Journal of Electrical and Computer Engineering Innovations
(JECEI)
Journal homepage: http://www.jecei.sru.ac.ir

Research paper

Using β-Hill Climbing Optimizer to Generate Optimal Test Suite

Seyyed AmirHossein Eshghazadi , ***Einollah Pira*** * , ***Mohammad Khodizadeh-Nahari*** , ***Alireza Rouhi***

Faculty of Information Technology and Computer Engineering, Azarbaijan Shahid Madani University, Tabriz, Iran.

Article Info

Article History:
Received 15 March 2025
Reviewed 08 June 2025
Revised 18 May 2025
Accepted 12 July 2025

Keywords:
Hill climbing
Meta-heuristic
Optimizer
Software testing
Test suite

*Corresponding Author's Email Address: *pira@azaruniv.ac.ir*

Abstract

Background and Objectives: Software testing plays a vital role in software development, aimed at verifying the reliability and stability of software systems. The generation of an effective test suite is key to this process, as it directly impacts the detection of defects and vulnerabilities. However, for software systems with numerous input parameters, the combinatorial explosion problem hinders the creation of comprehensive test suites. This research introduces a novel approach using the β-Hill Climbing optimizer, an advanced variant of the traditional hill climbing algorithm, to efficiently generate optimal test suites.

Methods: The β-Hill Climbing optimizer introduces a dynamic parameter, β, which facilitates a precise balance between exploration and exploitation throughout the search process. To evaluate the performance of this proposed strategy (referred to as BHC), it is compared with TConfig as a mathematical approach, PICT and IPOG as greedy algorithms, and GS, GALP, DPSO, WOA, BAPSO, and GSTG as meta-heuristic methods. These strategies are tested across a variety of configurations to assess their relative efficiency.

Results: The reported results confirm that BHC outperforms the others in terms of the size of generated test suites and convergence speed. The statistical analysis of the experimental results on several different configurations shows that BHC outperforms TConfig as a mathematical strategy, PICT and IPOG as greedy strategies, GS, GALP, DPSO, WOA, BAPSO, and GSTG as meta-heuristics by 83%, 88%, 87%, 61%, 61%, 46%, 61%, 62%, and 70%, respectively.

Conclusion: The BHC strategy presents a novel and effective approach to optimization, inspired by *β*-Hill Climbing optimizer for the generation of an optimal test suite. It has superior performance in the generation of test suites with a smaller size and higher convergence speed compared to other strategies.

How to cite this paper:
S. A. Eshghazadi, E. Pira, M. Khodizadeh-Nahari, A. Rouhi "Using β-Hill climbing optimizer to generate optimal test suite," J. Electr. Comput. Eng. Innovations, 14(1): 83-98, 2026.

DOI: 10.22061/jecei.2025.11277.787

URL: https://jecei.sru.ac.ir/article_2367.html

Introduction

Software testing plays a fundamental role in the software development process, ensuring that software systems operate reliably and meet predefined quality standards [1]. This validation process encompasses a range of methodologies, including white box testing and black box testing. White box testing scrutinizes the internal workings of software to identify potential flaws before release. Model checking is a formal method, belonging to white box testing methods that systematically examine the properties of a system by exploring all possible states of a model of a system. However, this method faces challenges with state space explosion in large and complex systems, as the memory requirements grow exponentially [2]. Black box testing evaluates software functionality from an end-user perspective to ensure alignment with user requirements [3]. The integration of these approaches aims to produce an optimal test suite that guarantees the software's expected performance under diverse conditions.

Nevertheless, crafting an optimal test suite that achieves comprehensive coverage across all conceivable scenarios remains a formidable challenge, particularly for intricate software systems. This challenge is exacerbated by the combinatorial explosion problem, wherein the number of necessary test cases grows exponentially as the quantity of input parameters and their potential values expands. Various techniques have emerged to confront this issue, including t-way testing, which focuses on examining combinations of input parameters that significantly influence system behavior [4].

This paper presents an innovative approach that leverages t-way testing in conjunction with the β-Hill Climbing Optimizer, a metaheuristic algorithm [5], to construct an optimal test suite that tackles the combinatorial explosion problem while maintaining a manageable size. Our strategy (so-called BHC) harnesses the β-Hill Climbing Optimizer to iteratively enhance candidate test suites by guiding them in the direction of steepest ascent until a satisfactory solution is reached. Meanwhile, the t-way testing technique identifies a subset of input combinations with a high likelihood of detecting software defects while keeping the test suite's size within bounds. This optimization method has demonstrated its effectiveness in producing optimal test suites that encompass a broad spectrum of test cases.

To evaluate and compare the efficiency of BHC, it is benchmarked against TConfig as a mathematical approach, PICT and IPOG as greedy algorithms, and GS, GALP, DPSO, WOA, BAPSO, and GSTG as meta-heuristic techniques. They are experimented on several different configurations. Our experimental evaluation illustrates that our strategy surpasses considered strategies, drastically reducing the number of required test cases to achieve comprehensive coverage.

Our investigation aims to spotlight the development of a comprehensive and effective testing methodology, merging t-way testing with the *β*-Hill Climbing Optimizer, to address the combinatorial explosion problem. This methodology, we argue, generates an optimal test suite that elevates software product quality and reliability, thereby mitigating potential issues that could impact end-users.

The remainder of the paper is structured as follows. Initially, we present the relevant background, including an overview of the BHC algorithm and the t-way strategy. Following that, the existing research on software testing and optimization, with a focus on metaheuristic algorithms for generating optimal test suites, will be provided. In the following, we introduce our proposed approach, utilizing the β-Hill Climbing Optimizer for comprehensive test suite generation. Then, the effectiveness of this method, comparing it to existing techniques and assessing the quality of the generated test suites, will be presented. Finally, the conclusion section concludes the paper and discusses directions for future research.

Background

A. β-Hill Climbing Optimizer

Hill-climbing optimizers are iterative optimization techniques that aim to improve a solution iteratively by making local changes. Over time, various versions of hill-climbing have been developed to address its inherent limitations, such as getting stuck in local optima or inefficiency in rugged landscapes. In the continuation of this section, we review the most notable versions. Basic hill climbing [6] , is the simplest version of this optimizer that moves to the best neighboring solution at each iteration, always choosing the most favorable option. First-choice hill climbing [7] is another version of this optimizer, which randomly evaluates neighbors and chooses the first one that is better than the current solution. Other versions of this optimizer include: random-restart hill climbing (repeatedly performs basic hill climbing from randomly generated initial states, keeping track of the best solution found) [8], simulated annealing (hill climbing with cooling) [9], steepest-ascent hill climbing (evaluates all neighbors and chooses the one with the steepest increase in fitness (largest improvement)) [10], parallel hill climbing (runs multiple hill-climbing processes in parallel from different starting points) [11], tabu search (memory-based hill climbing) [12] and β-hill climbing optimizer (introduces a tunable parameter β to balance exploration and exploitation) [5]. Table 1 covers the various versions of hill-climbing optimizers and highlights their unique strengths and

weaknesses. Moreover, this optimizer is hybrid with other algorithms such as PSO [13] and GA [14].

The Beta Hill Climbing (BHC) optimizer is an advanced iteration of the classic hill climbing algorithm, incorporating an exploratory operator denoted as β [5]. This operator, inspired by the uniform mutation operator found in genetic algorithms, equips BHC with the ability to navigate across diverse regions within the search space. This enables BHC to break free from local optima by experimenting with random values to choose for decision variables. Unlike the basic hill climbing algorithm [15], BHC effectively balances both exploration and exploitation, mitigating the risk of converging prematurely into local optima.

Table 1: The summary of strengths and weaknesses of various versions of hill-climbing optimizers

Variant	Strengths	Weaknesses
Basic Hill Climbing	Simple and fast	Prone to local optima
First-Choice Hill Climbing	Faster than stochastic	Settles for suboptimal moves
Random-Restart Hill Climbing	Escapes local optima effectively	High computational cost
Simulated Annealing	Good exploration-exploitation balance	Slow convergence
Steepest-Ascent	Finds locally best path quickly	Expensive for large neighborhoods
Parallel Hill Climbing	Increases global optima likelihood	High computational demand
Tabu Search	Avoids cycles in search	Requires memory management
β-HCO	Dynamic exploration-exploitation control	Computationally complex

The BHC algorithm starts with an initial solution $x = \{x_1, x_2, \ldots, x_k\}$ that is randomly generated within the bounds of the search space, where the value of each variable x_i lies within $[lb_i, ub_i]$. Then, the value of the objective function $f(x)$ is calculated, and a new solution x' is created by applying N and β operators on the current solution x. This process continues until the number of iterations falls below the specified maximum, *maxIterations*.

The N operator selects a neighboring solution x' from the current solution x as follows: First, it randomly chooses one of the variables in the current solution, denoted as variable j-th. Then, it modifies the value of this variable using (1):

$$x'_j = x_j + rnd * bw \quad (1)$$

here, bw represents the bandwidth between the current value and the new value, and rnd is a random number between zero and one. It's worth noting that all variable values in the new solution x', except for the j-th variable, remain identical to those in the current solution x.

The β operator generates a new solution based on the existing solution x as follows: It assigns values to the variables in the new solution using two different strategies. It either assigns values based on the values in the current solution with a probability of (1-β) or randomly samples values from the available range with a probability of β. For a more detailed representation of the algorithm, see Algorithm 1 in the provided pseudocode.

Algorithm 1: The BHC pseudocode

Input: *maxIterations*: the maximum number of iterations; *k*: the dimension of the given problem; *lb*, *ub*, *fit*: the fitness function; *bw* , *beta*;

Output: An individual with the highest fitness;

```
Individual x = Initialize (k, lb, ub);
rep =1;
while rep <= maxIterations do
    Individual x' = x;
        //// The N operator: This operator generates new candidate solutions by adding normally distributed noise, aiding in local exploitation around the current solution.
    index = a random integer number between 1 and k;
    rnd = a random number between 0 and 1;
    x' [index] = x [index] +(2*rnd-1)*bw;
        //// The β operator: This operator diversifies the search by using controlled random perturbations to escape local optima and explore new regions of the solution space.
        //////
    for j = 1 to k do
        if rand(0,1) <=beta then
            x' [j] = lb[j]+(ub[j]-lb[j])* rand(0,1);
        end if
    end for
    if fit (x') > fit (x) then
        x = x';
    end if
    rep ++;
end while
return x;
```

Overall, BHC stands out as a straightforward yet highly efficient optimization algorithm, renowned for its capability to surmount local optima and navigate across diverse territories within the search space. At its core, BHC leverages an exploratory operator, denoted as β, which draws inspiration from genetic algorithms. This distinctive feature empowers the algorithm to execute random leaps, fostering the potential for accelerated convergence towards the global optimum. In essence, the N and β operators within BHC can be viewed as complementary components, with N primarily serving as an exploitation mechanism and β serving as the key source of exploration in this algorithm.

B. The T-Way Strategy

Comprehensive testing of a software system requires accounting for all possible combinations of input parameters in the test cases. However, when dealing with systems that have a large number of input parameters, this approach can result in a challenge known as the "combinatorial explosion" problem.

To tackle this challenge, the t-way combinatorial testing technique is employed. Instead of covering all conceivable combinations, it focuses on generating a set of test cases that encompasses only t combinations of system parameters [16]. The effectiveness of this strategy hinges on selecting an appropriate value for t.

Selecting a small t could result in missing critical combinations needed to detect errors in the generated test suite. On the other hand, using a large t may bring back the issue of combinatorial explosion. In this method, t is known as the "coverage power", which dictates the depth of coverage. The test suite generated using the t-way approach is called a "covering array" (CA).

For a system that takes p parameters as input, where each parameter can assume di ($1 \le i \le p$) possible values, the covering array is denoted as $CA(N; t, p, d_1, \ldots, d_p)$, where N signifies the number of test cases, and t denotes the coverage strength. Assuming all d_i ($1 \le i \le p$) are equal to d, the covering array can be abbreviated as $CA(N; t, p, d)$ or simply $CA(N; t, d^p)$.

Let's illustrate this concept with a hypothetical medical system featuring six input parameters: patient age, blood pressure, heart rate, cholesterol level, diabetes status, and smoking status (Table 2). Suppose age can be one of two values, blood pressure can be one of two values, heart rate can be one of two values, cholesterol level can be one of three values, and smoking status can be one of three values.

To comprehensively test this system, testing all possible combinations of input parameters would entail 2*2*2*3*3 = 72 test cases. However, this exhaustive approach can result in a combinatorial explosion, rendering it impractical to test every single combination.

Table 2: Hypothetical medical system

Patient age	Blood pressure	Heart rate	Cholesterol level	Smoking status
<18	Normal	Normal	High	Healthy
>18	Elevated	Bradycardia	Normal	Addicted
			low	Sometimes

To address this issue, we can employ t-way testing. This method generates a set of test cases that encompasses only t combinations of input parameters. For instance, if we select t = 2, we would only need to examine 9 test cases, covering pairs of input parameters (Table 3). This approach substantially cuts down on the necessary number of test cases while ensuring sufficient coverage to identify most errors.

Table 3: Covering array CA (9; 2, 23, 32)

#	patient age	blood pressure	heart rate	cholesterol level	smoking status
1	<18	Elevated	Bradycardia	low	Addicted
2	>18	Normal	Normal	Normal	Addicted
3	>18	Normal	Bradycardia	High	Healthy
4	<18	Elevated	Normal	High	Sometimes
5	<18	Elevated	Normal	Normal	Healthy
6	>18	Normal	Normal	low	Sometimes
7	<18	Normal	Bradycardia	Normal	Sometimes
8	>18	Elevated	Normal	High	Addicted
9	>18	Elevated	Normal	low	Healthy

Related Work

The t-way strategy causes the number of test cases in a complete (non-minimal) covering array to grow exponentially as the interaction strength, represented by t, increases. To address this challenge, various methods have been proposed in the literature. Some approaches, such as Combinatorial Test Services (CTS) [17] and Tconfig [18], utilize mathematical concepts like orthogonal arrays (OA). However, these techniques often face difficulties in generating optimal covering arrays, particularly when dealing with larger or more complex configurations.

In contrast, approximate techniques require less time to identify nearly ideal covering arrays and can be categorized into two main groups: greedy strategies and metaheuristics.

C. Greedy Strategies

In this section, we delve into two key methods employed by greedy strategies [19]: "One-Row-at-a-

Time" (ORT) and "One-Parameter-at-a-Time" (OPT).

ORT Method:

- ORT builds the covering array (*CA*) incrementally, row by row, ensuring optimal coverage.
- The "Automatic Efficient Test Generator" (AETG) was the first ORT-based strategy. It selects a test case from multiple candidates in a greedy manner [20].
- Alternatives to AETG include "mAETG" and "mAETG-SAT" [21].
- "PICT" is another ORT-based strategy that generates interactions while randomly selecting required test cases. However, its randomness can result in inconsistent results [22].
- "Jenny" adopts a unique approach, initially considering 1-way interactions and gradually incorporating higher-order interactions. It is known for its speed and ability to produce compact covering arrays for many configurations [23].

ORT-Based Classification-Tree Method:

- The "Classification-Tree Editor eXtended Logics" (CTE-XL) partitions the input domain into subsets and combines them to create test cases, effectively addressing $t = 3$ interactions [24].
- "GTWay" is another ORT-based strategy that generates test suites up to $t = 12$. It employs bit structures to store test case components, with an index table facilitating efficient access [25].

OPT Method:

- OPT, as the name suggests, expands the covering array by gradually adding more parameters.
- Initially, it creates a *CA* for two parameters, progressively incorporating additional parameters and their interactions.
- Notable OPT-based strategies include "IPOG-F" [26], and SCIPOG [27].

These strategies offer diverse approaches to efficiently construct covering arrays, each suited to different testing requirements and scenarios.

D. Meta-Heuristic Strategies

Meta-heuristic strategies aim to find near-optimal covering arrays efficiently, while also avoiding local optima. These methods, similar to ORT, select test cases with the highest weight from a list of candidates. They operate as follows:

- Generating Candidates: A group of potential test cases is randomly generated.
- Increasing Weights: Several operators are applied to increase the weights of these candidates.
- Expanding Test Suite: The test suite grows by selecting the test case with the highest weight.

This process repeats until all possible combinations of the t input parameters are accounted for. Metaheuristic algorithms are generally classified into nine categories [28], [29]: biology-based, physics-based, social-based, music-based, chemical-based, sport-based, mathematics-based, swarm-based, and hybrid algorithms. The following are key strategies within these categories.

- Social-Based (TLBO) [30]**:** TLBO draws inspiration from classroom learning environments. It involves three stages: population creation, training, and learning. TLBO outperforms other strategies like TConfig, IPOG, Jenny, and PICT, generating compact covering arrays for various configurations, even supporting higher strengths up to $t = 15$.
- Swarm-Based (PSO) [31]**:** This approach models test cases as birds searching for food. Birds represent test cases with positions and speeds. The PSO algorithm [32] and its variants, like BAPSO [33], are based on this concept, supporting higher strengths (BAPSO: up to $t = 16$). However, they do not handle variable strength interactions and have issues with parameter settings and early convergence.
- Physics-Based (GSTG) [34]**:** GSTG draws inspiration from gravitational interactions. Test cases act as objects with mass, and their gravitational pull influences their movement. This strategy can generate covering arrays up to $t = 16$ but lacks support for variable strength interactions.
- Biology-Based (GA and GS): GA [35] uses an evolutionary principle, creating new test cases from the current population through crossover and mutation. GS [16] builds on GA, continually applying crossover and mutation, achieving efficiency and supporting up to $t = 20$. It can handle variable strength interactions but not constraints.
- Multiple Black Hole (MBH) [36]**:** Inspired by black holes and the behavior of stars, MBH moves test cases toward "black holes" with more energy (heavier test cases). It can generate covering arrays up to $t = 4$, but cannot handle variable strength interactions or constraints.
- Mathematics-Based (SCAVS) [37], [38]**:** SCAVS uses the sine cosine algorithm to solve the problem. It can generate covering arrays up to $t = 6$ but does not support constraints, although it addresses variable strength interactions.
- Swarm-Based (ABC [39] and ABCVS [40]): These strategies simulate the social behavior of honey bees. HABC [41]-[45] offers interactions with variable strength ($t \leq 6$) and supports constraints. PhABC [45], [46] is a more recent iteration.

Additional examples of metaheuristic strategies include: HC-BAT (hybrid-based) [47], ABO (swarm-based)

[48], BDA (swarm-based) [49], LSHADE (biology-based) [50], GALP (hybrid-based) [51], ACOF (swarm-based) [52], FATG (swarm-based) [53], SCA (mathematics-based) [54], HAS (meta-heuristic-based) [55], [56], HGHC (hybrid-based) [57], QWOA-EMC (hybrid-based) [58], QSMA (meta-heuristic-based) [59], HHOA (meta-heuristic-based) [60], BOA (meta-heuristic-based) [61], TWGH (meta-heuristic-based) [62], TPA (hybrid-based) [63], ImpARO (meta-heuristic-based) [64], ROBDDs (meta-heuristic-based) [65], and SCHOP (meta-heuristic-based) [56].

The Presented Strategy

Given the uncertainty surrounding the size of the optimal covering array for test set generation, metaheuristic-based strategies require multiple runs of these algorithms to generate test samples with maximum coverage. The algorithms are repeatedly executed until all possible combinations of the *t* coverage parameter are fully addressed. When using any metaheuristic algorithm to solve the optimal test set generation problem, two key adjustments are made to its structure:

- Each test sample is represented as a solution (chromosome or individual).
- A weight calculation function, acting as a fitness function, is employed to compute the weight of a test sample (representing the number of covered combinations).

In the context of generating an optimal covering array with *p* parameters, each taking values within the range [0, *d*-1], the covering array is denoted as *CA*(*N*; *t*, *p*, *d*). To utilize the Beta Hill Climbing (BHC) algorithm for generating a test sample with maximum weight (i.e., covering the most combinations), we need to define the structure of chromosomes and the fitness function.

- Chromosomes are represented as vectors of length *p*, denoted as (v1, ..., vp), to represent each test sample. Each gene vi takes values within the range [0, *d*-1].
- The fitness function serves as a weight calculation function, computing the weight of a test sample, which represents the number of covered combinations.

Following the principles outlined in Algorithm 1, the BHC algorithm starts by randomly generating an initial solution (test sample) where each gene lies within the range [0, *d*-1]. It then iteratively enhances this test sample by applying the *N* and *β* operators for a maximum number of iterations specified as *maxIterations*.

In the following sections, we delve into the details of the BHC strategy, including the creation of the coverage matrix, the method for calculating the weight of a test sample, and the updating of the coverage matrix.

According to Algorithm 1, BHC first generates an initial solution (test sample) randomly, where the value of each gene lies within the range [0, *d*-1]. Then, it improves this test sample by applying the *N* and *β* operators for a maximum number of iterations called maxIterations. In the following section, the BHC strategy is presented, and subsequently, the creation of the coverage matrix, the method for calculating the weight of a test sample, and updating the coverage matrix are explained.

E. BHC Strategy Details

Fig. 1 shows the flowchart of the BHC strategy. The BHC strategy is employed iteratively to create a test sample with maximum coverage. It relies on a coverage matrix (*CM*) to monitor the coverage status of various combinations and consists of three key stages:

- Initialization: In this phase, the coverage matrix (*CM*) is established, and a variable called *RemCov*, representing the number of remaining uncovered combinations, is initialized to $\binom{p}{t} \times d^t$ .Additionally, an empty test set (TS) is created.
- BHC Algorithm Execution: The BHC algorithm is executed in this stage to produce a test sample, saved in the variable *tc*.
- Update Stage: In this phase, the weight of the previously generated test sample (*tc*) is calculated using the fitness function *calcWeight*, and this value is subtracted from the *RemCov* total. The test sample *tc* is subsequently added to the test set *TS*, and the coverage matrix *CM* is updated accordingly.

The BHC algorithm and update stage continue until either *RemCov* reaches *zero* (indicating that all possible combinations of the *t* parameters are covered by the generated test set) or the execution time surpasses 24 hours. At the conclusion of the strategy, the generated test set *TS* is presented to the user.

For more clarification, the pseudocode for the BHC strategy is presented in Algorithm 2. Similar to the flowchart of this strategy, there are three important phases. In the first phase (so-called *initialization*: lines 1-2), the coverage matrix (*CM*) is constructed, and the variable *RemCov* is set to the value of $\binom{p}{t} \times d^t$. Additionally, the empty test suite (*TS*) is created. In the second phase (so-called *running*: lines 4-21), the BHC algorithm is run, and the resulting test case is stored in *tc*. In the third phase (so-called *updating*: line 22), the weight of *tc* is calculated using the *calcWeight* fitness function and subtracted from *RemCov*. Furthermore, the test case *tc* is appended to the test suite *TS*, and the coverage matrix *CM* is updated accordingly.

Overall, this explanation clarifies the approach of the BHC strategy for generating an optimal test set, emphasizing its iterative nature and utilization of the coverage matrix to monitor different combination coverage.

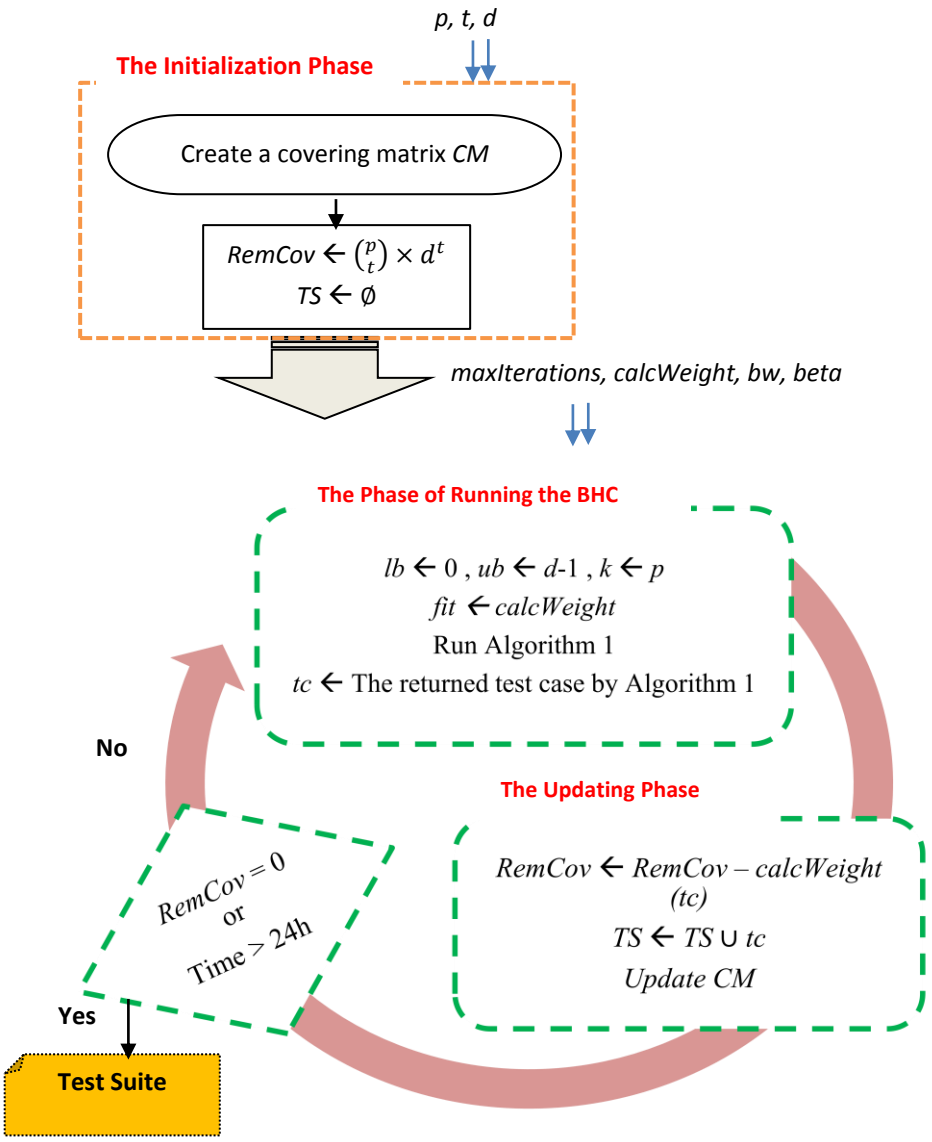

Fig. 1: Flowchart of the BHC strategy.

F. Create Coverage Matrix

As the BHC strategy generates test samples incrementally using the BHC algorithm, a data structure is essential to track the coverage status of all potential combinations of the *t* parameters at each step. This data structure is known as the "coverage matrix," and it comprises $\binom{p}{t}$ rows and d^t columns. Here's how it works:

- The variable *p* signifies the number of input parameters in the system, each capable of taking on *d* different values. Consequently, the total number of combinations for such a system equals $\binom{p}{t} \times d^t$.
- The coverage matrix needs to hold information for each row, including d^t columns to represent the combinations and *t* columns to keep track of the indices of combinations corresponding to that row.
- An additional column (the last column) in each row indicates the number of combinations from that row that are still uncovered by any test sample. Therefore, the total number of columns in the coverage matrix becomes $t+d^t+1$.

Initially, when the test set is empty and no combinations are covered, the values in all d^t columns for each row are set to zero.

The values in the last column for all rows are set to d^t indicating that all combinations are yet to be covered by any test sample.

In summary, the coverage matrix is a critical data structure employed in the BHC strategy to keep tabs on the coverage status of all possible *t*-parameter combinations during the algorithm's execution. It efficiently tracks coverage progress and plays a key role in generating an optimal test set.

Algorithm 2: The BHC strategy

Input: p: the number of parameters, d: the number of parameter values, t: the interaction strength, *maxIterations*: the maximum number of iterations, *bw, beta*;

Output: a test suite;

////// The initialization phase //////

1: Create a coverage matrix *CM*;

2: *RemCov* = $\binom{p}{t} \times d^t$; *TS* = ∅;

3: **while** *RemCov*>0 && *ConsumedTime* < 24h **do**

////// The running phase //////

4: *lb* = 0 ; *ub* = *d*-1; *k* = *p*; *fit* = *calcWeight* ();

5: Individual *x*=new Individual ();

6: *x* = Initialize (*k*, *lb*, *ub*); *rep* =1;

7: **while** *rep* **<=** *maxIterations* **do**

8: Individual *x'* = *x*;

//// The $\mathcal{N}$ operator: This operator generates new candidate solutions by adding normally distributed noise, aiding in local exploitation around the current solution.

9: index = a random number between 1 and *k*;

10: rnd = a random number between 0 and 1;

11: *x'* [index] = *x* [index] +(2*rnd-1)**bw*;

//// The β operator

//// The β operator: This operator diversifies the search by using controlled random perturbations to escape local optima and explore new regions of the solution space.

//////

12: **for** j = 1 **to** *k* **do**

13: **if** rand(0,1) <=*beta* **then**

14: *x'* [j] = *lb*[j]+(*ub*[j]-*lb*[j])* rand(0,1);

15: **end if**

16: **end for**

17: **if** *fit* (*x'*) > *fit* (*x*) **then**

18: *x* = *x'*;

19: **end if**

20: *rep* ++;

21: **end while**

////// The updating phase //////

22: TS = *TS* ∪ *x*; *RemCov* = *RemCov* – *fit* (*x*); Update *CM*;

23: **end while**

24: **return** *TS;*

To provide a clearer understanding of the coverage matrix, let's consider a hypothetical system with p = 4 input parameters and d = 3 possible values for each parameter. Our objective is to create the coverage matrix for the covering array *CA* (N; 2, 3^4). Here's how we determine the matrix dimensions:

The number of rows, denoted as $\binom{p}{t}$, is calculated as $\binom{4}{2}$ which equals 6.

The total number of columns, represented as $t+d^t+1$ is determined as $2+3^2+1$, summing up to 12 columns.

Now, let's break down the columns of this matrix based on Fig. 2:

The first two columns (0 and 1) are responsible for indicating the indices of the 6 different combinations (12, 13, 14, 23, 24, 34).

The following nine columns (2 to 10) encompass all possible values corresponding to the two parameters (00, 01, 02, 10, 11, 12, 20, 21, 22).

The last column (11) is dedicated to keeping track of the number of elements that remain uncovered.

To calculate the column number (c) for a given combination of indices $i_0 i_1 \ldots i_{t-1}$, we can utilize (2):

$$c = t + \sum_{j=0}^{t-1} (i_j \times d^{t-1-j}) \tag{2}$$

In sum, the coverage matrix is a structured representation used to track combinations of parameters in the context of covering arrays. In our hypothetical system, it enables us to efficiently monitor the coverage status of various combinations and their corresponding values, aiding in the quest to generate an optimal test set.

For instance, in the hypothetical system we're examining, consider the combination of indices (i_0 = 2 and i_1 = 1). To determine the corresponding column number (c), you can use the following calculation: c = 2 + $(2\times3^1+1\times3^0)$ = 9.

Furthermore, the initial value of *RemCov* (representing the count of elements that have not been covered yet) equals $\binom{p}{t}\times d^t$, which in our scenario is $\binom{4}{2}\times 3^2$, resulting in 54, as illustrated in Fig. 2.

1	**2**	0	0	0	0	0	0	0	0	0	**9**
1	**3**	0	0	0	0	0	0	0	0	0	**9**
1	**4**	0	0	0	0	0	0	0	0	0	**9**
2	**3**	0	0	0	0	0	0	0	0	0	**9**
2	**4**	0	0	0	0	0	0	0	0	0	**9**
3	**4**	0	0	0	0	0	0	0	0	0	**9**
0	1	2	3	4	5	6	7	8	9	10	11

$$\boldsymbol{RemCov = 9 + 9 + \ldots + 9 = 54}$$

Fig. 2: Structure of coverage matrix for *CA* (N; 2,3^4) configuration.

G. Calculating Test Sample Weight and Updating the Coverage Matrix

To determine the weight of a given test sample, we count the number of new combinations it covers. Here's the process:

- We consider both the values within the test sample itself and those in the first *t* columns of each row in the coverage matrix.
- First, we calculate the column number for each combination (row) using (2) based on the test sample's values and the *t* columns.
- If the value in the corresponding cell of the coverage matrix is one, it means that this combination has already been covered by previous test samples and added to the test set. In such cases, there's no need to update the weight of the current test sample.
- However, if the value is zero, we add one to the weight of the current test sample, and we set the value of that cell to one, indicating that this combination should be excluded when calculating the weight of the next test sample.
- It's important to subtract one from the value in the last column of the specific row to account for the updated weight of the current test sample.
- Finally, after examining all rows, we subtract the obtained weight for this test sample from the *RemCov* value. *RemCov* signifies the count of remaining combinations that have yet to be covered.

To illustrate this process, let's consider the test sample *tc* = (0, 1, 2, 0) in Fig. 2. In Fig. 3, we visually depict how the weight is calculated for this test sample based on the coverage matrix. As shown, we first calculate the column number for each combination (row) using (2) and update all corresponding cells from zero to one. Additionally, the values in the last columns, initially at 9, are reduced to 8. Finally, after all the updates, the *RemCov* value decreases to 48. This process ensures efficient tracking of covered combinations and contributes to generating an optimal test set.

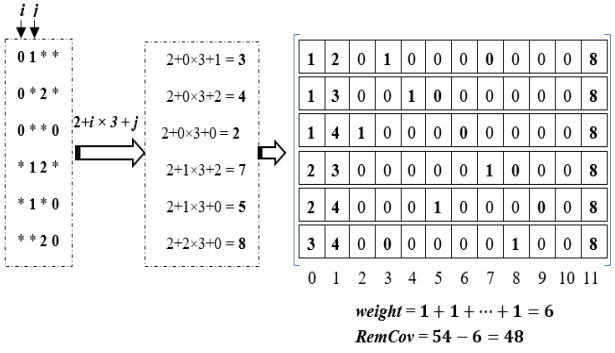

Fig. 3: Calculating the weight of the test sample *tc* = (0,1,2,0) according to the coverage matrix of Figure 2 and updating it.

Evaluation Results

In order to assess the effectiveness of the BHC strategy, we conducted an evaluation using MATLAB 2017a software. Our analysis involved comparing the BHC strategy against various other strategies, including TConfig as a mathematical approach, PICT and IPOG as greedy methods, and GS, GALP, DPSO, WOA, BAPSO, and GSTG as metaheuristic strategies.

Before commencing the evaluation and comparison, we needed to determine appropriate values for two crucial parameters: *bw*, which is related to the *N* operator, and *β*, which relates to the *β* operator. To achieve this, we executed the BHC strategy ten times within the context of the *CA* (*N*; 2, *p*, 4) configuration, where *p* ranged from 11 to 16. Our analysis of the results, as depicted in Fig. 4, revealed that the BHC strategy exhibited its best performance when configured with *bw = 0.5* and *β = 0.2*. Additionally, we acquired suitable parameter values for the other strategies from relevant literature, and these values are summarized in Table 4.

Another essential parameter in our assessment is the maximum number of function evaluations (*maxFFE*), which we set to 10,000 for all strategies under consideration. It's worth noting that the parameters *maxFFE* and the number of iterations (*MaxIterations*) in the algorithms are interchangeable. If we denote *nFFE* as the number of function evaluations performed in each iteration of an algorithm, then the number of iterations can be calculated using (3):

$$MaxIterations = \frac{\text{maxFFE}}{nFFE} \tag{3}$$

As an example, consider the WOA algorithm within our chosen strategy, where the population size is set to 180. Consequently, the value of *nFFE* corresponds to 180. Using (3), we can calculate the number of iterations, which yields *MaxIterations* equal to 10,000 / 180, resulting in approximately 56 iterations.

H. Results Generation

To obtain our results, we executed all the strategies a total of 100 times across five distinct datasets, as follows:

- *CA* (*N*; 2, *p*, 3) for $3 \le p \le 12$
- *CA* (*N*; 3, *p*, *d*) for $7 \le p \le 12$ and $2 \le d \le 3$
- *CA* (*N*; *t*, *p*, 3) for $7 \le t \le 11$ and $t+1 \le p \le 12$
- *CA* (*N*; 4, *p*, 5) for $5 \le p \le 15$
- *CA* (*N*; *t*, *p*, 2) for $12 \le t \le 16$ and $14 \le p \le 17$

The hardware environment for running these strategies was equipped with an Intel® Core™ i5 CPU and 6GB of RAM. It's important to note that in our results table, we use "NA" and ">day" when a specific strategy execution result is unavailable for a given configuration or when a strategy failed to produce an optimal covering array within a 24-hour timeframe.

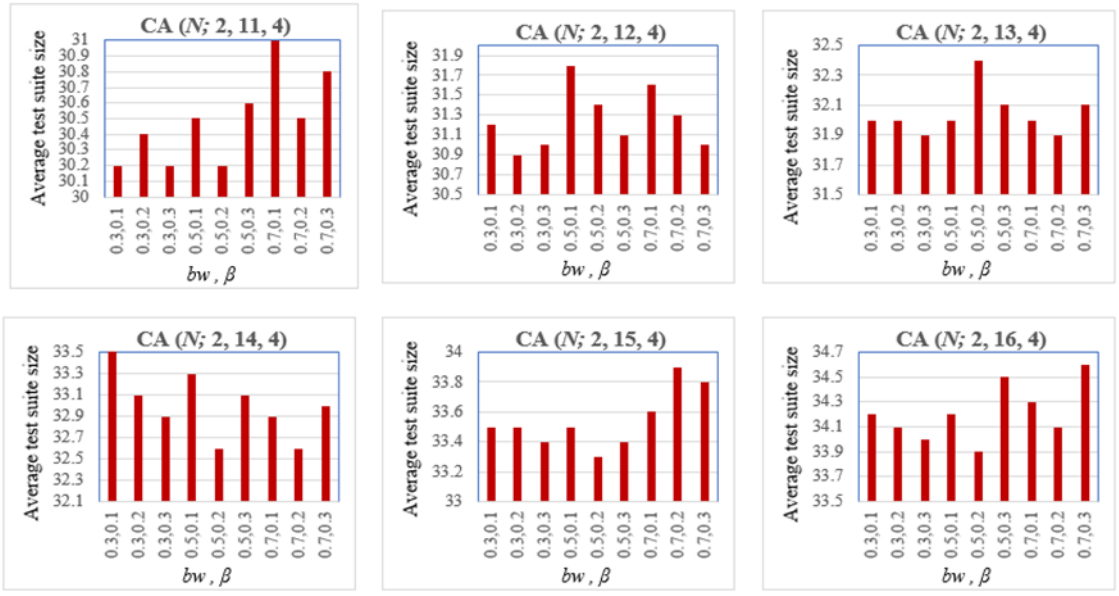

Fig. 4: Average size of production test set in *CA* (*N*; 2,*p*,4) configuration for 11≤ p≤16.

Table 4: Appropriate values for parameters of the meta-heuristic strategies

Strategy	Parameters and their appropriate values
BHC	*bw* = 0.5, *beta* = 0.2
GS	*N* = 120, *MutRate* = 0.4, *CossRate* = 0.4
GALP	*N* = 10-300, *MutRate* = 0.4..1.0, *CossRate* = 0.4..1.0 *Mutation method* : Uniform *Crossover method :* ALPSOFV (*Kmax* = 10-20, *a* = 0.9, *r* = 4, *R1* and *R2*: random)
WOA	*N* = 180
BAPSO	*N* = 100, *Loudness* = 0.9, *PulseemissionRate* = 0.9, *Minfrequency* = 0, *Maxfrequency* =1, *Tolerance* = 0.025, *WeightValue* = 0.4, *CycWalkvalue* = 1.49
GSTG	*N* = 100, *ElitistCheck* = 1, *Rpower* = 1, *Rnorm* = 2

In Table 5, we present the sizes of the test sets generated by all strategies for the first dataset, *CA* (*N*; 2, *p*, 3), where 3 ≤ *p* ≤ 12. This dataset encompasses 10 distinct configurations. Notably, in 9 out of these cases, both GALP and GS produced the smallest test sets. Following closely, BHC generated the smallest test sets in 8 of these configurations. In contrast, TConfig, employed as a mathematical strategy, failed to produce any test set with the smallest size for this dataset. Among the considered greedy strategies, IPOG consistently produced the smallest test sets in the majority of cases.

In Table 6, we present the sizes of the test sets generated by all strategies for the second dataset, *CA* (*N*; 3, *p*, *d*), where 7 ≤ *p* ≤ 12 and 2 ≤ *d* ≤ 3. This dataset encompasses 13 distinct configurations. Notably, in terms of generating test sets with the smallest size, both GALP and BHC demonstrated the best performance across this dataset.

In Table 7, we provide the sizes of the test sets generated by all strategies for the third dataset, denoted as *CA* (*N*; *t*, *p*, 3)**,** where 7 ≤ *t* ≤ 11 and *t* + 1 ≤ *p* ≤ 12**.** This dataset encompasses a total of 15 distinct configurations. Notably, in 8 of these configurations, the BHC strategy stands out by producing test sets with the smallest size.

Tables 8 and 9 present the sizes of the test sets generated by all strategies for the fourth and fifth datasets, respectively. The fourth dataset corresponds to

CA (*N*; 4, *p*, 5), where 5 ≤ *p* ≤ 15, while the fifth dataset pertains to *CA* (*N*; *t*, *p*, 2), where 12 ≤ *t* ≤ 16 and 14 ≤ *p* ≤ 17.

Upon reviewing these tables, it becomes evident that BHC consistently excels by producing test sets with the smallest size across the majority of configurations in these datasets. To demonstrate the superiority of the BHC strategy over other existing methods, we utilized the Friedman test. This non-parametric statistical test is designed for comparing multiple related samples [66] and is effective in identifying performance differences among the various strategies. In the Friedman test output report, the strategy with the lowest average rank is designated as Rank 1, signifying the best performance. Subsequently, the second lowest rank is assigned as Rank 2, and so forth.

Table 10 provides an overview of the overall ranks achieved by both BHC and other strategies across all the considered datasets, totaling 54 distinct configurations. Notably, BHC consistently secures the first rank, demonstrating its effectiveness in generating test sets with the smallest size.

Table 5: Test set size generated by all strategies in $CA\ (N;\ 2, p, 3)$ for 3 ≤ *p* ≤ 12

d, p	BHC N.Best	BHC N.Avg	GALP N.Best	GSTG N.Best	GS N.Best	BAPSO N.Best	DPSO N.Best	WOA N.Best	TConfig N.Best	PICT N.Best	IPOG N.Best
3, 3	**9**	9.69	**9**	**9**	**9**	**9**	**9**	**9**	10	10	**9**
3, 4	**9**	10.28	**9**	**9**	**9**	**10**	**9**	**9**	10	13	**9**
3, 5	**11**	12.88	**11**	**11**	**11**	**11**	**11**	**11**	14	13	15
3, 6	**13**	14.55	**13**	**13**	**13**	**14**	14	14	15	14	15
3, 7	**14**	15.24	**14**	15	**14**	15	15	**14**	15	16	15
3, 8	**15**	15.92	**15**	**15**	**15**	**15**	**15**	**15**	17	16	**15**
3, 9	**15**	16.55	**15**	17	**15**	16	**15**	16	17	17	**15**
3, 10	**16**	17.43	16	18	16	17	16	16	17	18	**15**
3, 11	17	17.82	**16**	19	**16**	17	17	17	20	18	17
3, 12	17	18.23	**16**	19	**16**	17	**16**	17	20	19	21

Table 6: Size of the test set generated by all strategies in *CA* (*N*; 3, *p*, *d*) for 7 ≤ *p* ≤12 and 2 ≤ *d* ≤3

d, p	BHC N.Best	BHC N.Avg	GALP N.Best	GSTG N.Best	GS N.Best	BAPSO N.Best	DPSO N.Best	WOA N.Best	TConfig N.Best	PICT N.Best	IPOG N.Best
2, 7	**12**	15.09	**12**	**12**	**12**	12	15	12	16	15	16
2, 8	**12**	16.05	**12**	14	14	14	16	14	18	17	18
2, 9	**16**	16.25	16	16	16	**15**	16	**15**	20	17	20
2, 10	**16**	16.65	16	16	16	16	16	16	20	18	20
3, 4	**27**	31.12	**27**	28	**27**	**27**	28	**27**	32	34	32
3, 5	38	40.39	**37**	40	38	38	41	38	40	43	41
3, 6	39	45.10	40	43	43	43	**33**	42	48	48	46
3, 7	**48**	50.66	**48**	**48**	49	**48**	**48**	**48**	55	51	55
3, 8	**52**	54.75	**52**	53	54	**52**	**52**	53	58	59	56
3, 9	**56**	58.51	**56**	**56**	58	57	**56**	57	64	63	63
3, 10	60	61.78	**59**	61	61	59	**59**	**59**	68	65	66
3, 11	63	64.81	**62**	63	63	63	63	64	72	70	70
3, 12	**65**	67.51	**65**	**65**	67	**65**	**65**	**65**	77	72	73

Table 7: Size of the test set generated by all strategies in *CA* (*N*; *t*, *p*, 3) for 7 ≤ *t* ≤11 and *t*+1 ≤ *p* ≤12

t , *p*	BHC N.Best	BHC N.Avg	GALP N.Best	GSTG N.Best	GS N.Best	BAPSO N.Best	DPSO N.Best	WOA N.Best	TConfig N.Best	PICT N.Best	IPOG N.Best
7, 8	2989	3013.12	3029	3031	3032	3029	**2241**	3031	>day	3143	NA
7, 9	**4417**	4435.23	4425	4425	4437	4425	4427	4425	> day	4618	NA
7, 10	5476	5484.32	5459	5473	5464	5472	**5434**	5474	>day	5884	NA
7, 11	6427	6437.13	6468	6543	6533	6515	**6413**	6531	>day	7116	NA
7, 12	**7557**	7563.12	7593	7598	7603	7614	>day	7610	>day	8314	NA
8, 9	9283	9292.25	**9258**	9372	9266	9273	9284	**9258**	>day	9763	NA
8, 10	**13894**	13902.42	13903	13912	13907	13912	13939	13912	> day	14599	NA
8,11	17713	17832.31	**17639**	17753	17792	17831	>day	17641	>day	18,859	NA
8, 12	**20415**	20513.21	20963	22731	21670	21631	>day	21310	>day	23112	NA
9, 10	**28434**	28445.31	28312	29631	28629	27813	**21433**	28319	>day	30181	NA
9, 11	**43474**	43495.31	43543	43591	43809	43931	>day	43592	> day	45521	NA
9, 12	**53813**	53865.21	56219	57301	56481	57931	>day	55931	>day	59966	NA
10, 11	**86960**	87031.42	87712	88391	87766	88351	>day	87721	>day	92435	NA
10, 12	**133842**	13394.2	135962	139041	136096	140315	>day	136841	> day	141990	NA
11, 12	**246309**	257342.4	267085	26518	267131	274102	>day	259341	>day	278993	NA

Table 8: Size of the test set generated by all strategies in *CA* (*N*; 4, *p*, 5) for 5 ≤ *p* ≤15

p	BHC N.Best	BHC N.Avg	GALP N.Best	GSTG N.Best	GS N.Best	BAPSO N.Best	WOA N.Best	TConfig N.Best	PICT N.Best	IPOG N.Best
5	775	782.65	**762**	774	769	774	784	773	810	773
6	**984**	992.32	986	1021	984	1031	1015	1092	1072	1058
7	**1168**	1174.31	**1168**	1172	1176	1181	**1168**	1320	1279	1293
8	**1338**	1341.45	1364	1353	1371	1384	1379	1532	1468	1511
9	**1488**	1494.50	1553	1587	1548	1531	1592	1724	1643	1702
10	**1634**	1635.21	**1634**	1747	1638	**1634**	1641	1878	1812	1869
11	**1766**	1767.75	1812	1769	1838	1853	1769	2038	1957	2024
12	**1890**	1893.50	1910	1923	1967	1983	1941	2178	2103	2150
13	**2003**	2005.25	2013	2104	2041	2031	2112	>day	2238	2296
14	**2115**	2117.75	2131	2251	2156	2293	2123	>day	2359	2436
15	**2217**	2220.17	2245	2371	2268	2258	2259	>day	2480	2538

Table 9: Size of the test set generated by all strategies in *CA* (*N*; *t*, *p*, 2) for 12 ≤ *t* ≤16 and 14 ≤ *p* ≤17

	BHC N.Best	BHC N.Avg	GALP N.Best	GSTG N.Best	GS N.Best	BAPSO N.Best	DPSO N.Best	WOA N.Best	TConfig N.Best	PICT N.Best	IPOG N.Best
***CA* (*N*; 12, 2^{14})**	**8873**	8891.32	8904	9021	8893	8891	8972	8874	> day	9112	NA
***CA* (*N*; 13, 2^{14})**	11152	11210.63	11051	11714	**10251**	11731	>day	11217	> day	12441	NA
***CA* (*N*; 14, 2^{15})**	22163	22231.33	22642	21983	23377	**21725**	>day	22862	> day	25036	NA
***CA* (*N*; 15, 2^{16})**	**40415**	40512.75	41820	42930	46575	43624	>day	42641	> day	51127	NA
***CA* (*N*; 16, 2^{17})**	**94431**	94451.63	94932	94941	95680	96320	>day	95326	> day	100266	NA

Table 10: Overall ranks of BHC and other strategies in all considered data sets

		BHC	GALP	GSTG	GS	BAPSO	DPSO	WOA	TConfig	PICT	IPOG
Friedman Test	Mean Rank	2.69	2.75	4.93	4.35	4.63	6.50	4.09	9.03	7.91	8.13
	Rank	**1**	**2**	**6**	**4**	**5**	**7**	**3**	**10**	**8**	**9**

Conclusion and Future Directions

Generating an optimal test set, also known as a coverage array, is a complex NP-Hard problem. Metaheuristic algorithms such as Genetic Algorithms, Particle Swarm Optimization, Ant Colony Optimization, and Tabu Search have demonstrated their remarkable efficiency in tackling this problem. However, the issue of handling large test set sizes remains unresolved. In this paper, we introduced an innovative strategy based on the Beta Hill Climbing (BHC) optimization method to address the optimal test set generation problem. In our evaluation and comparison, we pitted BHC against TConfig (a mathematical strategy), PICT and IPOG (greedy strategies), and GS, GALP, DPSO, WOA, BAPSO, and GSTG (metaheuristic strategies) using five well-established datasets. According to the results derived from the Friedman test, BHC consistently ranked first in generating test sets with the smallest size.

The primary contributions of this research include the first-time application of the β-hill climbing (BHC) optimizer for generating minimum covering arrays, which demonstrates greater efficiency and supports higher interaction strengths (t>15) than most existing strategies.

However, BHC has certain limitations, such as its inability to handle variable-strength interactions and constraints, which are essential features of t-way testing. To address these gaps, future work can focus on extending BHC to incorporate these features. Additionally, integrating BHC with established metaheuristic algorithms can further improve its efficiency and expand its utility.

Author Contributions

All authors of the paper developed the algorithm algorithm, executed the experiments, analyzed the results, and authored the manuscript.

Funding

This research received no external funding.

Acknowledgment

We would like to express our sincere gratitude to the editor and anonymous reviewers for their valuable time, insightful feedback, and constructive comments, which greatly contributed to the improvement and quality of this paper.

Conflict of Interest

The authors declare no potential conflict of interest regarding the publication of this work. In addition, the ethical issues including plagiarism, informed consent, misconduct, data fabrication and, or falsification, double publication and, or submission, and redundancy have been completely witnessed by the authors.

Abbreviations

ABC	Artificial Bee Colony
ABCVS	Artificial Bee Colony for generating Variable t-way Test Sets
AETG	Automatic Efficient Test Generator
ALPSOFV	Adaptive Layered Population Size with Optimal Fitness Value
BAPSO	Hybrid of Bat Algorithm and Particle Swarm Optimization
BDA	Bi-Objective Dragonfly Algorithm
BHC	Beta Hill Climbing
β-HCO	Beta Hill Climbing Optimizer
BOA	Butterfly Optimization Algorithm
CA	Covering Array
CM	Coverage Matrix
CTEXL	Classification-Tree Editor eXtended Logics
DPSO	Discrete Particle Swarm Optimization
GALP	Genetic Algorithm with Local Path
GA	Genetic Algorithm
GS	Genetic Strategy
GSTG	Gravitational Search Test Generator
HABC	Hybrid Artificial Bee Colony
HC-BAT	Hybrid Hill Climbing and Bat Algorithm
HGHC	Hybrid Greedy Hill Climbing
HHOA	Harris Hawks Optimization Algorithm

IPOG	In-Parameter-Order General
LSHADE	Linear Success-History Adaptive Differential Evolution
MBH	Multiple Black Hole
OPT	One-Parameter-at-a-Time
ORT	One-Row-at-a-Time
PICT	Pairwise Independent Combinatorial Testing
PSO	Particle Swarm Optimization
QWOA-EMC	Q-learning Whale Optimization Algorithm with Ensemble Model Checking
REMCOV	Remaining Combinations
ROBDDs	Reduced Ordered Binary Decision Diagrams
SCA	Sine Cosine Algorithm
SCAVS	Sine Cosine Algorithm for Variable t-way Test Suite
SCIPOG	Seeding and Constraint Support in IPOG
TC	Test Case
TLBO	Teaching–Learning-Based Optimization
TP	Test Parameter
TS	Test Set
TPA	Three-Phase Approach
TWAY	t-way Combinatorial Testing
WOA	Whale Optimization Algorithm

References

[1] B. S. Ahmed, T. S. Abdulsamad, M. Y. Potrus, "Achievement of minimized combinatorial test suite for configuration-aware software functional testing using the cuckoo search algorithm," Inf. Software Technol., 66: 13-29, 2015.

[2] R. Jhala, R. Majumdar, "Software model checking," ACM Comput. Surv. (CSUR), 41(4): 1-54, 2009.

[3] L. Luo, "Software testing techniques," Institute for software research international Carnegie mellon university Pittsburgh, PA, vol. 15232, no. 1-19, p. 19, 2001.

[4] D. R. Kuhn, M. J. Reilly, "An investigation of the applicability of design of experiments to software testing," in Proc. 27th Annual NASA Goddard/IEEE Software Engineering Workshop: 91-95, 2002.

[5] M. A. Al-Betar, "β-hill climbing: an exploratory local search," Neural Comput. Appl., 28(Suppl 1): 153-168, 2017.

[6] B. Selman, C. P. Gomes, "Hill-climbing search," Encycl. Cognit. Sci., 81, 2006.

[7] S. Chinnasamy, M. Ramachandran, M. Amudha, K. Ramu, "A review on hill climbing optimization methodology," Recent Trends Manage. Commerce, 3(1), 2022.

[8] E. R. R. Kato, G. D. de Aguiar Aranha, R. H. Tsunaki, "A new approach to solve the flexible job shop problem based on a hybrid particle swarm optimization and Random-Restart Hill Climbing," Comput. Ind. Eng., 125: 178-189, 2018.

[9] E. Aarts, J. Korst, W. Michiels, "Simulated annealing," in Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques: 187-210, 2005.

[10] J. Arriaga, M. Valenzuela-Rendón, "Steepest ascent hill climbing for portfolio selection," in Proc. European Conference on the Applications of Evolutionary Computation: 145-154, 2012.

[11] D. LaSalle, G. Karypis, "A parallel hill-climbing refinement algorithm for graph partitioning," in Proc. 45th International Conference on Parallel Processing (ICPP): 236-241, 2016.

[12] M. Gendreau, J. Y. Potvin, "Tabu search," in Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques: 165-186, 2005.

[13] M. H. Shirvani, "A hybrid meta-heuristic algorithm for scientific workflow scheduling in heterogeneous distributed computing systems," Eng. Appl. Artif. Intell., 90: 103501, 2020.

[14] B. Keshanchi, N. J. Navimipour, "Priority-based task scheduling in the cloud systems using a memetic algorithm," J. Circuits Syst. Comput., 25(10): 1650119, 2016.

[15] B. Selman, C. P. Gomes, "Hill-climbing search," Encycl. Cognit. Sci., 81(10): 333-335, 2006.

[16] S. Esfandyari, V. Rafe, "A tuned version of genetic algorithm for efficient test suite generation in interactive t-way testing strategy," Inf. Software Technol., 94: 165-185, 2018.

[17] A. Hartman, "Software and hardware testing using combinatorial covering suites," in Graph theory, combinatorics and algorithms: Springer, pp: 237-266, 2005.

[18] A. W. Williams, R. L. Probert, "A practical strategy for testing pair-wise coverage of network interfaces," in Proc. 7th International Symposium on Software Reliability Engineering (ISSRE'96) :246-254, 1996.

[19] C. Nie, H. Leung, "A survey of combinatorial testing," ACM Comput. Surv. (CSUR), 43(2): 1-29, 2011.

[20] D. M. Cohen, S. R. Dalal, M. L. Fredman, G. C. Patton, "The AETG system: An approach to testing based on combinatorial design," IEEE Trans. Software Eng., 23(7): 437-444, 1997.

[21] M. B. Cohen, "Designing test suites for software interactions testing," Auckland Univ (New Zealand), 2004.

[22] J. Czerwonka, "Pairwise testing in the real world: Practical extensions to test-case scenarios," Microsoft Corporation, Software Testing Technical Articles, 2008.

[23] B. Jenkins, "Jenny test tool," ed, 2009.

[24] B. S. Ahmed, M. A. Sahib, M. Y. Potrus, "Generating combinatorial test cases using Simplified Swarm Optimization (SSO) algorithm for automated GUI functional testing," Eng. Sci. Technol., Int. J., 17(4): 218-226, 2014.

[25] K. Z. Zamli, M. F. Klaib, M. I. Younis, N. A. M. Isa, R. Abdullah, "Design and implementation of a t-way test data generation strategy with automated execution tool support," Inf. Sci., 181(9): 1741-1758, 2011.

[26] M. Forbes, J. Lawrence, Y. Lei, R. N. Kacker, D. R. Kuhn, "Refining the in-parameter-order strategy for constructing covering arrays," J. Res. Natl. Inst. Stand. Technol., 113(5): 287, 2008.

[27] A. A. Muazu, A. S. Hashim, A. Sarlan, M. Abdullahi, "SCIPOG: Seeding and constraint support in IPOG strategy for combinatorial t-way testing to generate optimum test cases," J. King Saud Univ. Comput. Inf. Sci., 35(1): 185-201, 2023.

[28] S. Akyol, B. Alatas, "Plant intelligence based metaheuristic optimization algorithms," Artif. Intell. Rev., 47: 417-462, 2017.

[29] B. Alatas, "ACROA: Artificial chemical reaction optimization algorithm for global optimization," Expert Syst. Appl., 38(10): 13170-13180, 2011.

[30] Z. Abbasi, S. Esfandyari, and V. Rafe, "Covering array generation using teaching learning base optimization algorithm," Tabriz J. Electr. Eng., 48(1): 161-171, 2018.

[31] J. Kennedy, R. Eberhart, "Particle swarm optimization," in Proc. ICNN'95-International Conference on Neural Networks, 4: 1942-1948, 1995.

[32] S. Esfandyari, V. Rafe, "Using the particle swarm optimization algorithm to generate the minimum test suite in covering array with uniform strength," Soft Comput. J., 8(2): 66-79, 2021.

[33] Y. A. Alsariera, A. H. Al Omari, M. A. Albawaleez, Y. K. Sanjalawe, K. Z. Zamli, "Hybridized BA & PSO t-way Algorithm for Test Case Generation," Int. J. Comput. Sci. Network Secur. (IJCSNS), 21(10): 343, 2021.

[34] K. M. Htay, R. R. Othman, A. Amir, J. M. H. Alkanaani, "Gravitational search algorithm based strategy for combinatorial t-way test suite generation," J. King Saud Univ. Comput. Inf. Sci., 34(8A): 4860- 4873, 2022.

[35] D. Whitley, "A genetic algorithm tutorial," Stat. Comput., 4(2): 65-85, 1994.

[36] H. N. N. Al-Sammarraie, D. N. Jawawi, "Multiple black hole inspired meta-heuristic searching optimization for combinatorial testing," IEEE Access, 8: 33406-33418, 2020.

[37] J. M. Altmemi, R. Othman, R. Ahmad, "SCAVS: Implement Sine Cosine Algorithm for generating Variable t-way test suite," in Proc. IOP Conference Series: Materials Science and Engineering, 917(1): 012011, 2020.

[38] S. Mirjalili, "SCA: A sine cosine algorithm for solving optimization problems," Knowledge-Based Syst., 96: 120-133, 2016.

[39] A. K. Alazzawi, H. M. Rais, S. Basri, "Artificial bee colony algorithm for t-way test suite generation," in Proc. 2018 4th International Conference on Computer and Information Sciences (ICCOINS): 1-6, 2018.

[40] A. K. Alazzawi, H. M. Rais, S. Basri, "ABCVS: An artificial bee colony for generating variable t-way test sets," Int. J. Adv. Comput. Sci. Appl., 10(4): 259-274, 2019.

[41] A. K. Alazzawi, H. M. Rais, S. Basri, Y. A. Alsariera, "Pairwise test suite generation based on hybrid artificial bee colony algorithm," in Advances in Electronics Engineering, Springer, pp. 137-145, 2020.

[42] A. K. Alazzawi, H. Rais, S. Basri, "HABC: Hybrid artificial bee colony for generating variable t-way test sets," J. Eng. Sci. Technol., 15(2): 746-767, 2020.

[43] A. K. Alazzawi, H. M. Rais, S. Basri, "Parameters tuning of hybrid artificial bee colony search based strategy for t-way testing," Int. J. Innov. Technol. Exploring Eng, 8(5S): 204-212, 2019.

[44] A. K. Alazzawi, H. M. Rais, S. Basri, "Hybrid artificial bee colony algorithm for t-way interaction test suite generation," in Proc. Computer Science On-line Conference: 192-199, 2019.

[45] A. K. Alazzawi, H. M. Rais, S. Basri, Y. A. Alsariera, A. O. Balogun, A. A. Imam, "A hybrid artificial bee colony strategy for t-way test set generation with constraints support," J. Phys. Conf. Ser., 1529(4): 042068, 2020.

[46] A. K. Alazzawi, H. M. Rais, S. Basri, Y. A. Alsariera, "PhABC: A hybrid artificial bee colony strategy for pairwise test suite generation with constraints support," in Proc. 2019 IEEE Student Conference on Research and Development (SCOReD): 106-111, 2019.

[47] S. Esfandyari and V. Rafe, "A hybrid solution for software testing to minimum test suite generation using hill climbing and bat search algorithms," Tabriz J. Electr. Eng., 46(3): 25-35, 2016.

[48] J. B. Odili, A. B. Nasser, A. Noraziah, M. H. A. Wahab, M. Ahmed, "African buffalo optimization algorithm based t-way test suite generation strategy for electronic-payment transactions," in Proc. International Conference on Emerging Technologies and Intelligent Systems (ICETIS 2021): 160-174, 2022.

[49] M. Ahmed, A. B. Nasser, K. Z. Zamli, "Construction of prioritized t-way test suite using bi-objective dragonfly algorithm," IEEE Access, 10: 71683-71698, 2022.

[50] E. Pira, V. Rafe, S. Esfandyari, "Minimum covering array generation using success-history and linear population size reduction based adaptive differential evolution algorithm," Tabriz J. Electr. Eng., 52(2): 77-89, 2022.

[51] S. Esfandyari, V. Rafe, "GALP: A hybrid artificial intelligence algorithm for generating covering array," Soft Comput., 25(11): 7673-7689, 2021.

[52] M. Z. Z. Ahmad, R. R. Othman, M. S. A. R. Ali, N. Ramli, "A self-adapting Ant Colony Optimization Algorithm Using Fuzzy logic (ACOF) for combinatorial test suite generation," in Proc. IOP Conference Series: Materials Science and Engineering, 767(1): 012017, 2020.

[53] A. A. Alsewari, L. M. Xuan, K. Z. Zamli, "Firefly combinatorial testing strategy," in Proc. Science and Information Conference: 936-944, 2018.

[54] K. Z. Zamli, F. Din, A. B. Nasser, A. Alsewari, "Combinatorial test suite generation strategy using enhanced sine cosine algorithm," in Proc. 5th International Conference on Electrical, Control & Computer Engineering: 127-137, 2020.

[55] A. S. Ghiduk, A. Alharbi, "Generating of test data by harmony search against genetic algorithms," Intell. Autom. Soft Comput., 36(1): 647, 2023.

[56] A. A. Muazu, A. S. Hashim, U. I. i. Audi, U. D. Maiwada, "Refining a one-parameter-at-a-time approach using harmony search for optimizing test suite size in combinatorial t-way testing," IEEE Access, 12: 137373-137398, 2024.

[57] H. M. Fadhil, M. Abdullah, M. Younis, "Innovations in T-way test creation based on a hybrid hill climbing-greedy algorithm," IAES Int. J. Artif. Intell., 12(2): 794, 2023.

[58] A. A. Hassan, S. Abdullah, K. Z. Zamli, R. Razali, "Q-learning whale optimization algorithm for test suite generation with constraints support," Neural Comput. Appl., 35(34): 24069-24090, 2023.

[59] X. Guo, X. Song, J. t. Zhou, F. Wang, "A tolerance-based memetic algorithm for constrained covering array generation," Memet. Comput., 15(3): 319-340, 2023.

[60] B. Arasteh, K. Arasteh, A. Ghaffari, "An automatic software test-generation method to discover the faults using fusion of machine learning and horse herd algorithm," J. Supercomput., 81(5): 1-36, 2025.

[61] B. Arasteh et al., "A bioinspired test generation method using discretized and modified bat optimization algorithm," Mathematics, 12(2): 186, 2024.

[62] H. M. Fadhil, M. N. Abdullah, M. I. Younis, "Dynamic TWGH: Client-server optimization for scalable combinatorial test suite generation," in Proc. BIO Web of Conferences: 00115, 2024.

[63] E. Pira, V. Rafe, S. Esfandyari, "A three-phase approach to improve the functionality of t-way strategy," Soft Comput., 28(1): 415-435, 2024.

[64] E. Pira, M. Khodizadeh-Nahari, "Combinatorial t-way test suite generation using an improved asexual reproduction optimization algorithm," Appl. Soft Comput., 150: 111070, 2024.

[65] S. Li, Y. Song, Y. Zhang, "Combinatorial test case generation based on ROBDD and improved particle swarm optimization algorithm," Appl. Sci., 14(2): 753, 2024.

[66] M. Friedman, "A comparison of alternative tests of significance for the problem of m rankings," Ann. Math. Stat., 11(1): 86-92, 1940.

Biographies

Seyyed AmirHossein Eshghazadi received his B.Sc. degree in Computer Sciense from the Azarbaijan Shahid Madani University, Tabriz, Iran. [2017- 2021], the M.Sc. degree in Computer Engineering (Software) from the Azarbaijan Shahid Madani University, Tabriz, Iran. [2021- 2023] Currently, he is the CEO and developer of Dolphin Technology Pioneers Co. His activities are in the field of medical software engineering and software testing and implementation.

- Email: ea.amirhossein@gmail.com
- ORCID: 0009-0009-1282-4326
- Web of Science Researcher ID: NA
- Scopus Author ID: NA
- Homepage: NA

Einollah Pira received his B.Sc. degree in Computer Engineering (software) from the University of Kharazmi, Tehran, Iran [1996–2000], the M.Sc. degree in Computer Engineering (software) from the Sharif University of Technology, Tehran, Iran [2000–2002], and Ph.D degree in Computer Engineering (software) from Arak University, Iran [2013-2017]. Currently, he is an Associate Professor with Department of Computer Engineering, Azarbaijan Shahid Madani University, Tabriz, Iran. His research interests include model checking, formal methods, software testing, evolutionary computation, and machine learning.

- Email: pira@azaruniv.ac.ir
- ORCID: 0000-0001-9010-6113
- Web of Science Researcher ID: NA
- Scopus Author ID: 55941352000
- Homepage: http://pajouhesh.azaruniv.ac.ir/_Pages/ResearcherEn.aspx?ID=6617

Mohammad Khodizadeh-Nahari received his B.Sc. degree in Computer Software Engineering from the Isfahan University of Technology in 1998, and the M.Sc. degree in Information Technology (e-commerce) from the Amirkabir University of Technology in 2008, and his Ph.D. in Computer Software Engineering from the Isfahan University of Technology in 2020. He is currently an Assistant Professor at Azarbaijan Shahid Madani University (ASMU). He worked for different companies in public and private sectors for 10+ years before joining ASMU. He also served as the head of Information Technology department at the ASMU from 2010 to 2015, and as the head of E-Learning department at the ASMU from 2020 until now. His research interests include big data algorithms, data mining, machine learning, data fusion, data integrity and Blockchain.

- Email: m.khodizadeh@azaruniv.ac.ir
- ORCID: 0009-0007-7416-3100
- Web of Science Researcher ID: NKP-5936-2025
- Scopus Author ID: 57201367872
- Homepage: http://pajouhesh.azaruniv.ac.ir/_Pages/Researcher.aspx?ID=1122

Alireza Rouhi received his B.Sc. at Kharazmi University of Tehran in September, 2000; M.Sc. at Sharif University of Technology in June 2004, and Ph.D. at University of Isfahan in September 2017, all in Software Engineering field. He rewarded as outstanding researcher of Ph.D. students at Faculty of Computer Engineering, University of Isfahan in 2017. Currently, he is an Associate Professor at Azarbaijan Shahid Madani University, Tabriz, Iran. He is interested in Software Engineering in general and Formal Specification, Model Transformation, Metaheuristics, and Social Networks in particular.

- Email: rouhi@azaruniv.ac.ir
- ORCID: 0000-0003-1494-3467
- Web of Science Researcher ID: L-2209-2018
- Scopus Author ID: 57189992181
- Homepage: http://pajouhesh.azaruniv.ac.ir/_Pages/ResearcherEn.aspx?ID=5384