., 2007). In communications, it optimizes resource allocation and manages wireless networks (Karaboga & Ozturk, 2011). In engineering design, ABC solves complex design and optimization problems (Akay & Karaboga, 2012). In data mining and machine learning, it is used for feature selection, clustering, and parameter optimization of machine learning models (Karaboga & Basturk, 2008). In distributed computing, ABC optimizes resource allocation and load management (Gao et al., 2012).

The aim of this article is to develop a metaheuristic algorithm with the goal of improving the solution quality in comparison to the published sources in literature. In this study, the Beasley (2017) dataset has been employed for the implementation of the proposed algorithm. A total of 30 problems, each comprising 100 items and 5 knapsacks, has been selected for evaluating the performance of the proposed algorithm. The contributions presented in this article can be summarized as follows:

(1) designing an Artificial Bee Colony (ABC) algorithm for MDKP,
(2) developing a heuristic method to make employed bees (solutions),
(3) investigating the roulette wheel and *k*-tournament methods for selecting employed bees by onlooker bees,
(4) evaluation of two uniform and one-point crossover methods,
(5) applying the Design of Experiments (DOE) method for parameter tuning,
(6) improving the quality of solutions in comparison with the previous studies.

In the following, the literature review on the topic is addressed in the second section, focusing on implemented heuristic and metaheuristic algorithms for the MDKP. The proposed ABC algorithm in the third section is explained. The fourth section presents parameter tuning using the Design of Experiments. In the fifth section, we delve into presenting the results and the final section of the paper is dedicated to the conclusion.

2. LITERATURE REVIEW

Numerous studies have been conducted on developing solution algorithms for the Multidimensional Knapsack Problem (MDKP). Given the objective of the article, this section reviews solution methods based on metaheuristics. This literature review categorizes the existing research based on different types of metaheuristic algorithms: evolutionary metaheuristics, swarm intelligence, Tabu Search, hybrid methods, heuristic methods, and instance generation and parameter control.

Significant advancements in solving MDKP using evolutionary algorithms include Chu et al.'s (1998) genetic algorithm-based approach and Yaghini et al.'s (2012) Memetic Algorithm using DIMMA principles. Liu et al. (2016) proposed a Binary Differential Evolution to effectively address the MDKP, contributing to this method's applicability in complex optimization scenarios. Mingo et al. (2017) combined Binary Particle Swarm Optimization with Genetic Algorithms, while Gazioglu (2022) developed the Bayesian Multiploid Genetic Algorithm. Most recently, Li et al. (2024) introduced a Binary Quantum-Behaved Particle Swarm Optimization (BQPSO) algorithm, which excels in large-scale MDKP instances.

Swarm intelligence algorithms have significantly advanced MDKP solutions. Notable contributions include Karaboga et al.'s (2007, 2009) Artificial Bee Colony (ABC) algorithm, Ke et al. (2010) demonstrated the effectiveness of Ant Colony Optimization on benchmark problems, showcasing its potential in solving MDKP efficiently. Abdel-basset et al.'s (2017) Improved Whales Optimization Algorithm, Meng et al.'s (2017) Fruit Fly Optimization Algorithm, and Garcia et al.'s (2017) integration of K-Means clustering with Firefly and Black Hole search algorithms. He et al. (2019) implemented the Grey Wolves Optimization Algorithm, while Lai et al. (2020) introduced the Diverse Particle Quantum Optimization Algorithm. Feng et al. (2022) developed the Binary Moth Search Algorithm, and Gupta et al. (2022) enhanced the Sine-Cosine Algorithm for MDKP. Recent contributions include Olivares et al. (2023) with Q-Learning for PSO and Du et al. (2023) with a binary Multi-Swarm version of the Fruit Fly Optimization Algorithm (bMFOA).

Dammeyer et al. (1993) investigated combining Tabu Search with Simulated Annealing to enhance optimization outcomes. Glover et al. (1996) and Hanafi et al. (1998) demonstrated the effectiveness of Tabu Search for MDKP. Lai et al. (2018) introduced the Two-Phase Evolutionary Tabu Algorithm, a more sophisticated approach tailored specifically for MDKP.

Hybrid methods combine two or more optimization techniques to leverage their individual strengths. Zhang et al. (2015) developed Hybrid Harmony Search by combining Fruit Fly Optimization and Harmony Search. Similarly, Feng et al. (2023) combined Hybrid Learning MS (HLMS) for solving the 0–1 MDKP with Global-Best Harmony Search (GHS) learning and Baldwinian learning, demonstrating significant improvements in optimization results.

Heuristic methods provide practical, though not necessarily optimal, solutions to complex problems. Weingartner et al. (1967) developed a heuristic based on Dynamic Programming, while Petersen (1967) introduced the Balas Additive Algorithm. Angelelli et al. (2010) proposed the Kernel Search Heuristic, and Hill et al. (2012) presented a heuristic that reduces problem size using Lagrangian relaxation. Wilbaut et al. (2009) proposed a dynamic variable fixation method for MDKP, and Della et al. (2012) evaluated a heuristic method for parallel computing.

Instance generation and parameter control methods focus on creating diverse problem instances and adaptively controlling algorithm parameters. Scherer et al. (2024) developed a novel instance generator for the multi-demand multidimensional knapsack problem, producing diverse and feasible

Table 1. Input data

Row	Data Name	Type	Description
1	*Knapsack_No*	Int	Number of knapsacks
2	*Capacity*	Int[]	Capacity array of knapsacks
3	*Profits*	Int[]	Profits array of each items
4	*Weights*	Int[][]	Weights matrix of the items

instances. Vega et al. (2024) proposed a self-adaptive strategy based on online parameter balance, improving the performance of population-based metaheuristics.

The above studies collectively showcase diverse approaches towards tackling MDKP, ranging from heuristic methods to sophisticated metaheuristic algorithms, contributing significantly to the optimization literature. The purpose of this paper is to continue the path of previous studies in improving the performance of algorithms and producing solutions with higher quality.

3. THE PROPOSED ARTIFICIAL BEE COLONY ALGORITHM

In this section, we will provide a comprehensive explanation of the proposed ABC algorithm. The ABC algorithm is a swarm intelligence optimization algorithm inspired by the foraging behavior of bees. Introduced by Karaboğa (2005), ABC is widely used for solving optimization problems. It starts with a population of artificial bees representing potential solutions to an optimization problem. The employed bees explore the solution space by adjusting their positions based on local information. The onlooker bees select solutions probabilistically according to their fitness. Abandoned solutions, which fail to improve over iterations, are replaced by new randomly generated bees. The scout bee is responsible for handling solutions that are no longer promising or have not been improved over a certain number of iterations. The purpose of the scout bee is to introduce diversity into the population by replacing these unproductive solutions with new randomly generated solutions.

The algorithm employs a balance between exploration and exploitation, seeking optimal solutions. The parameters like colony size and abandonment limit impact performance need careful adjustment. Its iterative process continues until a termination criterion, like a maximum number of iterations, is satisfied. In the subsequent part of this section, the presented algorithm is fully outlined.

3.1. The Data Structure

Data structures and algorithms are intricately connected, each relying on the other for effective computational solutions. Therefore, it is essential to carefully select an appropriate data structure before designing the algorithm. The algorithm begins by input data of the test problem. The problem input data is illustrated in a Table 1. Subsequently, the input parameters are presented in Table 2.

Other variables used include the number of employed bees and onlooker bees (each of which is half of the total number of colony bees), the time spent by the algorithm, the best bee found in each iteration and its fitness value, the best bee found in the whole algorithm and its fitness are presented in Table 3.

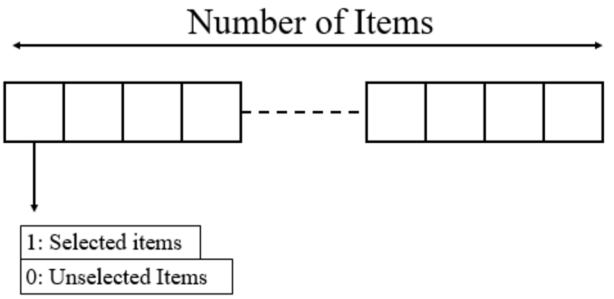
3.2. Overview of the Proposed ABC Algorithm

Figure 2 outlines the structure of the proposed ABC algorithm. It involves initializing a population of employed bees, iteratively performing employed and onlooker bee phases, tracking the best solution and fitness, and incorporating a scout bee phase to replace new solutions. The algorithm continues until the specified time limit is reached, and it returns the best fitness and corresponding solution

Table 2. Input parameters

Row	Parameter Name	Type	Description
1	*Time_Limit*	Int	Limitation of algorithm run time
2	*Bees_No*	Int	Number of bees
3	*Max_Try_Improve*	Int	Maximum iterative for improve
4	*Selection_Type*	String	Roulette Wheel / Tournament
5	*Crossover_Type*	String	One Point / Uniform
6	*Pc_Uniform*	Percent	Probability of crossover
7	*Pm*	Percent	Probability of mutation

Figure 1. Solution representation



found during the process. Let's delve into a step-by-step analysis of the pseudocode for the proposed ABC Algorithm.

The algorithm starts by receiving input data from Table 1 and parameters from Table 2, and it declares necessary variables from Table 3. Line 4, Captures the starting time for tracking elapsed time. In the following, a loop generates the initial population of employed bees using the **make_a_bee()** function. The algorithm enters the main loop, which continues until the elapsed time exceeds the

Table 3. Variables

Row	Var Name	Type	Initial Value	Description
	Bee	Class	NA	A class for generating bees
	Bee.selected_items	Boolean[]	[0]	Selected items in the bee
	Bee.fitness	Float	0	The fitness of the bee
	Bee.improve_try_num	Int	0	The improve try number of the bee
	employed_bees_no	Int	Bees_No/2	Employed bees number
	onlooker_bees_no	Object	Bees_No/2	Onlooker bees number
	elapsed_time	Int	0	Time elapsed of the algorithm
	best_fitness_of_iteration	Float	0	Best fitness of iteration
	best_bee_of_iteration	Object	Bee(Items_No)	Best bee of iteration
	best_fitness_so_far	Float	0	Best fitness so far
	best_bee_so_far	List	Null	Best bee so far

Figure 2. The pseudocode for the proposed ABC algorithm

```
Pseudocode for ABC_Algorithm()
 1:   Input data
 2:   Input parameters
 3:   Declare variables
 4:   start_time ← time()
 5:   For range (employed_bees_no):
 6:       bees.append(make_a_bee())
 7:   EndFor
 8:   While (elapsed_time < Time_Limit):
 9:     Call employed_bees_phase()
10:     Call onlooker_bees_phase()
11:     Find best_bee_of_iteration & best_fitness_of_iteration
12:     If (best_fitness_of_iteration > best_fitness_so_far):
13:         best_fitness_so_far ← best_fitness_of_iteration
14:         best_bee_so_far ← best_bee_of_iteration
15:     EndIf
16:     Call scout_bees_phase()
17:     elapsed_time ← start_time - time()
18:   EndWhile
19:   Return best_fitness_so_far, best_bee_so_far
```

specified time limit. Line 9 calls the employed bees phase, where employed bees explore and adjust their solutions. Line 10 executes the onlooker bees phase. For tracking the best solution, the algorithm identifies the best solution and its corresponding fitness in the current iteration. If the fitness of the best solution in the current iteration is greater than the best fitness observed so far, the best fitness and the corresponding bee are updated. Line 16 calls the scout bees phase, where scout bees replace new solutions. After this step, elapsed time has been updated. The main loop continues until the elapsed time surpasses the specified time limit. At the end of the proposed ABC algorithm in Line 19 returns the best fitness and the corresponding solution found during the algorithm's execution. Next, we proceed to explain the main functions of the algorithm.

3.3. Making a Bee

For making a bee a heuristic algorithm is presented. Making a bee is performed in the starting time of the algorithm and in scout bees phase. The function used for making a bee is illustrated in Figure 3. In the presented heuristic algorithm, *new_bee* is initialized as a new instance of the Bee class, and *feasibility_flag* is set to True initially. The algorithm enters a loop that continues until the generating solution is feasible. Inside the loop, a random item is selected. The algorithm examines if the selected item is not already chosen (Line 5). If the item is not selected, it sets it as selected (Line 6). Then it checks the feasibility of the current solution. If the solution is not feasible, it undoes the selection by setting the chosen item back to 0. Once the algorithm exits from the loop, the fitness of the solution is calculated (Line 13).

Finally, the function returns the *new_bee* object containing the feasible solution and its calculated fitness. The result is a bee with a set of selected items that satisfies the problem constraints, and its fitness is evaluated based on the chosen items.

Figure 3. The pseudocode for the make bee

```
Pseudocode for make_a_bee():
1:    new_bee ← Bee(Items_No)
2:    feasibility_flag ← True
3:    While (feasibility_flag):
4:        rand_num ← randint(0, Items_No)
5:        If (new_bee.selected_items[rand_num]=0):
6:            new_bee.selected_items[rand_num] ← 1
7:            feasibility_flag ← check_feasiblity(new_bee)
8:            If (not feasibility_flag):
9:                new_bee.selected_items[rand_num] ← 0
10:           EndIf
11:       EndIf
12:   EndWhile
13:   Calculate fitness of new_bee
14:   Return new_bee
```

3.4. The Employed Bees Phase

Figure 4 is representing the pseudocode of the employed bees phase. The employed bees phase is a crucial component of the ABC algorithm, designed to explore and improve solutions within the search space.

The employed bees phase starts with a population of artificial bees, where each bee represents a potential solution to the optimization problem. In the proposed ABC algorithm, each employed bee systematically selects another solution (another employed bee) for collaboration. Then, by using crossover and mutation operators, a new solution (*new_bee*) is generated.

The fitness of the new solution (*new_bee.fitness*) is then evaluated using the objective function. For solution update, if the fitness of the new solution is better than the fitness of the current solution (*current_bee.fitness*), the current bee is replaced with the new bee. If not, the current bee remains unchanged.

The employed bees phase attempts to enhance the quality of a bee's solution through a combination of crossover and mutation operations. It initializes a flag, *improve_flag*, to track whether an improvement is made during the proces. A deep copy of the current bee (*new_bee*) is created to avoid direct modifications. This phase dynamically selects the type of crossover operation based on the specified *crossover_type* (*"one_point" or "uniform"*). Following crossover, a mutation is applied to introduce random changes. Subsequently, it checks the feasibility of the new solution, calculates its fitness, and assesses whether an improvement is achieved. If an improvement occurs, the current bee's data and fitness are updated, and a counter (*try_improve*) is reset. In the absence of improvement, the counter is incremented, allowing the algorithm to adapt its strategy over successive iterations.

The employed bees, having potentially improved their solutions, share information with the onlooker bees. This information influences the onlookers in the subsequent phase. The algorithm iteratively refines the solutions through the employed bees' efforts, contributing to the overall search for optimal or near-optimal solutions.

3.5. The Onlooker Bees Phase

The onlooker bees phase is illustrated in Figure 5. The process involves random perturbing of the selected solution to generate a new candidate solution. The onlooker bees phase in the ABC algorithm involves the selection of employed bees by onlooker bees based on their fitness values.

Figure 4. The pseudocode for the employed bees phase

```
Pseudocode for employed_bees_phase()
For current_bee in bees:
    improve_flag = False
    new_bee ← current_bee
    If (crossover_type = "one_point"):
        Call crossover_one_point(new_bee)
    Elif (crossover_type = "uniform"):
        Call crossover_uniform(new_bee)
    EndIf
    Call mutation(new_bee)
    If (check_feasiblity(new_bee)):
        Call calculate_fitness(new_bee)
        If (new_bee.fitness > bee.fitness):
            improve_flag = True
            current_bee.selected_items ← new_bee.selected_items
            current_bee.fitness ← new_bee.fitness
            current_bee.try_improve ← 0
        EndIf
    EndIf
    If (not improve_flag):
        current_bee.try_improve += 1
    EndIf
EndFor
```

For the selection of the employed bees by the onlooker bees, two methods including roulette wheel selection and *k*-tournament selection, have been implemented. In the roulette wheel selection method, the probability of selecting each employed bees are calculated based on the equation (4). The onlooker bees select employed bees based on these probabilities. Solutions with higher fitness have a higher chance of being selected.

$$P_i = \frac{f(x_i)}{\sum_{i=1}^{N} f(x_i)} \tag{4}$$

Where $f(x_i)$ is the fitness level of i^{th} employed bees, and N is the total number of employed bees. Additionally, in the proposed ABC algorithm, we have utilized the tournament selection method to select employed bees. In the *k*-tournament selection method, *k* bees are randomly chosen, and the best bee based on the fitness is selected. Here, *k* is a parameter determining the size of the tournament.

Roulette Wheel assigns probabilities to each employed bee based on their fitness values. Bees with higher fitness values are given a higher probability of being selected by onlooker bees. This approach ensures that bees with better solutions have a higher chance of being chosen, mimicking the natural selection process where fitter individuals are more likely to be chosen for reproduction. *K*-Tournament involves randomly selecting k employed bees and then choosing the best one among them based on their fitness values. By setting k to a small value, typically 2 or 3, this approach introduces randomness into the selection process while still favoring bees with higher fitness values. *K*-tournament selection allows for a balance between exploration and exploitation, as it ensures that not only the best solution is selected but also provides opportunities for less fit solutions to be chosen, thus allowing for exploration of different areas of the search space.

Figure 5. The pseudocode for the onlooker bees phase

```
Pseudocode for onlooker_bees_phase()
For range (onlooker_bees_no):
    If (Selection_Type = "Roulette_Wheel"):
        current_bee ← roulette_wheel()
    Elif (Selection_Type = "Tournament"):
        current_bee ← tournament()
    EndIf
     improve_flag ←False
     new_bee ← current_bee
     If (crossover_type ="one_point"):
          Call crossover_one_point(new_bee)
     Elif (crossover_type ="uniform"):
          Call crossover_uniform(new_bee)
    EndIf
    Call mutation(new_bee)
    If (check_feasiblity(new_bee)):
          Call calculate_fitness(new_bee)
          If (new_bee.fitness > bee.fitness):
              improve_flag ←True
               current_bee.selected_items ← new_bee.selected_items
               current_bee.fitness ←  new_bee.fitness
               current_bee.try_improve ← 0
          EndIf
    EndIf
    If (not improve_flag):
         current_bee.try_improve ← current_bee.try_improve +1
    EndIf
EndFor
```

In this phase, a loop is constructed based on the number of onlooker bees. The *improve_flag* is initialized as False. This flag tracks whether an improvement is made during the process. The *new_bee* is created as a copy of the selected bee (*current_bee*). This avoids directly modifying the current solution. Depending on the specified *crossover_type* (“one_point” or “uniform”), the code applies the corresponding crossover operation to *new_bee.* The type of crossover determines how solutions are combined. The *new_bee* undergoes a mutation operation, introducing random changes to explore the solution space further. Then, the function checks the feasibility of the new solution, if the new solution is feasible, the its fitness is calculated. If the fitness of *new_bee* is better than the fitness of the *current_bee*, an improvement is considered. The *current_bee* is replaced with *new_bee*. If no improvement is made, the *try_improve* counter of the employed bee is incremented.

3.6. Scout Bees Phase

The scout bee phase is showed in Figure 6. The scout bee phase in the ABC algorithm is responsible for handling solutions that are no longer promising or have not been improved over a certain number of iterations. The purpose of the scout bee phase is to introduce diversity into the population by replacing these unproductive solutions with new randomly generated solutions.

In this phase, the variable *first_max_flag* is set to False initially. This flag is used to exit the loop after the first replacement. The variable *index* is initialized to 0, representing the index of the current bee being checked. Then, the algorithm enters a while loop that continues until either all bees are checked or the *first_max_flag* becomes True. Inside the loop, the current bee is selected from the list of employed bees. The function checks if the *try_improve* counter of the current bee exceeds the specified threshold (*max_try_improve*). If the threshold is reached, the current bee is removed from

Figure 6. Pseudocode for scout bees phase

```
Pseudocode for scout_bees_phase():
1:      first_max_flag = False
2:      index = 0
3:      While ((index < len(bees)) and (not first_max_flag)):
4:          current_bee = bees[index]
5:          If current_bee.try_improve >= max_try_improve:
6:              bees.delete(index)
7:              bees.append(make_a_bee())
8:              first_max_flag ← True
9:          EndIf
10:         index ← index + 1
11:     EndWhile
```

and a new bee is created using the *make_a_bee*() function and added to the bees. The *first_max_flag* is set to True to ensure that only the first bee exceeding the threshold triggers replacement. The *index* is incremented to move on to the next bee in the list. The main loop terminates with two conditions: either no bee has shown improvement for a maximum of *max_try_improve* iterations, or when the first bee with *max_try_improve* is found.

4. THE PARAMETER TUNING USING DESIGN OF EXPERIMENTS

The parameter tuning for metaheuristics involves finding the optimal values for the parameters to improve their performance. The Design of Experiments (DOE) is a systematic approach that can be used to efficiently explore the parameter space and identify the most influential factors affecting the algorithm's performance. In this study, the parameter tuning has been performed according to the steps illustrated in Figure 7.

In the rest of this section, step-by-step procedure are given on how to perform parameter tuning for the proposed ABC algorithm using DOE.

4.1. Choosing Test Problems

Selecting the appropriate test problems for tuning metaheuristic algorithms is a critical step in the parameter tuning process. The chosen test problems should be representative of the characteristics and challenges that the algorithm may encounter.

In this study, the test problem that selected for algorithm evaluation includes three groups with 30 problems, the first ten problems have a tightness ratio of 0.25, the second ten problems have a tightness ratio of 0.50 and the last ten problems have a tightness ratio of 0.75. Given this subject, one problem from each group has been selected for parameter tuning.

4.2. Parameters Selection and Their Ranges

In the proposed algorithm, we have a total of 7 parameters. Parameters are shown in Table 2 in the data structure section. In this study, based on the results obtained from the initial experimental runs, it was decided to set the maximum solution time to 60 seconds and choose the roulette wheel selection method and the uniform crossover type. Four remaining parameters were entered into the process of parameter tuning with the DOE. After obtaining the results from the experimental runs, the upper and lower bounds for the parameters were determined and presented in Table 4.

Table 4. Range of parameters

Row	Parameter	Unit	Low	High
1	*Bee_Num*	Number	150	250
2	*max_improvement_try*	Number	150	250
3	*Pc_Uniform*	Percent	0.2	0.6
4	*Pm*	Percent	0.005	0.015

4.3. Response Selection

In the context of the methodology for tuning metaheuristic algorithms, the step "Select the Responses" refers to the process of identifying and choosing the performance metrics or responses that will be used to evaluate the effectiveness and efficiency of the algorithm. The responses serve as the objective criteria to measure how well the algorithm performs on the chosen test problems for tuning. The gap between the fitness value of the obtained solution by running the proposed algorithm with different values of parameters and the best-known solution represented as the response. The gap is calculated based on Equation)5(.

$$Relative\ GAP = \frac{Obtained\ Solution - Best\ Known\ Solution}{Best\ Known\ Solution} \times 100 \quad (5)$$

Figure 7. Procedure of parameter tuning using design of experiments

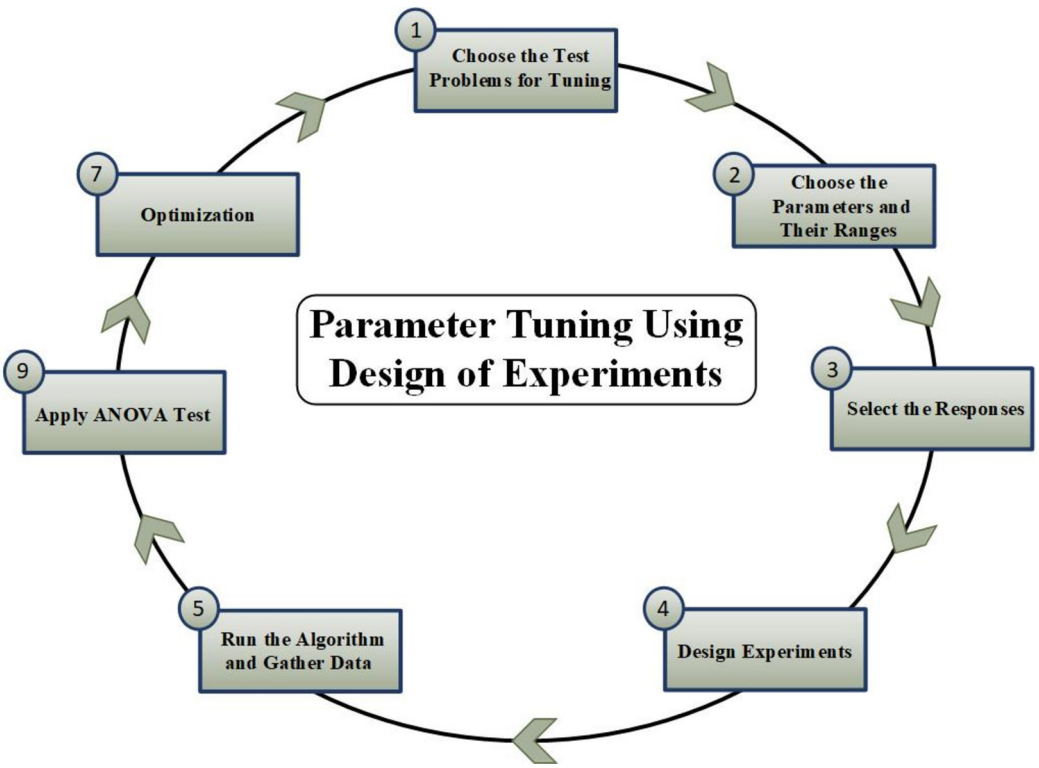


Table 5. Sample of runs

Run	Block	Factor 1	Factor 2	Factor 3	Factor 4	Response 1
		A:Bee_num	B:max_improvement_try	C:Pc_Uniform	D:pm	GAP
		Number	Number	Percent	Percent	Percent
1	0	150	150	0.2	0.015	
2	0	150	250	0.6	0.015	
3	0	250	250	0.2	0.015	
4	0	150	150	0.6	0.005	
5	0	250	250	0.6	0.005	
…	…	…	…	…	…	

4.4. Design of Experiments

For tuning metaheuristic algorithms, designing experiments is a critical step that involves designing a set of experiments to systematically explore the parameter space. The goal is to collect data that will be used to build a response surface model, allowing for a deeper understanding of how algorithm parameters impact performance. This involves selecting combinations of parameter values to assess their impact on the algorithm's performance. In this study, Central Composite Design (CCD) that introduced in Montgomery (2008), has been used. After entering the parameter values, the Design Expert software provided a total of 81 runs with different combinations of parameter values. A sample table of runs is presented in Table 5.

4.5. Running the Algorithm and Gathering Data

After designing the experiments, the next step is to execute the proposed algorithm for each set of the parameter values as defined by the experimental design. In this stage, the gap values of the responses obtained are calculated for further analysis. The calculated gaps have been entered into the Design Expert software.

4.6. ANOVA Test

After running the algorithm and collecting data, the next step is to apply the Analysis of Variance (ANOVA) test. ANOVA is a statistical technique used to analyze the variance in the response data and assess the significance of different factors (parameters) and their interactions. This step helps identify which parameters have a significant impact on the selected response. In Table 6, we delve into the implementation and analysis of the ANOVA test. Degrees of freedom (df) signifies the number of independent pieces of information present in the data, or in other words, the number of independent parameters used in computing the data.

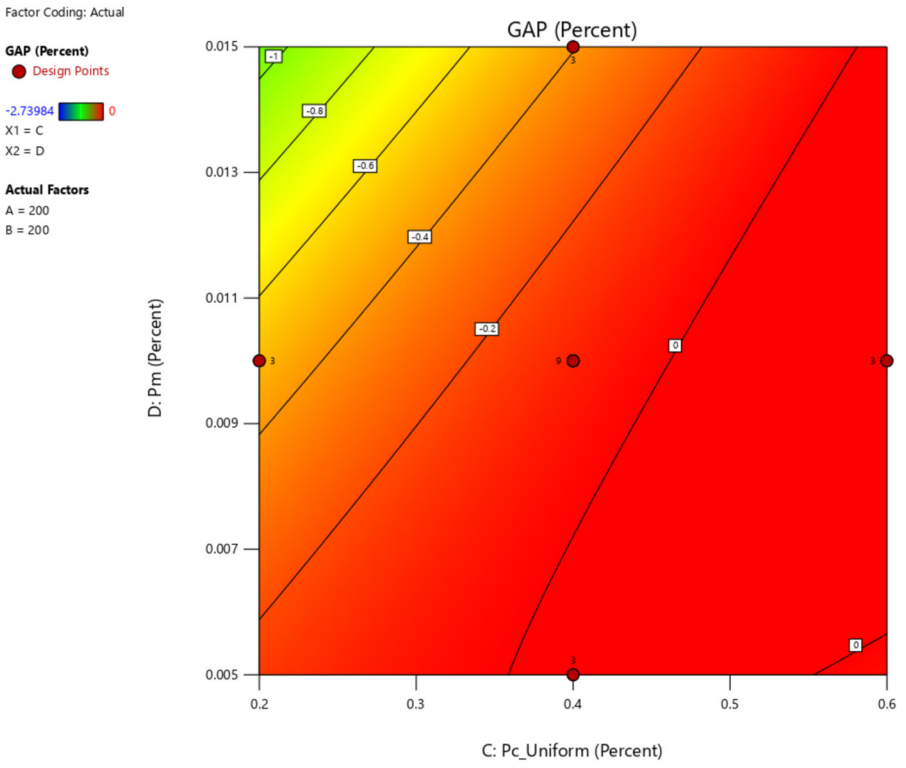
According to Table 6, the overall model is significant (p-value < 0.0001), indicating that at least one of the factors significantly influences the dependent variable. Factor B (*max_improvement_try*), Factor C (*Pc_Uniform*), and Factor D (*Pm*) are individually significant (p-values $= 0.0039$, < 0.0001, < 0.0001, respectively) and Factor A (*Bee_num*) is not significant. Among the interactions, AC and BC are significant (p-value $= 0.0012$, 0.0067), suggesting that the interaction between factors A and C and between B and C also have significant effects. Other interactions are not found to be statistically significant in this analysis.

In the following, we will proceed with the analysis of the ANOVA diagrams. Figure 8 illustrates the relationship between the GAP level and *Pc-Uniform* and *Pm*. As observed in Figure 8, with a decrease in the *Pm* and increase in *Pc-Uniform*, the GAP tends to decrease.

Table 6. ANOVA test outputs

Source	Sum of Squares	df	Mean Square	F-value	p-value	Status
Model	13.55	14	0.9679	15.22	< 0.0001	**significant**
A-Bee_num	0.0145	1	0.0145	0.2276	0.6349	
B-max_improvement_try	0.5703	1	0.5703	8.97	0.0039	
C-Pc_Uniform	5.07	1	5.07	79.65	< 0.0001	
D-Pm	2.43	1	2.43	38.27	< 0.0001	
AB	0.1592	1	0.1592	2.50	0.1185	
AC	0.7358	1	0.7358	11.57	0.0012	
AD	0.0159	1	0.0159	0.2502	0.6186	
BC	0.4989	1	0.4989	7.84	0.0067	
BD	0.0498	1	0.0498	0.7830	0.3795	
CD	2.86	1	2.86	45.02	< 0.0001	

Figure 8. Pm and Pc-uniform contour diagram



To illustrate the performance of parameter tuning, Figure 9 displays the actual values against the predicted values. It allows us to visually inspect how well the model's predicted fitness values align with the actual fitness values and identify any discrepancies or areas for improvement. In this

Table 7. The optimal parameter values

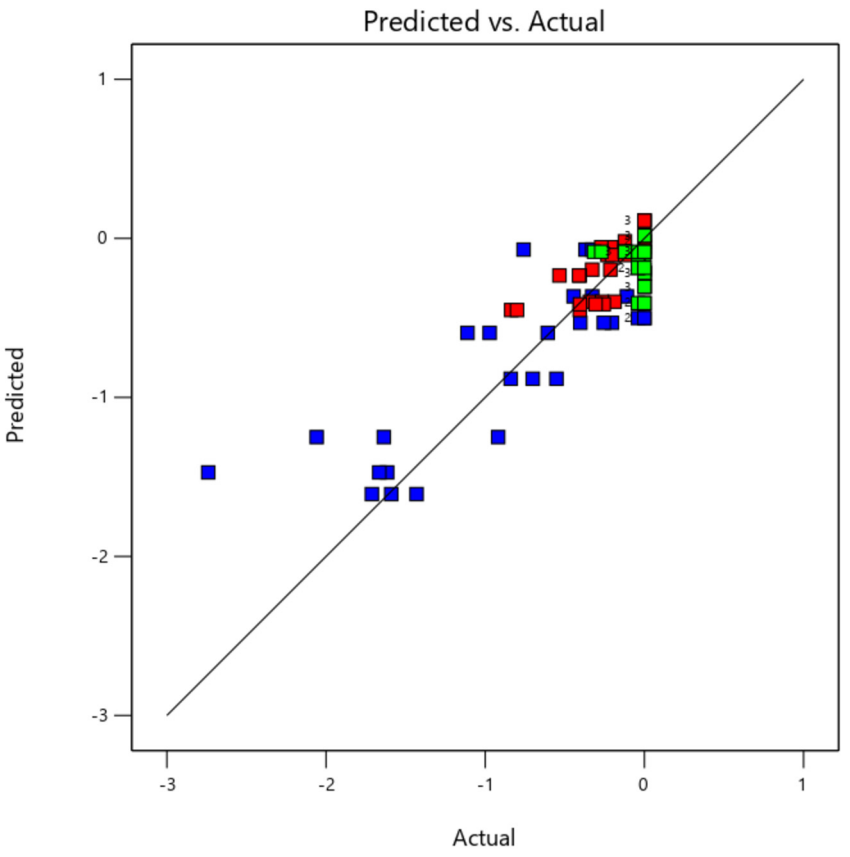
Row	Parameter	Unit	Optimal Parameter Value
1	*Bee_num*	Number	223
2	*max_improvement_try*	Number	226
4	*Pc_Uniform*	Percent	0.55
5	*Pm*	Percent	0.014

diagram, points closer to the line indicate accurate predictions. As observed in Figure 9, the most points are close to the line. This indicates the good performance of the statistical model.

4.7. Optimization of Parameters

The optimization of parameters refers to the process of finding the optimal values for input parameters in a DOE study. In this step, we want to optimize the model that are created in Step 6. In this step, the type of the objective function (target value, maximum, or minimum) needs to be specified and the well-established Nelder-Mead downhill simplex is then applied to optimize the function that explained in Design-Expert Tutorials (2022). The optimal values of the parameters were obtained and presented in Table 7.

Figure 9. Predicted vs. actual diagram



5. RESULTS

After obtaining the values of the proposed algorithm's parameters, this section focuses on evaluating the algorithm's performance on test problems. As mentioned before, the algorithm described has undergone testing using 30 benchmark instances of the MDKP with tightness of 0.25, 0.5, and 0.75. Each of these instances contains 100 items to be packed into knapsacks. Additionally, the problem instances are characterized by having 5 knapsacks available.

The program is run on a personal computer with Intel(R) Core (TM) i7-6650U CPU 2.20GHz and 8 GB RAM. The algorithm has been implemented using the Python programming language version 3.11.5, and the NumPy library version 1.26.2. For each problem, it has been executed 30 times, and each run has a time limit capped at 60 seconds.

Additionally, the obtained results are compared with the results from the published paper by He et al. (2019). In that paper, two hybrid algorithms, SACRO[1]-BPSO[2]-TVAC[3] and SACRO-CBPSO[4]-TVAC, have been introduced called SBT (SACRO-BPSO-TVAC) and SCT (SACRO-CBPSO-TVAC) here, respectively. The proposed algorithm in this paper has been compared with these two algorithms. Table 8 displays the results obtained from these three algorithms. In each row, the best results obtained are highlighted in bold font. In this table, the first column represents the problem number, and the second column indicates the tightness level. For each algorithm, there are three columns that present the best solution (Best), the average of solutions (Mean), and the standard deviation (Std) of solutions.

As shown in Table 8, our proposed algorithm achieved the best solution in 29 problems (97%) compared to the SBT and SCT. Out of these 29 problems, the proposed algorithm exclusively obtained high-quality solutions in 17 problems (57%). In 12 problems, all three algorithms reached identical solutions. Among the obtained mean solutions, the proposed algorithm had the highest mean in 26 problems (87%) compared to the other two algorithms. Additionally, in comparing the standard deviation (Std) of the obtained solutions, our proposed ABC algorithm had the lowest standard deviation compared SBT and SCT algorithms. The proposed algorithm has a better standard deviation in 27 problems. The results indicate that the proposed algorithm performs better in problems with high tightness levels.

In Table 9, the extent of improvement in solutions by the proposed algorithm compared to the other two algorithms indicates that this algorithm shows a significant improvement in terms of Best, Mean, and Std values compared to SBT and SCT. The average improvement in solutions for the proposed algorithm, compared to SBT and SCT, is 0.021% and 0.030%, respectively. Additionally, this algorithm has increased the average mean values compared to SBT and SCT by 0.1% and 0.09%, respectively. Comparison of standard deviations shows that the proposed ABC algorithm has reduced the standard deviation by 36.5%.

Figure 10 compares the gap values of the proposed algorithm with SBT and SCT algorithms. The numbers above zero indicate better solutions of our proposed algorithm. In Figures 11 and 12, the same type of comparison is done for the average solutions and their standard deviations. Comparing the gap values of the proposed algorithm with SCT indicates improvements in the best solutions in 13 problems, mean values in 27 problems, and standard deviations in 26 problems.

6. CONCLUSION

In this paper, an ABC algorithm was developed to enhance the quality of solutions generated for the MDKP. The proposed ABC algorithm incorporates a novel heuristic method to generate the employed bees, which significantly contributes to the efficiency of the search process. Additionally, the Design of Experiments (DOE) method was employed for parameter tuning, ensuring optimal performance of the algorithm. The effectiveness of the proposed ABC algorithm was rigorously evaluated using well-known benchmark problems. The quality of the solutions generated by the ABC algorithm was compared against results from a published paper, demonstrating substantial

Table 8. Algorithm's outputs

Prob. No.	Tig.	SBT			SCT			Proposed ABC Algorithm		
		Best	Mean	Std	Best	Mean	Std	Best	Mean	Std
0	0.25	24343	24296.0	49.20	24343	24301.7	45.30	**24381**	**24312.9**	**29.09**
1	0.25	**24274**	24153.5	44.57	**24274**	24153.3	64.67	**24274**	**24251.7**	**35.47**
2	0.25	23538	23510.2	24.74	23538	23509.7	33.17	**23551**	**23519.1**	**21.72**
3	0.25	23527	23432.9	57.41	23527	23457.6	50.27	**23534**	**23482.4**	**15.20**
4	0.25	23991	23932.5	37.59	**23966**	23924.3	39.56	**23966**	**23961.5**	**3.34**
5	0.25	24601	24516.4	64.26	24601	24546.2	66.42	**24613**	**24569.3**	**23.07**
6	0.25	**25591**	25440.1	58.63	**25591**	25451.0	65.48	**25591**	**25503.4**	**58.54**
7	0.25	**23410**	23321.8	67.56	**23410**	23349.0	46.47	**23410**	**23373.5**	**39.63**
8	0.25	24204	24163.5	46.86	**24216**	24148.5	62.90	**24216**	**24205.2**	**3.60**
9	0.25	24399	24295.2	56.44	**24411**	**24320.6**	67.43	**24411**	24294.4	**57.76**
10	0.50	42705	42660.9	48.73	42705	42666.2	32.49	**42757**	**42722.6**	**37.48**
11	0.50	42494	42441.8	26.50	42471	42434.2	33.57	**42545**	**42471.2**	**29.70**
12	0.50	41959	41904.0	36.11	41959	41904.8	32.11	**41967**	**41951.7**	**8.66**
13	0.50	**45090**	45021.8	31.63	**45090**	45010.0	35.17	**45090**	**45043.0**	**26.31**
14	0.50	**42218**	42149.4	60.37	**42218**	**42173.1**	56.35	**42218**	42170.2	**41.35**
15	0.50	**42927**	42899.2	**36.86**	**42927**	**42890.3**	45.15	**42927**	42870.1	47.71
16	0.50	**42009**	41904.0	54.30	**42009**	41871.4	63.49	**42009**	**41975.3**	**34.09**
17	0.50	45010	44910.7	65.67	**45020**	44948.8	**42.16**	**45020**	**44970.3**	44.27
18	0.50	**43441**	43300.9	62.78	43381	43290.0	**48.71**	**43441**	**43315.9**	87.76
19	0.50	**44554**	44493.8	37.41	44529	44472.4	45.34	44540	**44512.5**	**6.32**
20	0.75	**59822**	59776.4	78.76	**59822**	59791.9	73.47	**59822**	**59815.0**	**10.70**
21	0.75	**62081**	61936.9	68.42	**62081**	61960.3	58.38	**62081**	**62010.3**	**39.39**
22	0.75	**59802**	59696.5	46.56	59754	59693.3	46.11	**59802**	**59754.9**	**35.49**
23	0.75	60478	**60404.0**	66.05	60478	60388.8	91.93	**60479**	60438.1	**32.44**
24	0.75	61055	60996.4	49.77	61079	61005.6	43.54	**61091**	**61041.4**	**31.10**
25	0.75	**58959**	58904.2	43.54	58937	58886.3	47.92	**58959**	**58938.0**	**12.39**
26	0.75	**61538**	61437.2	57.87	**61538**	61469.1	46.88	**61538**	**61470.0**	**42.83**
27	0.75	61489	61411.0	46.08	**61520**	61428.1	70.13	**61520**	**61462.0**	**57.02**
28	0.75	**59453**	59301.2	68.92	**59453**	59295.1	65.54	**59453**	**59379.6**	**47.33**
29	0.75	**59960**	59928.2	51.00	**59960**	59943.6	32.95	**59960**	**59958.5**	**2.29**

improvements. Specifically, the performance comparison indicated that the proposed algorithm improved the best-known solutions for 13 benchmark problems, highlighting its ability to produce high-quality solutions. Furthermore, the proposed algorithm showed a significant reduction in the standard deviation (Std) of the solutions by 36.5% when compared to the SBT and SCT algorithms. This reduction in variability underscores the robustness and consistency of the ABC algorithm in generating optimal or near-optimal solutions. The improvement in the mean of the best values achieved by the proposed ABC algorithm compared to the SCT and SBT algorithms further indicates

Table 9. Comparing the improvement of proposed ABC algorithm solution

Problem No.	Tightness	Gap Best - SBT- ABC (%)	Gap Mean - SBT- ABC (%)	Gap Std - SBT- ABC (%)	Gap Best - SCT- ABC (%)	Gap Mean - SCT- ABC (%)	Gap Std - SCT- ABC (%)
0	0.25	**0.1561**	**0.0696**	**-40.9**	**0.1561**	**0.0461**	**-35.8**
1	0.25	0.0000	**0.4066**	**-20.4**	0.0000	**0.4074**	**-45.2**
2	0.25	**0.0552**	**0.0379**	**-12.2**	**0.0552**	**0.0400**	**-34.5**
3	0.25	**0.0298**	**0.2112**	**-73.5**	**0.0298**	**0.1057**	**-69.8**
4	0.25	-0.1042	**0.1212**	**-91.1**	0.0000	**0.1555**	**-91.6**
5	0.25	**0.0488**	**0.2158**	**-64.1**	**0.0488**	**0.0941**	**-65.3**
6	0.25	0.0000	**0.2488**	**-0.2**	0.0000	**0.2059**	**-10.6**
7	0.25	0.0000	**0.2217**	**-41.3**	0.0000	**0.1049**	**-14.7**
8	0.25	**0.0496**	**0.1726**	**-92.3**	0.0000	**0.2348**	**-94.3**
9	0.25	**0.0492**	-0.0033	2.3	0.0000	-0.1077	**-14.3**
10	0.5	**0.1218**	**0.1446**	**-23.1**	**0.1218**	**0.1322**	15.4
11	0.5	**0.1200**	**0.0693**	12.1	**0.1742**	**0.0872**	**-11.5**
12	0.5	**0.0191**	**0.1138**	**-76.0**	**0.0191**	**0.1119**	**-73.0**
13	0.5	0.0000	**0.0471**	**-16.8**	0.0000	**0.0733**	**-25.2**
14	0.5	0.0000	**0.0493**	**-31.5**	0.0000	-0.0069	**-26.6**
15	0.5	0.0000	-0.0678	29.4	0.0000	-0.0471	5.7
16	0.5	0.0000	**0.1702**	**-37.2**	0.0000	**0.2481**	**-46.3**
17	0.5	**0.0222**	**0.1327**	**-32.6**	0.0000	**0.0478**	5.0
18	0.5	0.0000	**0.0346**	39.8	**0.1383**	**0.0598**	80.2
19	0.5	-0.0314	**0.0420**	**-83.1**	**0.0247**	**0.0902**	**-86.1**
20	0.75	0.0000	**0.0646**	**-86.4**	0.0000	**0.0386**	**-85.4**
21	0.75	0.0000	**0.1185**	**-42.4**	0.0000	**0.0807**	**-32.5**
22	0.75	0.0000	**0.0978**	**-23.8**	**0.0803**	**0.1032**	**-23.0**
23	0.75	**0.0017**	**0.0565**	**-50.9**	**0.0017**	**0.0816**	**-64.7**
24	0.75	**0.0590**	**0.0738**	**-37.5**	**0.0196**	**0.0587**	**-28.6**
25	0.75	0.0000	**0.0574**	**-71.5**	**0.0373**	**0.0878**	**-74.1**
26	0.75	0.0000	**0.0534**	**-26.0**	0.0000	**0.0015**	**-8.6**
27	0.75	**0.0504**	**0.0830**	23.7	0.0000	**0.0552**	**-18.7**
28	0.75	0.0000	**0.1322**	**-31.3**	0.0000	**0.1425**	**-27.8**
29	0.75	0.0000	**0.0506**	**-95.5**	0.0000	**0.0249**	**-93.1**
Average		**0.0216**	**0.1075**	**-36.5**	**0.0302**	**0.0919**	**-36.5**

its superior performance. This enhancement in the mean values suggests that the ABC algorithm consistently finds better solutions across different instances of the MDKP.

The method described faces constraints as problem complexity increases, leading to longer solution times and diminished solution quality with larger dimensions. To address these challenges, hybrid algorithms combining exact and heuristic methods can be employed. Exact methods ensure

Figure 10. Proposed ABC- SBT-SCT best gap comparison

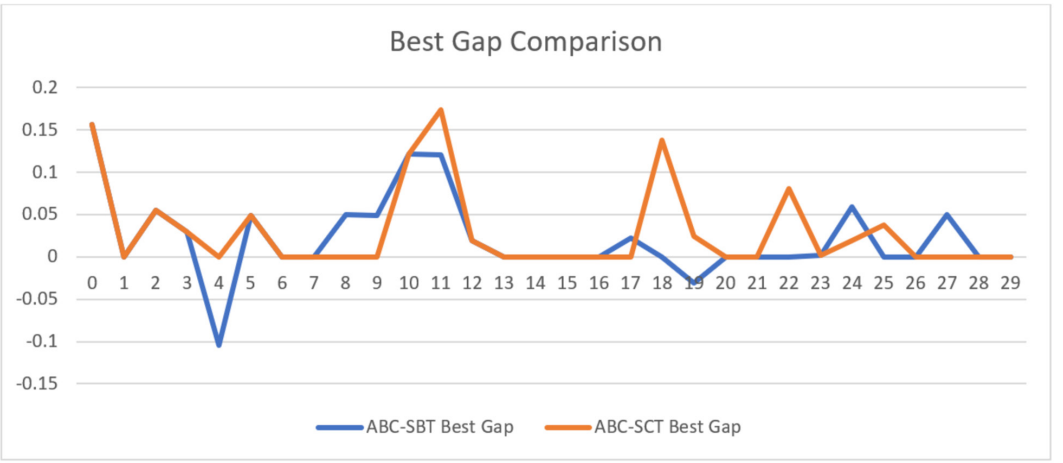
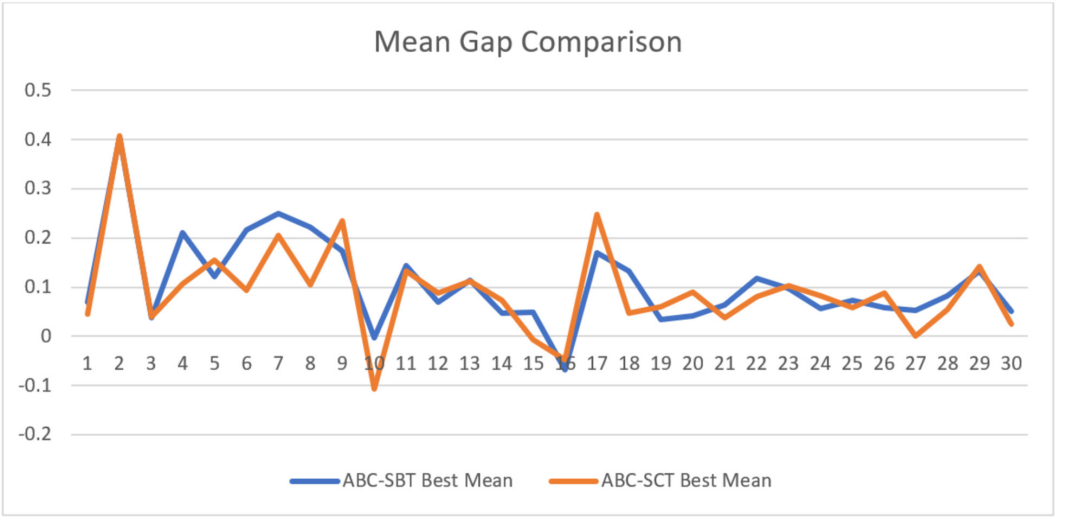
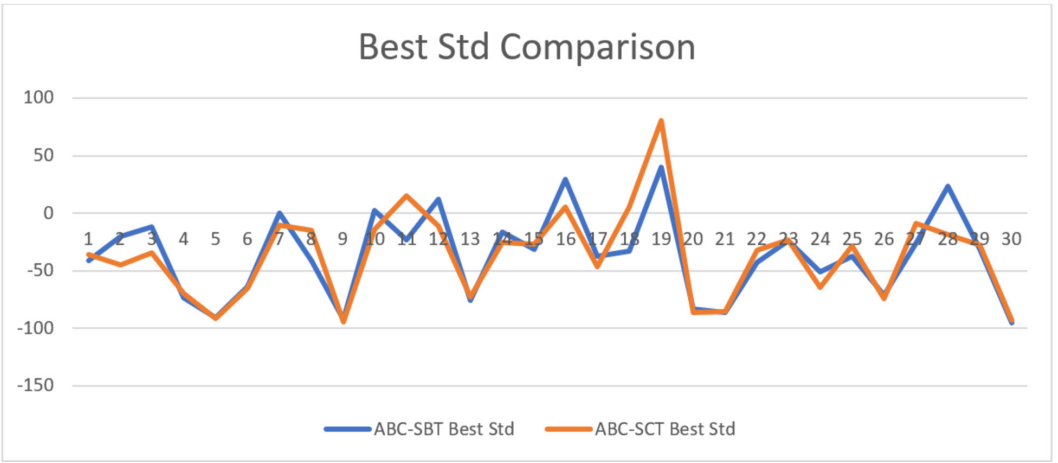


Figure 11. Proposed ABC- SBT-SCT mean gap comparison



optimal solutions but may lack scalability, while heuristics offer efficiency but may sacrifice optimality. By integrating both approaches, hybrid algorithms leverage the precision of exact methods and the scalability of heuristics, allowing for more effective exploration of the solution space and improved performance in terms of both solution quality and computational efficiency, particularly for complex optimization problems. The results of this future study should be compared with an exact method to validate the effectiveness and efficiency of the proposed ABC algorithm. Such a comparison would provide a benchmark for evaluating the trade-offs between solution quality and computational time. Overall, the results unequivocally demonstrate the superiority of the proposed ABC algorithm.

Figure 12. Proposed ABC- SBT-SCT std gap comparison



CONFLICTS OF INTEREST

We wish to confirm that there are no known conflicts of interest associated with this publication and there has been no significant financial support for this work that could have influenced its outcome.

FUNDING STATEMENT

No funding was received for this work.

PROCESS DATES

07, 2024

This manuscript was initially received for consideration for the journal on 04/13/2024, revisions were received for the manuscript following the double-anonymized peer review on 07/02/2024, the manuscript was formally accepted on 06/24/2024, and the manuscript was finalized for publication on 07/12/2024

CORRESPONDING AUTHOR

Correspondence should be addressed to Niusha Yaghini; niusha.yaghini@gmail.com

REFERENCES

Abdel-Basset, M., El-Shahat, D., & Sangaiah, A. K. (2017). A modified nature inspired meta-heuristic whale optimization algorithm for solving 0–1 knapsack problem. *International Journal of Machine Learning and Cybernetics*, *10*(3), 495–514. 10.1007/s13042-017-0731-3

Akay, B., & Karaboga, D. (2012). Artificial bee colony algorithm for large-scale problems and engineering design optimization. *Journal of Intelligent Manufacturing*, *23*(4), 1001–1014. 10.1007/s10845-010-0393-4

Angelelli, E., Mansini, R., & Grazia Speranza, M. (2010). Kernel search: A general heuristic for the multidimensional knapsack problem. *Computers & Operations Research*, *37*(11), 2017–2026. 10.1016/j.cor.2010.02.002

Beasley JE (2017) Orlib-operations research library.

Chih, M. (2015). Self-adaptive check and repair operator-based particle swarm optimization for the multidimensional knapsack problem. *Applied Soft Computing*, *26*, 378–389. 10.1016/j.asoc.2014.10.030

Chu, P. C., & Beasley, J. E. (1998). A Genetic Algorithm for the Multidimensional Knapsack Problem. *Journal of Heuristics*, *4*(1), 63–86. 10.1023/A:1009642405419

Dammeyer, F., & Voß, S. (1993). Dynamic tabu list management using the reverse elimination method. *Annals of Operations Research*, *41*(2), 29–46. 10.1007/BF02022561

Della Croce, F., & Grosso, A. (2012). Improved core problem-based heuristics for the 0/1 multidimensional knapsack problem. *Computers & Operations Research*, *39*(1), 27–31. 10.1016/j.cor.2011.03.013

Design-Expert Tutorials. (2022). *Multifactor RSM Tutorial (Part 2 – Optimization)*. State-Ease Inc.

Douglas, C. (2008). *Montgomery, Design and Analysis of Experiments, Seven Edition*. John Wiley & Sons, Inc.

Du, X., Zhou, J., Ni, Y., Liu, W., Xiao, R., & Wu, X. (2023). A novel binary multi-swarms fruit fly optimization algorithm for the 0-1 multidimensional knapsack problem. *International Journal of Bio-inspired Computation*, *21*(1), 1–10. 10.1504/IJBIC.2023.129982

Feng, Y., & Wang, G.-G. (2022). A binary moth search algorithm based on self-learning for multidimensional knapsack problems. *Future Generation Computer Systems*, *126*, 48–64. 10.1016/j.future.2021.07.033

Feng, Y., Wang, H., Cai, Z., Li, M., & Li, X. (2023). Hybrid Learning Moth Search Algorithm for Solving Multidimensional Knapsack Problems. *Mathematics*, *11*(8), 1811. 10.3390/math11081811

Gao, W. F., Liu, S. Y., & Huang, L. L. (2012). A global best artificial bee colony algorithm for global optimization. *Journal of Computational and Applied Mathematics*, *236*(11), 2741–2753. 10.1016/j.cam.2012.01.013

García, J., Crawford, B., Soto, R., Castro, C., & Paredes, F. (2017). A k-means binarization framework applied to multidimensional knapsack problem. *Applied Intelligence*, *48*(2), 357–380. 10.1007/s10489-017-0972-6

García, J., & Maureira, C. (2021). A KNN quantum cuckoo search algorithm applied to the multidimensional knapsack problem. *Applied Soft Computing*, *102*, 102. 10.1016/j.asoc.2020.107077

Gazioğlu, E. (2022). Solving Multidimensional Knapsack Problem with Bayesian Multiploid Genetic Algorithm. *Journal of Soft Computing and Artificial Intelligence*, *3*(2), 58–64. 10.55195/jscai.1216193

Glover, F., & Kochenberger, G. A. (1996). Critical Event Tabu Search for Multidimensional Knapsack Problems. In Meta-heuristics Springer.

Gupta, S., Su, R., & Singh, S. (2022). Diversified sine–cosine algorithm based on differential evolution for multidimensional knapsack problem. *Applied Soft Computing*, *130*, 130. 10.1016/j.asoc.2022.109682

Hanafi, S., & Freville, A. (1998). An efficient tabu search approach for the 0–1 multidimensional knapsack problem. *European Journal of Operational Research*, *106*(2-3), 659–675. 10.1016/S0377-2217(97)00296-8

He, Y., Zhang, X., Li, W., Wang, J., & Li, N. (2019). An efficient binary differential evolution algorithm for the multidimensional knapsack problem. *Engineering with Computers*, *37*(1), 745–761. 10.1007/s00366-019-00853-7

Hill, R. R., Kun Cho, Y., & Moore, J. T. (2012). Problem reduction heuristic for the 0–1 multidimensional knapsack problem. *Computers & Operations Research*, *39*(1), 19–26. 10.1016/j.cor.2010.06.009

Karaboga, D. (2005). An idea based on honey bee swarm for numerical optimization (Vol. 200, pp. 1-10). Technical report-tr06, Erciyes university, engineering faculty, computer engineering department.

Karaboga, D., & Akay, B. (2009). A comparative study of artificial bee colony algorithm. *Applied Mathematics and Computation*, *214*(1), 108–132. 10.1016/j.amc.2009.03.090

Karaboga, D., Akay, B., & Ozturk, C. (2007). Artificial bee colony (ABC) algorithm on training artificial neural networks. In *Proceedings of the 2007 IEEE congress on evolutionary computation* (pp. 325-331). 10.1109/SIU.2007.4298679

Karaboga, D., & Basturk, B. (2007). A powerful and efficient algorithm for numerical function optimization: Artificial bee colony (ABC) algorithm. *Journal of Global Optimization*, *39*(3), 459–471. 10.1007/s10898-007-9149-x

Karaboga, D., & Basturk, B. (2008). On the performance of artificial bee colony (ABC) algorithm. *Applied Soft Computing*, *8*(1), 687–697. 10.1016/j.asoc.2007.05.007

Karaboga, D., & Ozturk, C. (2011). A novel clustering approach: Artificial Bee Colony (ABC) algorithm. *Applied Soft Computing*, *11*(1), 652–657. 10.1016/j.asoc.2009.12.025

Ke, L., Feng, Z., Ren, Z., & Wei, X. (2010). An ant colony optimization approach for the multidimensional knapsack problem. *Journal of Heuristics*, *16*(1), 65–83. 10.1007/s10732-008-9087-x

Kellerer, H., Pferschy, U., & Pisinger, D. (2004). *Knapsack problems*. Springer-Verlag. 10.1007/978-3-540-24777-7

Lai, X., Hao, J.-K., Fu, Z.-H., & Yue, D. (2020). Diversity-preserving quantum particle swarm optimization for the multidimensional knapsack problem. *Expert Systems with Applications*, *149*, 149. 10.1016/j.eswa.2020.113310

Lai, X., Hao, J.-K., Glover, F., & Lü, Z. (2018). A two-phase tabu-evolutionary algorithm for the 0–1 multidimensional knapsack problem. *Information Sciences*, *436-437*, 282–301. 10.1016/j.ins.2018.01.026

Li, X., Fang, W., Zhu, S., & Zhang, X. (2024). An adaptive binary quantum-behaved particle swarm optimization algorithm for the multidimensional knapsack problem. *Swarm and Evolutionary Computation*, *86*, 101494. 10.1016/j.swevo.2024.101494

Liu, J., Wu, C., Cao, J., Wang, X., & Teo, K. L. (2016). A Binary differential search algorithm for the 0–1 multidimensional knapsack problem. *Applied Mathematical Modelling*, *40*(23-24), 9788–9805. 10.1016/j.apm.2016.06.002

Luo, K., & Zhao, Q. (2019). A binary grey wolf optimizer for the multidimensional knapsack problem. *Applied Soft Computing*, *83*, 83. 10.1016/j.asoc.2019.105645

Meng, T., & Pan, Q.-K. (2017). An improved fruit fly optimization algorithm for solving the multidimensional knapsack problem. *Applied Soft Computing*, *50*, 79–93. 10.1016/j.asoc.2016.11.023

Mingo López, L. F., Gómez Blas, N., & Arteta Albert, A. (2017). Multidimensional knapsack problem optimization using a binary particle swarm model with genetic operations. *Soft Computing*, *22*(8), 2567–2582. 10.1007/s00500-017-2511-0

Mkaouar, A., Htiouech, S., & Chabchoub, H. (2023). Modified artificial bee colony algorithm for multiple-choice multidimensional knapsack problem. *IEEE Access : Practical Innovations, Open Solutions*, *11*, 45255–45269. 10.1109/ACCESS.2023.3264966

Olivares, R., Soto, R., Crawford, B., Ríos, V., Olivares, P., Ravelo, C., Medina, S., & Nauduan, D. (2023). A learning—Based particle swarm optimizer for solving mathematical combinatorial problems. *Axioms*, *12*(7), 643. 10.3390/axioms12070643

Petersen, C. C. (1967). Computational Experience with Variants of the Balas Algorithm Applied to the Selection of R&D Projects. *Management Science*, *13*(9), 736–750. 10.1287/mnsc.13.9.736

Scherer, M. E., Hill, R. R., Lunday, B. J., Cox, B. A., & White, E. D. (2024). Verifying new instances of the multidemand multidimensional knapsack problem with instance space analysis. *Computers & Operations Research*, *162*, 106477. 10.1016/j.cor.2023.106477

Vega, E., Lemus-Romani, J., Soto, R., Crawford, B., Löffler, C., Peña, J., & Talbi, E. G. (2024). Autonomous Parameter Balance in Population-Based Approaches: A Self-Adaptive Learning-Based Strategy. *Biomimetics*, *9*(2), 82. 10.3390/biomimetics902008238392128

Wang, L., Yang, R., Ni, H., Ye, W., Fei, M., & Pardalos, P. M. (2015). A human learning optimization algorithm and its application to multidimensional knapsack problems. *Applied Soft Computing*, *34*, 736–743. 10.1016/j.asoc.2015.06.004

Weingartner, H. M., & Ness, D. N. (1967). Methods for the Solution of the Multidimensional 0/1 Knapsack Problem. *Operations Research*, *15*(1), 83–103. 10.1287/opre.15.1.83

Wilbaut, C., Salhi, S., & Hanafi, S. (2009). An iterative variable-based fixation heuristic for the 0-1 multidimensional knapsack problem. *European Journal of Operational Research*, *199*(2), 339–348. 10.1016/j.ejor.2008.11.036

Yaghini, M., Momeni, M., & Sarmadi, M. (2012). A DIMMA-Based Memetic Algorithm for 0-1 Multidimensional Knapsack Problem Using DOE Approach for Parameter Tuning. *International Journal of Applied Metaheuristic Computing*, *3*(2), 43–55. 10.4018/jamc.2012040104

Zhang, B., Pan, Q.-K., Zhang, X.-L., & Duan, P.-Y. (2015). An effective hybrid harmony search-based algorithm for solving multidimensional knapsack problems. *Applied Soft Computing*, *29*, 288–297. 10.1016/j.asoc.2015.01.022

ENDNOTES

[1] Self-Adaptive Crossover and Repair Operator (SACRO)
[2] Binary Particle Swarm Optimization (BPSO)
[3] Time-Varying Acceleration Coefficients (TVAC)
[4] Comprehensive Binary Particle Swarm Optimization (CBPSO)

Niusha Yaghini is a Computer Engineering student at Iran University of Science and Technology, who won the second place in the Genetic Algorithms section of her university's machine learning competition. She served as a teacher's assistant for Fundamental Computer Programming, Advanced Programming, and Advanced Algorithms courses, and also led a Python programming language course in her university. She is passionate about technology, artificial intelligence and metaheuristic algorithms.

Mir Yasin Seyed Valizadeh graduated from Iran University of Science and Technology with a master's degree in Rail Transportation Engineering. He has a bachelor's degree in Industrial Engineering. His research interests are primarily focused on Operations Research, Robust Optimization, and the Strategic Planning and Timetabling of transportation systems. Additionally, he harbors a profound interest in Heuristic and Metaheuristic Algorithms, aiming to make a substantial contribution to the field of transportation engineering.