applied sciences

MDPI

*Article*

# Combinatorial Test Case Generation Based on ROBDD and Improved Particle Swarm Optimization Algorithm

**Shunxin Li [1], Yinglei Song [1,*] and Yaying Zhang [2]**

[1] School of Automation, Jiangsu University of Science and Technology, Zhenjiang 212003, China; 211110303112@stu.just.edu.cn
[2] China Ship Scientific Research Center, Wuxi 214082, China; jinyx@cssrc.com.cn
* Correspondence: jsjxy_syl@just.edu.cn

**Abstract:** In applications of software testing, the cause–effect graph method is an approach often used to design test cases by analyzing various combinations of inputs with a graphical approach. However, not all inputs have equal impacts on the results, and approaches based on exhaustive testing are generally time-consuming and laborious. As a statute-based software-testing method, combinatorial testing aims to select a small but effective number of test cases from the large space of all possible combinations of the input values for the software to be tested, and to generate a set of test cases with a high degree of coverage and high error detection capability. In this paper, the reduced ordered binary decision diagram is utilized to simplify the cause–effect graph so as to reduce the numbers of both the inputs and test cases, thereby saving the testing cost. In addition, an improved particle swarm optimization algorithm is proposed to significantly reduce the computation time needed to generate test cases. Experiments on several systems show that the proposed method can generate excellent results for test case generation.

**Keywords:** test case generation; reduced ordered binary decision diagram; improved particle swarm optimization algorithm; combinatorial test

check for updates

**Citation:** Li, S.; Song, Y.; Zhang, Y. Combinatorial Test Case Generation Based on ROBDD and Improved Particle Swarm Optimization Algorithm. *Appl. Sci.* **2024**, *14*, 753. https://doi.org/10.3390/app14020753

Academic Editors: Luigi Fortuna and Atsushi Mase

Received: 20 October 2023
Revised: 9 January 2024
Accepted: 10 January 2024
Published: 16 January 2024

**Copyright:** © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

## 1. Introduction

As the demand and scale of application software continue to increase, automated testing has become the mainstream trend in software testing. Software testing is the process of evaluating software applications to determine whether there are any defects, ensuring the quality of the software product. However, software testing is generally a time-consuming and labor-intensive process, accounting for 50% of the software development cost [1]. The automatic generation of test cases is a crucial aspect of software-testing automation. The traditional manual method of generating test cases often results in a limited number of them, which is not only costly but also requires highly skilled and experienced testing personnel to ensure code coverage. This leads to the fact that manual test case generation methods often adversely affect the progress of software testing during the rapid iteration process of software development. Therefore, they are gradually being replaced by automated test case generation [2]. Indeed, due to the diversity of the programs to be tested and the uncertainty surrounding test cases, efficiently generating a high-coverage and concise test case set has become a challenging technical task. The traditional techniques for generating test cases, primarily based on random generation or the conversion of software structures into symbolic calculations, are gradually no longer applicable to the automatic generation of test cases for large and complex software. Because test case generation is an undecidable problem, many search algorithms have inherent flaws and weaknesses. Test case generation aims to produce a finite and optimal set of test cases, where "optimal" means a minimum set of cases that can suggest as many errors as possible according to the test adequacy criteria Test adequacy criteria are used to determine the quality of the test cases and whether the testing process is complete [3].

The cause–effect graph method is a method that can be used to design test cases for software by analyzing various combinations of inputs based on a graphical method, which is suitable for examining the outputs of the software under various combinations of program input conditions. Because a cause–effect graph is a graphical representation of Boolean expressions, it can be easily converted between them, and the cause–effect graph provides more intuitive access to information. In the cause–effect graph approach, firstly, by looking at the cause–effect graph, we can know the condition (cause), the operation (effect), and the relationship between them, and then we are able to derive the truth table corresponding to the cause and effect. One of the main challenges in using cause–effect graphs for software testing is how to reduce the size of the set of generated test cases while maintaining the quality of the tests. Several approaches have been proposed to solve this problem [4–6], but there are still some difficulties: the methods are complex, and redundancy can lead to the generation of unnecessary test cases.

However, it is impractical to always test all possible use cases exhaustively because the size of the input space grows exponentially as the number of input variables increases. Therefore, approaches based on combinatorial testing are often used to generate test cases. Such methods find faults by detecting all combinations of values taken in the software system factors. The software's own factors and the interactions among them can trigger faults. As technology continues to evolve and the software complexity continues to increase, the combinatorial space of software factors becomes exceptionally large, making it difficult to provide comprehensive coverage of the combined test cases. Therefore, the efficient way to select and generate test cases with small sizes and high error detection rates is a critical task in the field of combinatorial testing. Generating the smallest set of combinatorial test cases is an NP-complete problem [7]. Researchers have developed and tested various methods to generate the smallest possible test case set, including the ant colony optimization (ACO) algorithm, simulated annealing (SA) algorithm, genetic algorithm (GA), particle swarm optimization (PSO) algorithm, etc. These metaheuristic search algorithms can efficiently and effectively search within the combinatorial space of all possible test cases and generate a combinatorial test case set [8]. Experimental results show that, compared with other methods, metaheuristic search algorithms can reduce the size of the generated combinatorial test case set to a certain extent. However, a large amount of computation time is still needed by metaheuristic optimization approaches to generate optimal test cases [9]. The basic optimization algorithm has been improved and upgraded based on the needs and characteristics of specific optimization problems, and the strategies for improvement include chaotic mapping [10,11], Cauchy variation [12], opposition-based learning [13], and so on.

Current research related to the application of PSO algorithms to combinatorial tests shows that they have certain advantages in terms of the size of the set of combinatorial test cases generated [14]. In addition, the PSO algorithm has the advantages of easy use, rapid computation, and good robustness. However, the traditional PSO algorithm also has several weaknesses, including a low convergence accuracy, the ease of falling into local optima, premature convergence or stagnation, etc. Intensive research has been performed to improve the performance of the PSO algorithm. However, most of the efforts focus on improving the optimality of the generated combinatorial test case set, with less consideration of the computation time needed for convergence. Therefore, a new PSO algorithm with improved computational efficiency is highly desirable for the efficient generation of optimal combinatorial test case sets for software testing.

This paper proposes a new approach for the efficient generation of optimal test case sets for software testing. As the first step of the proposed approach, the cause–effect diagram of the software system under testing is obtained through the if-then-else normal form (INF) specification. In the second step, the reduced ordered binary decision diagram (ROBDD) is utilized to simplify the cause–effect diagram to eliminate the influence of useless causes. Finally, test cases for pairwise testing are generated with an improved PSO algorithm. Experiments show that the proposed approach can reduce the number of

test cases generated while ensuring the coverage of pairwise tests, thereby achieving the purpose of cost reduction.

The rest of the paper is organized as follows. In Section 2, the work related to cause–effect graphs and PSO algorithms in generating test cases is reviewed. In Section 3, the required basics for the proposed test case generation approach are briefly introduced and the technical details of the approach are presented. In Section 4, the performance of the proposed approach is tested and evaluated on a known case study. Section 5 concludes the paper and provides an outlook for future work.

## 2. Related Work

### 2.1. Cause–Effect Graph

Lee et al. [15] discussed how to model captured requirements from natural language using the cause–effect graph. Subsequently, they transformed the cause–effect graph into Simulink® and Stateflow® to identify conflicting requirements and automatically generate a set of test vectors. Son et al. [16] proposed a methodology for generating test cases from cause–effect graphs based on model transformation, and the model transformation rules were generated using the ATLAS Transformation Language (ATL) and executed in the Eclipse development environment. Chung [17] presents CEGPairGen, a tool that employs an incremental test generation strategy specifically designed for scenarios in which the number of pairs to be covered expands rapidly. Akram et al. [18] employ the ROBDD to identify the major effects from the cause–effect graph and utilize the PSO algorithm to select the optimal test suite, ensuring the coverage of all pairwise combinations of input parameters. Woo et al. [19] present an approach for automatically generating cause–effect relationships from informal requirement specifications in Korean, using a mechanism that simplifies the requirements through the following procedures: (1) the automatic simplification of the informal requirement sentences, (2) the generation of a Condition/Conjunction/Clause Tree (C3Tree) Model, and (3) cause–effect relationship generation. A novel approach called Spectral Testing is proposed by Deniz et al. [6] for generating test inputs for Boolean specification models. This method utilizes the spectral analysis of Boolean expressions and applies mutations to the original expressions to generate test inputs.

### 2.2. The PSO Algorithm

Huang et al. [20] present an analysis of test case generation using the PSO algorithm, introducing two enhancements to improve its performance, including the group self-activity feedback (SAF) operator and the Gauss mutation (G) with a changing inertia weight. Sheng et al. [21] present two innovative methods for constraint test case generation (PCTG) based on PSO. These methods incorporate strategies to address the constraints by avoiding the selection of conflicting test cases and replacing them by adopting different approaches to handle the constraints. These methods offer distinct perspectives that guide the search process for optimal solutions. Based on a comprehensive review of recent research on diverse search-based optimal control methodologies, Prasad et al. [22] introduce a swarm-smart exploring technique aimed at generating test records that are relatively close in nature. The proposed approach seeks to enhance the effectiveness of the search process and promote the generation of diverse and comprehensive test cases. To tackle the issue of low efficiency in test case generation, Wang et al. [23] introduce a novel approach called Selection and Mutation Strategy-Based Elite Opposition-Learning Particle Swarm Optimization (SM-EOLPSO). Experimental results demonstrate that the proposed algorithm exhibits competitiveness in terms of the iteration count and generation time for automatic test case generation.

## 3. The Proposed Approach

The major focus of the research in this paper includes the following two aspects:

① The development of a method that can reduce the size of the test case set while maintaining the test quality when designing test cases using cause–effect graphs.

It is well known that exhaustive testing requires a very large set of test cases, which is both time-consuming and labor-intensive, and we can take a combination of tests to reduce the number of test cases. Sometimes, because of iterative updates to the program, bad code flavors, such as redundant code, are inevitable, which makes the designed test cases not always effective. We use the ROBDD to circumvent this problem by removing some unnecessary inputs;

② The provision of a solution to the problem that the particle swarm algorithm itself is time-consuming and thus may lead to inefficiency in generating test cases.

As the number of causes in the causal diagram increases, the solution space for the PSO to generate test cases increases, which can make the PSO much less efficient. In this paper, the search strategy of the particle swarm is improved to effectively reduce the convergence speed of the PSO.

### *3.1. Cause–Effect Graph*

#### 3.1.1. Related Concepts

A cause–effect graph is a graphical tool that describes the combinations of input conditions and the output corresponding to each combination. It is mainly applicable when there is some interconnection between various input conditions, or when the output depends on the combination of various input conditions.

The basic components of a cause–effect graph include variables and arrows. Two variables that are connected by arrows indicate that there is a causal relationship between the two variables for at least one individual in the total. If there is no arrow connection between two variables, it means that there is no causal relationship between the two variables for any individual overall. Figure 1 shows an example of a cause–effect diagram.

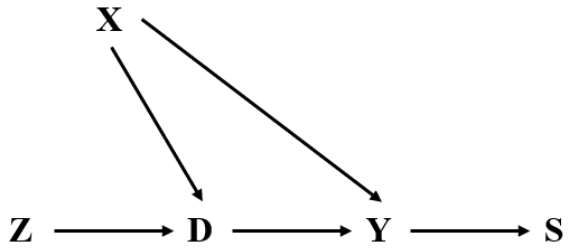



**Figure 1.** The basic components of a cause–effect graph.

① Confounding factor

In Figure 1, there are two arrows pointing to D and Y from X. X is called the common cause of D and Y, and X is also called the confounding factor of D affecting Y;

② Common result

In Figure 1, there are two arrows pointing to Y, indicating that Y is a common result of D and X;

③ Path

A path is formed by a series of arrows that connect two variables. For example, there are two paths connecting D and Y in Figure 1, including D→Y and D←X→Y. The former is called a causal path, and the latter is called a backdoor path. A backdoor path [24,25] is an on-causal path that connects the cause variable (D) and the result variable (Y). It has an arrow pointing to the cause variable (D).

In addition, other unobserved perturbations affecting a variable in the graph may not appear in the causal graph if they are all independent of the other variables.

3.1.2. Graphical Symbols in the Cause–Effect Graph

(1) Basic graphic symbols

① Identical
An identical symbol describes a relationship in which the result appears if and only if the cause appears. Figure 2a provides an example of an identical symbol. If $C_1$ occurs, then $E_1$ occurs; if $C_1$ does not occur, then $E_1$ does not occur;

② Not
A not symbol describes a relationship in which the result appears if and only if the cause does not appear. Figure 2b provides an example of a not symbol. If $C_1$ occurs, $E_1$ does not occur; if $C_1$ does not occur, $E_1$ occurs;

③ Or
An or symbol describes a relationship in which the result appears if and only if at least one of the related causes appears. Figure 2c provides an example of an or symbol. If one of $C_1$ and $C_2$ occurs, then $E_1$ occurs; if neither $C_1$ nor $C_2$ occurs, then $E_1$ does not occur;

④ And
An and symbol describes a relationship in which the result appears if and only if all related causes appear. Figure 2d provides an example of an and symbol. $E_1$ occurs only if both $C_1$ and $C_2$ occur; if one of $C_1$ and $C_2$ does not occur, $E_1$ does not occur.

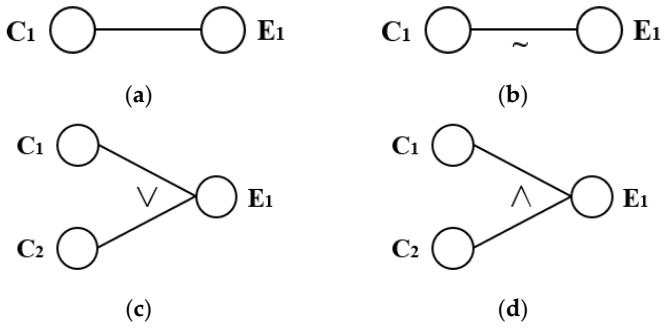



**Figure 2.** Basic graphical symbols in a cause–effect graph: (**a**) identical; (**b**) not; (**c**) or; (**d**) and.

(2) Constraint relationship graphic symbols
To represent the constraints in a cause–effect graph, some symbols are used to identify them. Constraint relationship graphs are either between causes (input conditions) or effects (output results).
Four main constraints are considered in terms of causes, as follows:

① E (Mutually Exclusive)
As shown in Figure 3a, an E constraint symbol connects the two causes a and b, as a and b do not occur at the same time. In other words, at most, one of them occurs at any time;

② I (Inclusive)
Figure 3b provides an example of an I constraint symbol, where at least one of the three causes (a, b, and c) appears;

③ O (Only)
Figure 3c shows an example of an O constraint symbol, where one and only one of the two causes (a and b) must occur;

④ R (Required)
Figure 3d provides an example of an R constraint symbol, where b must appear when a appears.

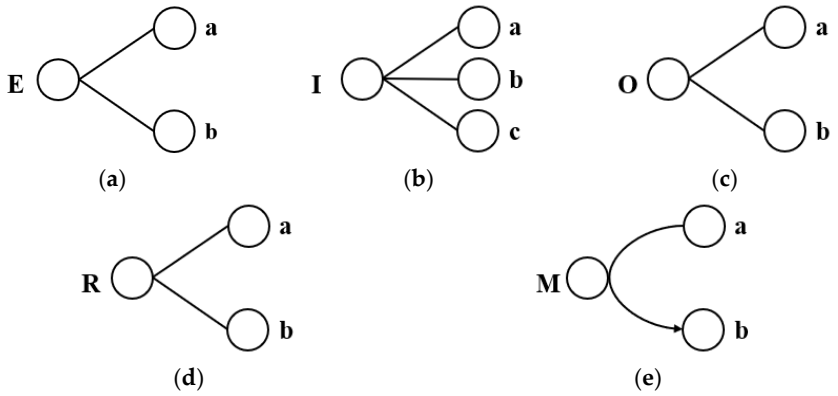



**Figure 3.** Graphical symbols of constraint relationships of the cause–effect graph: (**a**) E (mutually exclusive); (**b**) I (inclusive); (**c**) O (only); (**d**) R (required); (**e**) M (mask).

One constraint is considered in terms of the results as follows:

① M (Mask)
As shown in Figure 3e, an M constraint symbol exists between the two results a and b if b does not appear when a appears and b may or may not appear when a does not appear.

The difference between a mutually exclusive constraint and an only one is that a mutually exclusive constraint allows for not making a choice, and if a choice is required, then, at most, one choice is chosen. An only constraint requires that one and only one must be chosen. The main difference between them in a computer program is that there is no default option for a mutually exclusive constraint, but there is a default option for the only constraint.

#### 3.1.3. Steps in Designing Test Cases Using Cause–Effect Graph Approach

① Analyze software specification descriptions for causes (i.e., input conditions or equivalents of input conditions) and effects (i.e., output conditions), and assign an identifier to each cause and effect;
② Analyze the semantics of the software specification description to identify the relationships between the cause and effect, and construct a cause–effect graph based on these relationships;
③ Due to grammatical or contextual constraints, certain combinations formed between a cause and another cause, or a cause and an effect, are not possible, and in order to indicate these special cases, constraints or limitations are indicated by notations on the cause–effect graph;
④ Convert the cause–effect graph into a decision table;
⑤ Take each column of the decision table and use it as a basis for designing the test cases.

### *3.2. ROBDD*

#### 3.2.1. Related Concepts

(1) Boolean function

A Boolean function describes how to determine the Boolean output based on some logical calculation of a Boolean input. A Boolean function ($F$) with $n$ variables is shown in Equation (1):

$$F : \{0,1\}^n \longrightarrow \{0,1\} \tag{1}$$

The function has $2^n$ different possible inputs, and the output is a Boolean value of True or False, represented by 1 or 0.

Define new Boolean functions with $n-1$ variables, as shown in Equations (2) and (3):

$$F_{x_1}(x_2, \ldots, x_n) = F(1, x_2, x_3, \ldots, x_n) \tag{2}$$

$$F_{x_1'}(x_2, \ldots, x_n) = F(0, x_2, x_3, \ldots, x_n) \tag{3}$$

where $F_{x_1}$ and $F_{x_1'}$ are the cofactors of $F$. $F_{x_1}$ is a positive factor and $F_{x_1'}$ is a negative factor;

(2) Shannon decomposition

The Shannon decomposition is a transformation of a Boolean function. It expresses any Boolean function as any one of the variables multiplied by a subfunction, plus the inverse of this variable multiplied by another subfunction, as shown in Equation (4):

$$F(x_1, x_2, \ldots, x_n) = x_i F_{x_i} + x_i' F_{x_i'} \tag{4}$$

(3) If-then-else normal form

An if-then-else normal form (INF) operator is denoted by $x \longrightarrow y_0, y_1$ . An INF operator is defined as shown in Equation (5):

$$x \longrightarrow y_0, y_1 = (x \wedge y_0) \vee (\neg x \wedge y_1) \tag{5}$$

That is, the result is $y_0$ if the value of $x$ is 1; otherwise, the result is $y_1$.

It is straightforward to see that all the logical operators can be expressed using if-then-else operators and the constants 0 and 1 only. Furthermore, an if-then-else operator can be implemented in such a way that all tests are performed only on un-inverted variables. For example, $\neg x$ is $x \rightarrow 0, 1$. In this way, an INF paradigm can be constructed based on if-then-else operators.

An INF paradigm is a Boolean expression constructed entirely from if-then-else operators and constants, so only tests on the variables need to be performed. For the Boolean expression ($t$), let $t[0/x]$ denote the Boolean expression obtained by assigning the variable $x$ in the $t$ to 0, and Equation (6) can be obtained for the $t$:

$$t = x \longrightarrow t[1/x], t[0/x] \tag{6}$$

Equation (6) is the Shannon expansion of the expression $t$ for the variable $x$. This simple equation has several useful applications. The first is that an INF can be generated from any expression ($t$). If the $t$ does not contain any variables, then it is equal to 0 or 1 (i.e., an INF). Otherwise, a Shannon expansion can be generated for a variable ($x$) in the expression ($t$) where $t[1/x]$ and $t[0/x]$ have one less variable than the $t$, and an INF for the $t$ can thus be constructed recursively based on Equation (6).

3.2.2. ROBDD

(1) BDD

For a Boolean function ($f(x_1, x_2, \ldots, x_n)$) from $\{0,1\}^n$ to $\{0,1\}$, a binary decision diagram (BDD) can be constructed to represent $f(x_1, x_2, \ldots, x_n)$ and a family of Boolean functions ($\#f(x_1, x_2, x_3, \ldots, x_n)$) related to $f(x_1, x_2, \ldots, x_n)$ with a directed acyclic graph. The graph satisfies the following five points:

① Nodes in the BDD are divided into three categories: root nodes, final nodes, and internal nodes;

② There are only two end nodes, marked as 0 and 1. Let $t.val$ denote the value associated with an end node ($t$); $t.val \in \{0,1\}$ must hold for the $t$, denoting the Boolean constants 0 and 1;

③ Each internal node ($u$) is associated with a quaternion property: $(f^u, var, low, high)$, where $f^u$ denotes the Boolean function corresponding to node $u$ and $f^u \in \#_\pi f(x_1, x_2, \ldots, x_n)(f^u = f(x_1, x_2, \ldots, x_n)$ if u is the root node; $var$ denotes the marker variable of node $u$, where low denotes the child of node $u$ when $u.var = 0$ (indicating that the marker variable ($var$) of node $u$ takes a value of 0), and high denotes the child of node $u$ when $u.var = 1$ (indicating that the marker variable ($var$) of node $u$ takes a value of 1);

④ Each internal node has two output branching arcs that connect the node to its two child nodes. The arc connecting node $u$ to $u.low$ is called a 0-edge, and the arc connecting node $u$ to $u.high$ is called a 1-edge;

⑤ Each variable in the Boolean function $f(x_1, x_2, \ldots, x_n)$ appears once, at most, in any directed path in the BDD.

(2) OBDD

For a Boolean function ($f(x_1, x_2, \ldots, x_n)$) from $\{0,1\}^n$ to $\{0,1\}$ and a given variable order ($\pi$), a BDD representing $f(x_1, x_2, \ldots, x_n)$ and the family of Boolean functions $\#f(x_1, x_2, x_3, \ldots, x_n)$ is said to be an ordered binary decision diagram (OBDD) of $f(x_1, x_2, \ldots, x_n)$ if the variables on any directed path all appear in the order specified by the $\pi$. The node $u$ in each OBDD represents the Boolean function $f^u$ from $\{0,1\}^n$ to $\{0,1\}$, where $u$ satisfies the following two points:

① If $u$ is a final node, then $f^u = u.val$;

② If $u$ is an internal node, then the following equation is used:

$$f^u = u.var \cdot f^{u.high} + (u.var)' \cdot f^{u.low} = u.var \cdot f^u_{|u.var=1} + (u.var)' \cdot f^u_{|u.var=0} \quad (7)$$

where $f^u_{|u.var=1}$ and $f^u_{|u.var=0}$ denote the functions obtained after the variable $u.var$ in $f^u$ takes the values 1 and 0, respectively (i.e., the Boolean functions corresponding to the children $u.high$ and $u.low$ of node $u$). Clearly, in the decomposition process, choosing a different variable order will generally result in a different decomposition result for $f^u$. Therefore, different orders of variables lead to different OBDDs for the same function. For example, Figure 4 shows the OBDD of the Boolean function $f(x_1, x_2, x_3) = (x_1 + x_2) \cdot x_3$ under the variable order $\pi : x_1 < x_2 < x_3$.

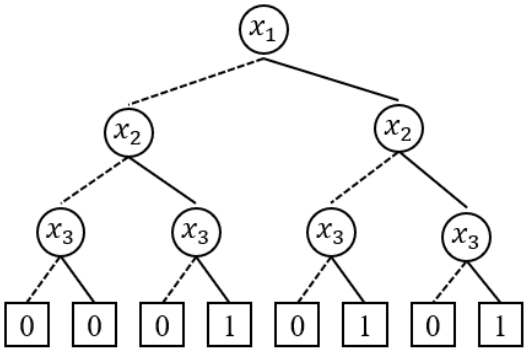



**Figure 4.** OBDD for Boolean function $f(x_1, x_2, x_3) = (x_1 + x_2) \cdot x_3$ given the variable order $\pi : x_1 < x_2 < x_3$.

(3) ROBDD

An OBDD is a reduced ordered binary decision diagram (ROBDD) if the internal nodes in the OBDD satisfy the following two conditions:

① For each internal node ($u$), $u.low \neq u.high$;

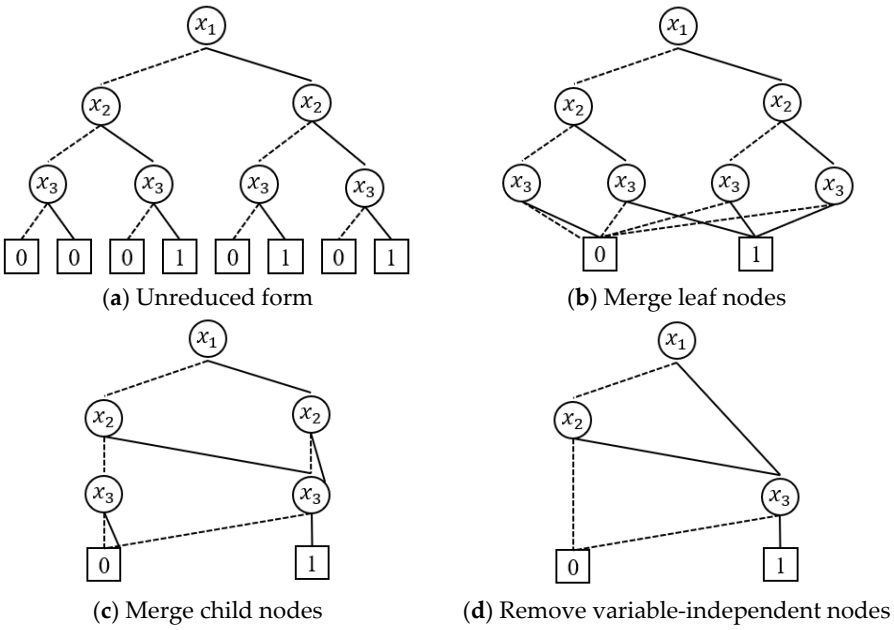
② For different internal nodes ($u$ and $v$) that satisfy $u.var = v.var$, $u.low \neq v.low$ or $u.high \neq v.high$ holds.

An OBDD that does not satisfy the above conditions can be simplified to an ROBDD by repeatedly applying the following simplification rules:

**Rule 1** (Deletion Rules)**.** *For node u in the OBDD, if u.low = u.high, node u is deleted and the parent node of node u is directly connected to the node corresponding to u.low.*

**Rule 2** (Merging Rules)**.** *For two different nodes (u and v) in the OBDD, if u.var = v.var, u.low = v.low, and u.high = v.high, node u is deleted, and the parent of node u is directly connected to node v .*

As an example, Figure 5a–d illustrate the simplification process of the OBDD for the Boolean function $f(x_1, x_2, x_3) = (x_1 + x_2) \cdot x_3$ under the variable order $\pi : x_1 < x_2 < x_3$.




(**a**) Unreduced form (**b**) Merge leaf nodes

(**c**) Merge child nodes (**d**) Remove variable-independent nodes

**Figure 5.** The simplification process of the OBDD for the Boolean function $f(x_1, x_2, x_3) = (x_1 + x_2) \cdot x_3$.

In many cases, the original cause–effect graph may have many events that are not related to the outcome or are recurring, so it is important to remove these extraneous factors without affecting the logical relationship among the events. This can be achieved via the deletion rules and merging rules. The ROBDD is a concise graphical representation of the Boolean functions. In the ROBDD, the paths from the root node to the endpoint "1" are the paths that must be traveled for normal operation. The nodes on these paths represent the causes associated with the results.

### *3.3. Pairwise Testing*

Before conducting the test, the input space needs to be modeled in order to obtain the factors and optional values affecting the System Under Test (SUT), which are defined as follows:

**Definition 1** (Factors and Optional Values)**.** *Each individual input for a software system is a factor of the system. Each possible value for a given factor is an optional value for the factor. A software system with* $k$ *factors can be represented by* $F = \{f_1, f_2, \ldots, f_k\}$*, where the* $l_i$ *optional values for the* $i$*-th factor* $f_i$ *can be represented as* $D_i = \{v_{i,1}, v_{i,2}, \ldots, v_{i,l_i}\}$*;*

**Definition 2** (Test Case)**.** *For a software system containing* $n$ *factors to be tested, a test case can be represented by test* $= \{x_1, x_2, \ldots, x_n\}(x_1 \in D_1, x_2 \in D_2, \ldots, x_n \in D_n)$*;*

**Definition 3** (Test Case Set)**.** *The matrix* $T = (x_{i,j})_{m \times k}$*, which contains* $m$ *test cases, is said to be the set of test cases of size* $m$ *of the test system, where* $T$ *is an* $m \times k$ *matrix and each row in* $T$ *is a test case.*

The traditional combination design approach is based on a coverage matrix. Specifically, the set of combination test cases is represented by a matrix in which each row represents a test case, each column represents a factor of the SUT, and each item (entry) represents the value of the corresponding factor. Cohen et al. [26] provides the definition of a coverage array (CA) to describe the set of test cases;

**Definition 4** (Coverage Matrix). *The coverage matrix $CA(N, k, s, t)$ is an $N \times k$ matrix with $s$ optional values for each column, and an arbitrary $N \times t$ submatrix that contains all t-tuples over the s-value domain. $N$ denotes the number of rows, $k$ denotes the number of factors, $s$ denotes the number of values for each factor, and $t$ denotes the interaction strength. The interaction strength (t) means that any valid combination of any t parameters in the range of values is covered by at least one test case. A coverage matrix with $t = 2$ is called a pairwise covering array;*

**Definition 5** (Optimal Test Case Set). *$TS$ is said to be the optimal test case set if $T$ satisfies all possible combinations that have occurred at least once and the number of rows (m) in the matrix (T) is the minimum value.*

Pairwise testing is a test case generation technique that can effectively reduce the number of test cases by considering all the dimensions and values of the test variables while avoiding the exhaustive testing of all combinations of variable values. The pairwise testing approach solves the problem of test case explosion to a certain extent.

The idea of the pairwise algorithm is to streamline the use cases by deleting a group of use cases if their two-by-two combinations occur in all other combinations. A pairwise algorithm is based on the following two assumptions: (i) each dimension is orthogonal, (i.e., there is no intersection of any dimension with any other dimension); (ii) the majority of defects arise from a single factor or the interaction of two different factors. In fact, according to mathematical statistical analysis, 73% of the defects are caused by single-factor or two-factor interactions, and 19% of the defects are caused by three-factor interactions.

An N-wise testing set comprises test cases that contain all possible combinations of the values of each set of N factors. In How We Test Software at Microsoft [27], it is recommended to start testing with two-pair (pairwise) testing sets and gradually increase the dimensionality of the combinations up to six-wise combination testing. Recent research has shown that six-pair testing sets, in terms of the defect removal efficiency and structural code coverage strategies, are comparable to exhaustive coverage. However, in practice, the number of test cases grows exponentially as the dimensionality of the combination increases, so pairwise or three-wise testing sets are more suitable for real projects. Furthermore, Telcodia et al. [28] showed that most field faults are caused by the interaction of incorrect individual values or pairs of values; thus, in this paper, we choose $t = 2$. In addition, due to the notation of the INF specification in the ROBDD, the sis is chosen to be 2.

For combinatorial tests, intensive research has been performed to optimize the size of the test sets. For the instance in which $t = 2$ and $s = 2$, optimal solutions can be obtained with the methods proposed by Rényi [29], Katona [30], and Kleitman and Spencer [31]. Results for solutions to more complex instances were obtained by Soohak Choi et al. [32]. The numbers of test cases ($m$) in the optimal test case sets for cases in which $t = 2$, $s = 2$, and $3 < k < 3003$ were calculated and are shown in Table 1. In practice, different individual factors are often not mutually independent and certain constraints may exist among them. The approaches mentioned above thus cannot be directly applied to optimize the number of test cases for many software systems.

**Table 1.** Statistics on the number of optimal test cases for $t = 2$, $s = 2$, and $3 < k < 3003$.

| k | 3 | 4 | 5–10 | 11–15 | 16–35 | 36–56 | 57–126 | 127–210 | 211–462 | 463–792 | 793–1716 | 1717–3003 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| m | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |

### *3.4. The PSO Algorithm*

#### 3.4.1. The Standard PSO Algorithm

The PSO algorithm is an evolutionary computational technique that originated from the study of the behavior of flocks of birds foraging. The idea of the algorithm originates from the study of the foraging behavior of bird flocks, where the flock finds the optimal destination through collective information sharing. Suppose that in a $D$-dimensional

target search space, there is a swarm that contains $N$ particles, where the $i$-th particle is represented by a vector with $D$ dimensions, denoted as $X_i = (x_{i1}, x_{i2}, \ldots, x_{iD}), i = 1, 2, \ldots, N$. The velocity of the $i$-th particle is also a $D$-dimensional vector, denoted as $V_i = (v_{i1}, v_{i2}, \ldots, v_{iD}), i = 1, 2, \ldots, N$. The optimal position obtained by the $i$-th particle is $P_{id,pbest} = (p_{i1}, p_{i2}, \ldots, p_{iD}), i = 1, 2, \ldots, N$. The optimal position based on the search results of all the particles in the population is $P_{d,gbest} = (p_{1,gbest}, p_{2,gbest}, \ldots, p_{D,gbest})$. The fitness value of the $i$-th particle, which is the value of the objective function for the optimal position searched by the particle, is denoted by $f_{i,p}$. The fitness value of the optimal position obtained by the population is $f_g$.

The $i$-th particle updates its velocity and position according to the following equations:

$$v_{id}^{k+1} = \omega v_{id}^{k} + c_1 r_1 \left( p_{id,pbest}^{k} - x_{id}^{k} \right) + c_2 r_2 \left( p_{d,gbest}^{k} - x_{id}^{k} \right) \tag{8}$$

$$x_{id}^{k+1} = x_{id}^{k} + v_{id}^{k+1} \tag{9}$$

where $c_1$ is the individual learning factor, and $c_2$ is the group learning factor.

In this paper, the goal of the PSO is to cover the maximum number of pairwise combinations of elements with the minimum number of test cases after using the ROBDD. In other words, the problem is how to generate the TS in Definition 5 of Section 3.3 when the number of inputs ($k$) and the number of input values (s) are known. So, the position of each particle ($X_i$) is an $m \times k$ matrix that represents $m$ test cases, and the system under the test has $k$ inputs. The fitness function (Fit($X_i$)) is defined as the combined coverage of the pairwise tests (i.e., the ratio of all possible value-taking outcomes that have occurred for all pairwise combinations). It can be calculated via Equation (10):

$$\text{Fit}(X_i) = \frac{n(X_i)}{4 \times C_k^2} \tag{10}$$

where $n(X_i)$ represents the number of combinations executed at least once, and $4 \times C_k^2$ represents the number of all possible combinations. The termination condition for the PSO is to make Fit($X_i$) reach 1. This means that enough test cases are designed so that every possible combination of the conditions in the judgment occurs at least once.

#### 3.4.2. Improved PSO Algorithm

Because the PSO algorithm needs a large particle swarm inertia factor ($\omega$) to traverse the entire space of solutions as much as possible in the early stage of the search, and a small $\omega$ in the late stage of the search to make the particles approach the optimal solution and reach convergence as soon as possible, the traditional PSO algorithm uses a fixed value for the $\omega$ in the process of searching, and the search efficiency is thus seriously affected. In this paper, an improvement rate ($\alpha_t$) is introduced to monitor the search process of the PSO algorithm so as to adjust the inertia parameter ($\omega$) adaptively to improve the search efficiency of the PSO algorithm, and the $\omega$ is updated according to Equation (11):

$$\omega_{(t+1)} = \begin{cases} \frac{\omega_{(t)}}{2} \; if \; \alpha_t \le IR_1 \\ \omega_{(t)} \; if \; IR_1 < \alpha_t \le IR_2 \\ 2 * \omega_{(t)} \; if \; \alpha_t > IR_2 \end{cases} \tag{11}$$

where $\alpha_t$ is the improvement rate and is calculated via Equation (12); $IR_1$ and $IR_2$ are the two constant threshold values determined based on experiments:

$$\alpha_t = \frac{\left( f_{g_{(t)}} - f_{g_{(t-1)}} \right)}{f_{g_{(t-1)}}} \tag{12}$$

A flowchart of the improved PSO algorithm is shown in Figure 6.

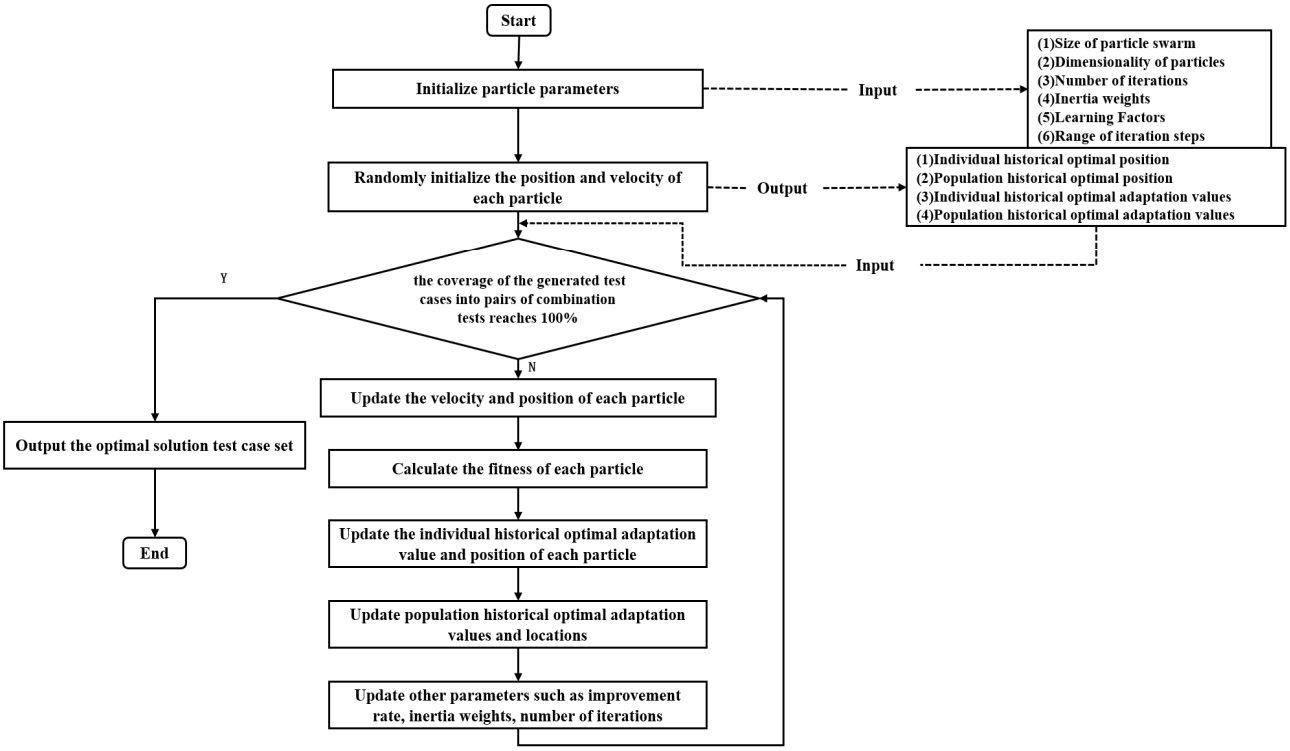



**Figure 6.** A flowchart of the improved PSO algorithm.

### *3.5. The Proposed Approach*

The proposed approach generates test cases based on the following steps:

① Obtain a cause–effect graph of the procedure according to the INF specification;
② The logical expression of the desired test part is obtained by assigning the cause logical variables according to the cause–effect graph;
③ An order of the logical variables is determined to construct an OBDD, which is then simplified using the deletion and merging rules to obtain an ROBDD. A node element in the ROBDD contains a variable, a left child node, and a right child node;
④ Determine the number of internal nodes in the ROBDD and denote it with $k$;
⑤ Initialize the number of particles ($N$), individual learning factor ($c_1$), population learning factor ($c_2$), inertia weight ($\omega$), improvement rate ($\alpha$), and maximum number of iterations ($N_g$). The position and the fitness value of the historical best for each individual particle are set to 0, and the position and the fitness value of the historical best for the entire population are set to 0;
⑥ The particle positions ($X_i$) and velocities ($V_i$) are initialized according to $k$ and Table 1, both of which are $m \times d$ matrices. Each particle position represents $m$ test cases, where each test case contains $d$ inputs, and the matrix elements are randomly assigned as 0 or 1. Each particle velocity is an $m \times d$ matrix of numbers randomly generated between 0 and 1;
⑦ Update the inertia weights according to Equation (11) and iterate the velocity according to Equation (8), where $r_1$ and $r_2$ are $m \times d$ matrices of numbers randomly generated between 0 and 1. The iteration process of the elements in each particle position matrix is as follows: if the absolute value of a position element determined from the velocity matrix is greater than a number randomly generated between 0 and 1, the position element is flipped;
⑧ Calculate the improvement rate ($\alpha_t$) according to Equation (12) and the fitness function according to Equation (10) and update the historical best fitness value for each individual and its position and the global historical best fitness value and its position.

If Fit($X_i$) > Fit($P_{ibest}$), replace the individual optimal position ($P_{ibest}$) with $X_i$. If Fit($X_i$) > Fit($P_{best}$), replace the population optimal position ($P_{best}$) with $X_i$;

⑨ Repeat steps ⑦ and ⑧ until the coverage of the generated test cases into pairs of combination tests reaches 100%;

⑩ Output the optimal test case set (TS).

## 4. Experimental Results

The hardware environment for all the experiments was an Intel Core i7 processor with 16 GB RAM; the software environment was the 64-bit Windows 11 operating system and MATALAB experiment platform.

### 4.1. Experimental Design

(1) To verify the effectiveness of the proposed method in this paper, a real-time boiler system was used as a test subject for comparison experiments with a non-norm-based method;

(2) To verify the effectiveness of the improved PSO algorithm, comparison experiments were conducted with the basic PSO algorithm on a large-scale input system. In addition, we compared the approach in this paper with other classical swarm intelligence algorithms.

### 4.2. Validity Experiments

The simplified boiler system used in this study consists of a natural-gas water tube boiler that produces saturated steam [4]. As the external demand changes, the steam flow varies rapidly and irregularly between zero and the maximum values. The water level of the boiler is regulated by controlling the inflow of water. The water level must be maintained between the upper and lower limits. If the water level is above the upper limit, water will be carried into the steam stream, causing damage to the equipment; if the water level is below the lower limit, the boiler piping may overheat and burst. If control of the water level is lost, the boiler will shut down. The water level and steam flow are measured via the instrumentation system. The sensor readings are transmitted to the control program via a communication link. The control program is expected to perform the following tasks:

① Adjusting the water level by turning the pump on or off to control the water inflow as needed;

② Monitoring all potential errors and, if any are found, issuing corrections or fixing requests;

③ Providing the boiler operators with the "best estimates" for different parameters;

④ Accepting the correct command from any operator.

The cause–effect graph of the system is shown in Figure 7. The meanings of the nodes are shown in Table 2.

**Table 2.** Illustration table of the cause–effect diagram.

| Number | Description |
|---|---|
| C221 | Externally initiated |
| C220 | Internally initiated |
| C203 | Instrumentation-system-initiated |
| C202 | Operator-initiated |
| C201 | Bad startup |
| C200 | Operational failure |
| C198 | Confirms the "shutdown" message |
| C197 | Confirms the keystroke entry |
| C196 | Multiple-pump failure (more than one) |
| C195 | Water level meter failure during startup |

**Table 2.** *Cont.*

| Number | Description |
|---|---|
| C194 | Steam rate meter failure during startup |
| C193 | Communication link failure |
| C192 | Instrumentation system failure |
| C191 | C180 and C181 |
| C190 | Water level out of range |
| C181 | Steam rate meter failure during operation |
| C180 | Water level meter failure during operation |

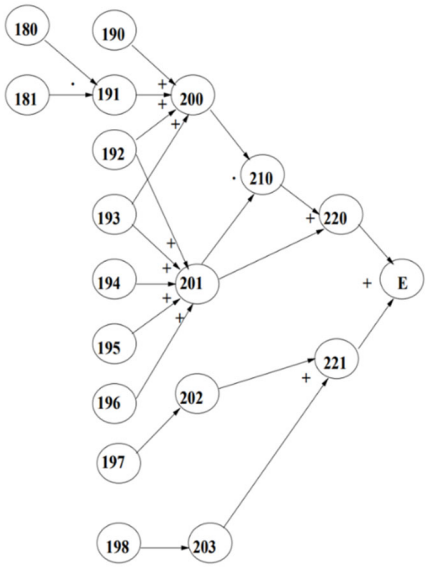



**Figure 7.** Cause–effect graph of the boiler system.

The logical variables assigned to the causes of the boiler system are shown in Table 3.

**Table 3.** Logical variables assigned to each cause in the system.

| Number | Variable Name |
|---|---|
| C180 | A |
| C181 | B |
| C190 | C |
| C192 | D |
| C193 | E |
| C194 | F |
| C195 | G |
| C196 | H |
| C197 | I |
| C198 | J |

This leads to a Boolean expression that is expressed as follows:

$$f = (\,AB + C + D + E)(D + E + F + G + H) + (D + E + F + G + H) + (I + J) \quad (13)$$

The order of the variables is determined to be $A < B < C < D < E < F < G < H < I < J$, and the simplified ROBDD can be obtained as shown in Figure 8. It is clear from Figure 8 that a ROBDD can effectively remove the cause nodes with no impact on the test.

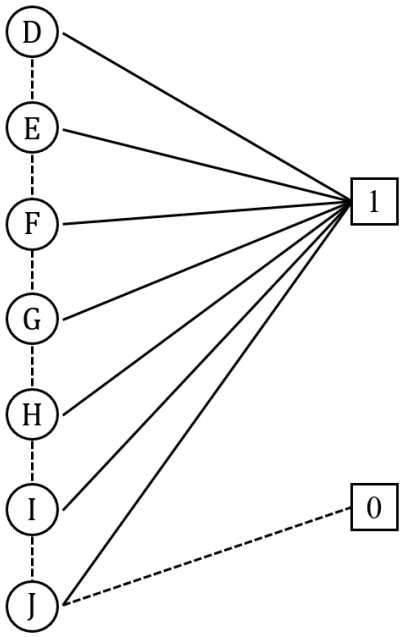



**Figure 8.** The ROBDD constructed for the boiler system.

Test cases were generated using the improved PSO algorithm, and the results are shown in Table 4. The parameters of the improved particle swarm algorithm that we used were set as follows: the number of particles (N) was set to 100, $c_1$ and $c_2$ were both set to 1.2, the initial value of $\omega$ was 0.8, and the maximum number of iterations ($N_g$) was 1000. The weight ($\omega$) was updated for each iteration via Equation (11), and $IR_1$ was taken as 0.001 and $IR_2$ was taken as 0.002.

**Table 4.** Table of test case generation results.

| Test Cases | D | E | F | G | H | I | J |
|---|---|---|---|---|---|---|---|
| C1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 |
| C2 | 0 | 0 | 1 | 1 | 0 | 1 | 0 |
| C3 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| C4 | 0 | 1 | 0 | 0 | 0 | 1 | 1 |
| C5 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| C6 | 0 | 1 | 1 | 0 | 1 | 0 | 1 |

To illustrate the advantages of the algorithm in this paper, comparison experiments were conducted with several non-norm-based methods for the generation of test cases, which included some of the more classical methods based on greedy algorithms and heuristics. The number of test cases obtained from the experiments in this paper was compared with the results obtained in other papers, including for the Jenny [33], IPOG [34], PICT [35], ABCS [36], GSTG [37], and TMWOA [38]. The comparison results are shown in Table 5.

**Table 5.** Comparison results on the number of test cases generated via different methods.

| Greedy Algorithms | | Heuristic Algorithms | | | Algebraic Methods | | Proposed |
|---|---|---|---|---|---|---|---|
| IPOG | PICT | ABCS | GSTG | TMWOA | TConfig | Jenny | |
| 8 | 9 | 7 | 8 | 8 | 9 | 10 | 6 |

In general, defects are often caused by one input or a combination of two inputs, and the principles for choosing a better combination for tests are thus as follows: the level

values of each factor can be tested, and each combination of the level values of any two factors can be tested. Combinatorial testing considers the system under testing as a system dependent on multiple factors, each of which has a discrete and finite value. The boiler system has 10 inputs, and each input has two levels. In total, $2^{10}$ test cases are required for exhaustive testing, while Table 5 shows that no more than 10 test cases are required for pairwise tests, which significantly reduces the number of test cases and saves the cost of testing.

Pairwise combinatorial testing generates a set of test cases that can cover all combinations of the values of any two factors, theoretically exposing all the defects triggered by the combined effect of the two factors. It is clear from Table 5 that the number of test cases in the test case set generated via the method in this paper was six, which is the least among all the tested methods. The fitness function used can ensure that the pairwise test combination coverage meets the requirements. It can be seen that the ROBDD can effectively reduce redundant inputs, thereby reducing the number of test cases generated and ultimately reducing costs.

We also used other examples: ① the RCCS (the railroad crossing critical system) [39], the cause–effect graph of which is given by Krupalija et al. [40]; ② the TCAS II (the aircraft collision avoidance system) [41]; ③ a very large cause–effect graph from [42], which contains eighteen cause nodes and seven effect nodes. We give the INF of a part of the system in Table 6.

**Table 6.** INF specifications as part of the system.

| Project | INF | No. of Variables |
|---|---|---|
| RCCS | $(\overline{C_1C_2})\,(\overline{C_1C_3})\,(\overline{C_2C_3})\left(C_2 \cup \left(\overline{((C_1\overline{C_4})C_4) \cup (C_3C_4)}\right)\right)$ | 4 |
| TCAS II | $a((c \cup d \cup e)g \cup af \cup c(f \cup g \cup h \cup i)) \cup (a \cup b)(c \cup d \cup e)i$ | 9 |
| | $(ac \cup bd)e\left(\bar{i} \cup \bar{g}\bar{k} \cup \bar{j}\left(\overline{h \cup k}\right)\right)$ | 10 |
| | $a\bar{b}\bar{c}d\bar{e}f(g \cup \bar{g}(h \cup i))\left(\overline{jk \cup \bar{j}l \cup m}\right)$ | 13 |
| | $a\bar{b}\bar{c}(f(g \cup \bar{g}(h \cup i))(\overline{en} \cup d)) \cup \overline{n}\left(jk \cup \bar{j}l\overline{m}\right)$ | 14 |
| Large cause–effect graph | $\overline{(\overline{a} \cup d)(e \cup h \cup k \cup o)}\left(\overline{b} \cup a\right)(\overline{c} \cup a)\left(\overline{d} \cup c\right)\left(\overline{d} \cup b\right)(\overline{r} \cup a)\left(\overline{\overline{q} \cup h}\right)(\overline{p} \cup o)$ | 11 |

The results of the same operation are shown in Table 7.

**Table 7.** Experimental results of different systems.

| Project | Greedy Algorithms | | Heuristic Algorithms | | | Algebraic Methods | | Proposed |
|---|---|---|---|---|---|---|---|---|
| | IPOG | PICT | ABCS | GSTG | TMWOA | TConfig | Jenny | |
| RCCS | 6 | 5 | 5 | 5 | 6 | 6 | 6 | 5 |
| TCAS II | 8 | 9 | 8 | 8 | 8 | 9 | 8 | 6 |
| | 10 | 9 | 8 | 8 | 8 | 9 | 10 | 6 |
| | 10 | 9 | 8 | 8 | 9 | 9 | 10 | 6 |
| | 10 | 10 | 9 | 9 | 9 | 9 | 10 | 7 |
| Large cause–effect graph | 10 | 9 | 9 | 8 | 9 | 9 | 9 | 7 |

Overall, our method achieved better results.

### *4.3. Experiment on the Effect of the Improved PSO Algorithm*

#### 4.3.1. Comparison with Basic Particle Swarm Algorithm

To further illustrate the advantages of the improved PSO algorithm, experiments were conducted using the basic PSO algorithm and the improved PSO algorithm in the case of a larger number of inputs (large-scale system), and the number of iterations and the time taken to generate the test cases were recorded.

The parameters of the basic particle swarm algorithm used in this study were set as follows: The number of particles was set to 100. Because the coverage of the initially generated test cases into pairs of tests was already high, $c_1$ and $c_2$ were set to 1.2, the $\omega_{max}$ was set to 0.6, the $\omega_{min}$ was set to 0.4, and the maximum number of iterations ($N_g$) was set to 8000. The weight ($\omega$) was updated via Equation (14):

$$\omega = \omega_{max} - \frac{(\omega_{max} - \omega_{min}) * iter}{ger} \tag{14}$$

where *iter* is the number of current iterations.

The parameters of the improved particle swarm algorithm used in this paper were set as follows: the number of particles was set to 100, $c_1$ and $c_2$ were set to 1.2, the initial value of the $\omega$ was 0.8, and the maximum number of iterations ($N_g$) was 8000. The weight ($\omega$) was updated via Equation (11), and $IR_1$ was taken as 0.001 and $IR_2$ was taken as 0.002.

The test was performed 20 times. The results obtained in each execution are shown in Table 8, and a statistical analysis of the results in Table 8 is shown in Table 9, where NA indicates that the search results exceeded the maximum number of iterations.

**Table 8.** The numbers of iterations needed by the basic and improved PSO algorithms to reach the goal for different numbers of inputs. NA indicates that the corresponding search requires more than the maximum number of iterations.

| Number | $k$ = 10 | | $k$ = 15 | | $k$ = 20 | | $k$ = 25 | |
|---|---|---|---|---|---|---|---|---|
| | PSO | Proposed | PSO | Proposed | PSO | Proposed | PSO | Proposed |
| 1 | 3042 | 31 | 4986 | 61 | 5406 | 25 | 7290 | 100 |
| 2 | 3564 | 25 | 5323 | 44 | 5700 | 38 | 7244 | 147 |
| 3 | 3726 | 35 | 4684 | 43 | 5692 | 26 | 7023 | 208 |
| 4 | 3584 | 26 | 5379 | 56 | 5890 | 58 | 7210 | 82 |
| 5 | 3308 | 49 | 5875 | 52 | 5896 | 32 | 7681 | 136 |
| 6 | 3646 | 35 | 5691 | 69 | 5748 | 43 | 7131 | 72 |
| 7 | 4321 | 44 | 6038 | 84 | 5810 | 47 | 6683 | 124 |
| 8 | 3403 | 46 | 5214 | 108 | 6208 | 31 | NA | 120 |
| 9 | 3687 | 28 | 5357 | 85 | 5892 | 38 | 7286 | 115 |
| 10 | 3626 | 22 | 5161 | 65 | 6120 | 34 | 7482 | 93 |
| 11 | 3689 | 30 | 5470 | 38 | 6158 | 43 | 7666 | 87 |
| 12 | 4244 | 35 | 4888 | 71 | 5947 | 37 | 7548 | 144 |
| 13 | 4205 | 44 | 5655 | 77 | 5888 | 53 | 7227 | 117 |
| 14 | 3545 | 40 | 5606 | 65 | 6001 | 44 | 6960 | 87 |
| 15 | 3596 | 27 | 5550 | NA | 6135 | 37 | 7061 | 114 |
| 16 | 3606 | 39 | 4785 | 81 | 5543 | 30 | NA | 126 |
| 17 | 3466 | 44 | 5908 | 64 | 5984 | 42 | 6932 | 73 |
| 18 | 3337 | 41 | 5258 | 51 | 6157 | 40 | 7876 | 67 |
| 19 | 3374 | 33 | 5443 | 43 | 6166 | 50 | 7833 | 209 |
| 20 | 3904 | 30 | 5579 | 38 | 5949 | 41 | 7363 | 155 |

From Tables 8 and 9, it can be seen that the proposed improved PSO algorithm significantly reduces the number of iterations needed to generate test cases when compared with the basic PSO algorithm, thereby reducing the amount of time required to generate test cases. In general, as the number of inputs increases, the number of iterations increases. The improved PSO algorithm achieves a better performance in terms of the mean, median, standard deviation, and maximum and minimum numbers of iterations. However, the improved PSO still occasionally falls into a local optimum, such as the case in which $k = 25$.

**Table 9.** Statistical results of the numbers of iterations needed by the basic and improved PSO algorithms to reach the goal with different numbers of inputs. NA indicates that the corresponding search requires more than the maximum number of iterations.

| Name | $k$ = 10 | | $k$ = 15 | | $k$ = 20 | | $k$ = 25 | |
|---|---|---|---|---|---|---|---|---|
| | PSO | Proposed | PSO | Proposed | PSO | Proposed | PSO | Proposed |
| Average | 3644 | 35 | 5393 | 63 | 5915 | 39 | 7305 | 119 |
| Median | 3473 | 31 | 5283 | 50 | 5678 | 33 | 7327 | 128 |
| Standard Deviation | 431 | 0.5 | 296.5 | 11.5 | 271.5 | 8 | 36.5 | 27.5 |
| Maximum | 4321 | 49 | 6038 | NA | 6208 | 58 | NA | 209 |
| Minimum | 3042 | 22 | 4684 | 38 | 5406 | 25 | 6683 | 67 |

Figures 9 and 10 illustrate the convergence process of the basic and improved PSO when $k = 25$. It is evident that the initial coverage rates for both algorithms are already high, at values around 0.95. It is evident from Figures 9 and 10 that the number of iterations needed for convergence can be significantly reduced via the proposed improved PSO. This suggests that updating the weight ($\omega$) based on the improvement rates can significantly speed up the convergence process. In addition, Figure 10 suggests that the improved PSO algorithm has a faster pre-search ability, and its curve of convergence is convex to the upper left corner, indicating that a higher coverage rate can be achieved with fewer iteration steps.

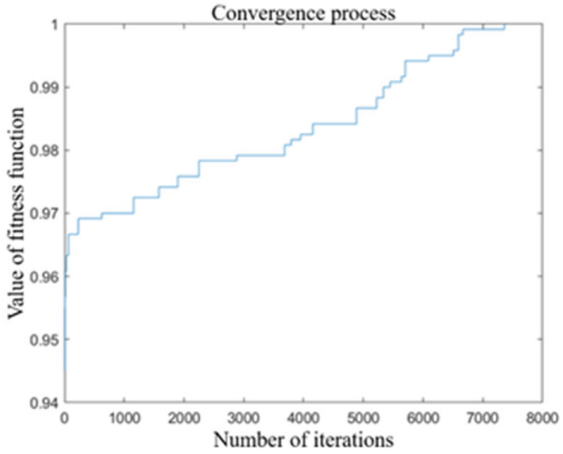



**Figure 9.** The convergence process of the basic PSO algorithm when $k = 25$.

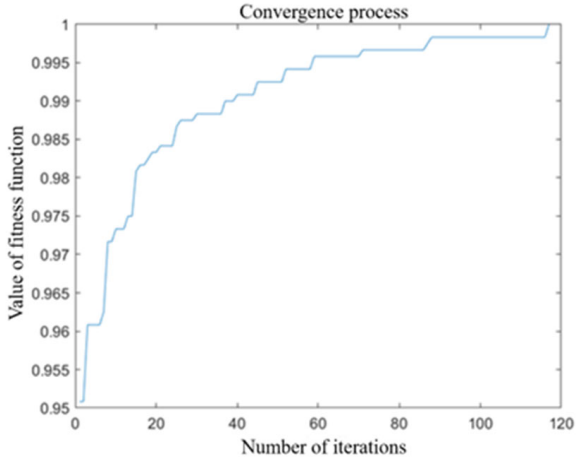



**Figure 10.** The convergence process of the proposed improved PSO at $k = 25$.

4.3.2. Comparison with Other Swarm Intelligence Algorithms

The proposed algorithm was compared with other classical swarm intelligence algorithms, including the Artificial Bee Colony (ABC) Algorithm [43], Firefly Algorithm (FA) [44], Grey Wolf Optimizer (GWO) [45], and Whale Optimization Algorithm (WOA) [46].

The evaluation shown in Figure 11 is used for different swarm intelligence algorithms. For an algorithm, R rounds of experiments are performed, where the maximum number of iterations for each experiment is T, and bf is the maximum fitness function value obtained with T iterations. The global best fitness values (last column) for each round of experiments can be analyzed by calculating their mean, variance, and other statistical metrics, as well as by plotting box-and-line plots. In this paper, a larger mean value, a fitness closer to 1 after the same number of iterations, and a higher combinatorial coverage of pairwise tests suggest the better performance for the corresponding algorithm. The mean value in each column is the mean value of the fitness at the current number of iterations in the R-round experiments, and the average convergence curve of the algorithm can be plotted. The convergence process of each tested algorithm can thus be analyzed.

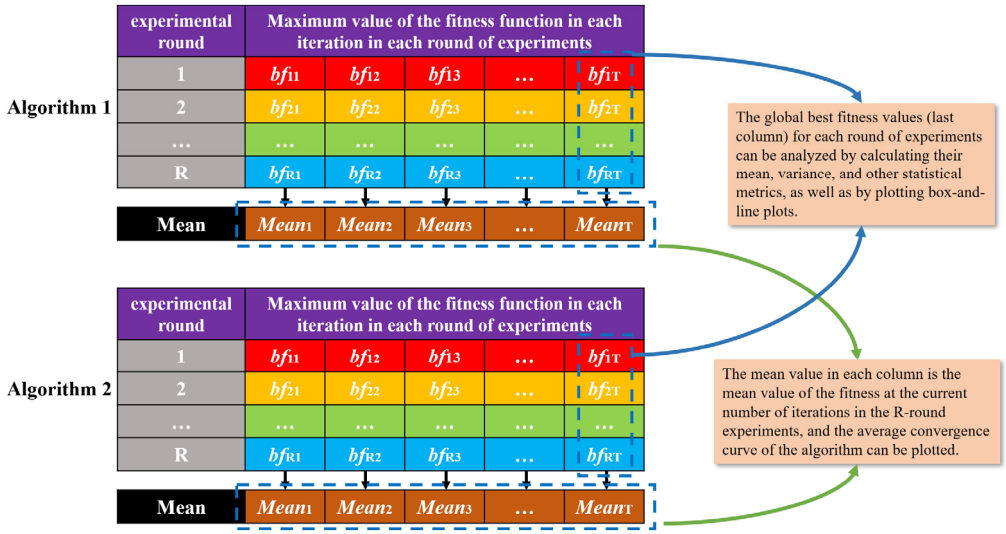



**Figure 11.** The evaluation of different swarm intelligence algorithms.

In this section, the number of experimental rounds was set to 20, the number of populations in all populations was set to 100, and the maximum number of iterations (T) was set to 2000. The parameters of each algorithm were as follows: ① In the ABC algorithm, the numbers of both the employed foragers bees and following bees were 50, and the following bees were generated via the roulette algorithm. If a nectar source was exploited 75 times, it was discarded and the hired bees corresponding to it became scout bees and a new nectar source was generated; ② The step coefficient $\alpha = 1$, the maximum attraction $\beta 0 = 0.8$, and the light absorption intensity coefficient $\gamma = 2$ were set in the FA; ③ There were no parameters that needed to be set in the GWO and WOA algorithms. ④ The parameterization of the method in this paper was the same as in Section 4.3.1. Experiments were performed with input variables ($k$) of 5 and 10.

Figure 12 shows the average convergence curves for all the algorithms for the case $K = 5$. The fitness functions of all the algorithms eventually converged quickly to 1, indicating that all the algorithms achieved 100% combinatorial coverage of the pairwise tests in 20 rounds of experiments. $k = 5$ is the relatively simple case. (All algorithms can reach the desired goal within 20 iterations, and for the sake of a more concise figure, only the first 20 iterations were plotted when making the graph).

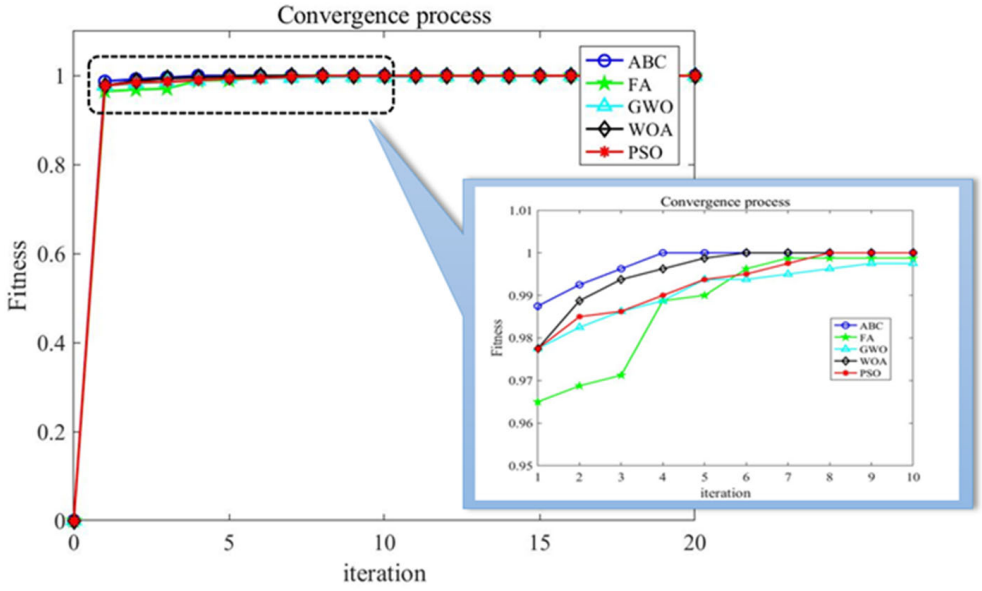



**Figure 12.** Average convergence curves for all algorithms for the case $K = 5$.

As can be seen from Figures 13 and 14, the improved PSO in this paper can have a very fast convergence rate in the early part of the iteration and can find the final objective smoothly. This fact suggests that the proposed approach is less likely to fall into a local optimal solution.

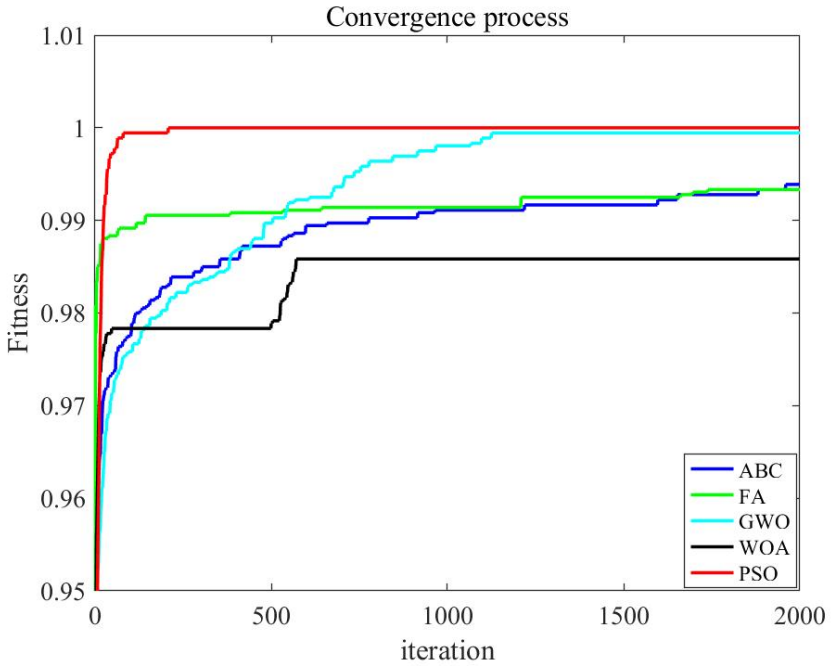



**Figure 13.** Average convergence curves for all algorithms for the case $K = 10$.

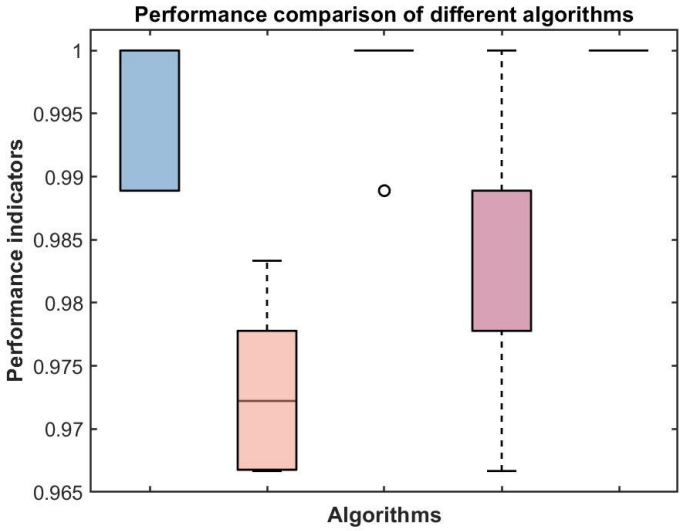



**Figure 14.** Comparison of box-and-line diagrams for all algorithms for the case $K = 10$; from left to right, the statistical results obtained for ABC, FA, GWO, WOA and PSO are shown in different colors. A circle represents an outlier in the testing results of an approach, and the top and bottom of a box represent the upper and lower quartiles of the testing results for the corresponding approach.

## 5. Conclusions

This paper considers the automatic generation of test cases for software testing and addresses the problem that some of the causes in a cause–effect graph may have no effect on the results when the test cases are generated. ROBDDs are utilized to simplify the graph such that the numbers of both the inputs and test cases can be reduced. For the case where the basic PSO algorithm is slow in generating test cases at large-scale input, the improved PSO algorithm proposed in this paper can significantly reduce the amount of time needed for test case generation. In addition, because the structures of many optimization problems in traffic control are similar to that of the test case generation problem, the methodology proposed in this paper can possibly be further generalized and applied to traffic control problems [47].

Because PSO algorithms have weaknesses, such as being trapped into the local optimum and slow convergence in the later stage, in future work, strategies that can lead to improvements in these aspects need to be developed to further improve the computational efficiency for the generation of test cases. In the current research on combinatorial testing techniques, most of the research work focuses on proposing novel and effective algorithms that can quickly construct smaller sets of combinatorial test cases. However, a number of important problems remain unsolved and need further in-depth research, including research on methods that can abstract input instances from actual commercial software products, techniques that can extend the combinatorial test so that it can be better applied to software testing, methods that can effectively combine the combinatorial test with other software-testing techniques, and approaches that can apply the combinatorial test technology to more testing scenarios.

**Author Contributions:** Methodology, S.L.; validation, S.L.; writing, S.L.; formal analysis, Y.S.; conceptualization, Y.S.; software, Y.S.; supervision, Y.S.; writing—review and editing, Y.S.; visualization, Y.Z.; data curation, Y.Z.; funding acquisition, Y.Z. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research was funded by the Innovative Research Foundation of Ship General Performance, grant number 25422208.

**Data Availability Statement:** The data related to the research reported in this paper are freely available upon request.

**Conflicts of Interest:** The authors declare no conflicts of interest.

## References


1. Jamil, M.A.; Arif, M.; Abubakar, N.S.A.; Ahmad, A. Software Testing Techniques: A Literature Review. In Proceedings of the 2016 6th International Conference on Information and Communication Technology for the Muslim World (ICT4M), Jakarta, Indonesia, 22–24 November 2016; pp. 177–182. [CrossRef]
2. Bashir, M.B.; Nadeem, A. An Experimental Tool for Search-Based Mutation Testing. In Proceedings of the 2018 International Conference on Frontiers of Information Technology (FIT), Islamabad, Pakistan, 17–19 December 2018; pp. 30–34. [CrossRef]
3. Varshney, S.; Mehrotra, M. Automated Software Test Data Generation for Data Flow Dependencies using Genetic Algorithm. *Int. J. Adv. Res. Comput. Sci. Softw. Eng.* **2014**, *4*, 472–479.
4. Paradkar, A.; Tai, K.; Vouk, M. Specification-based testing using cause-effect graphs. *Ann. Softw. Eng.* **1997**, *4*, 133–157. [CrossRef]
5. Sziray, J. Evaluation of boolean graphs in software testing. In Proceedings of the 2013 IEEE 9th International Conference on Computational Cybernetics (ICCC), Tihany, Hungary, 8–10 July 2013; pp. 225–230. [CrossRef]
6. Ufuktepe, K.D.; Ayav, T.; Belli, F. Test input generation from cause–effect graphs. *Softw. Qual. J.* **2021**, *29*, 733–782. [CrossRef]
7. Hua, A.I.; Song, X.; Heng, A. Method of Simple-cyclic-reduce Generating Three-way Combinatorial Testcase Set. *Comput. Eng. Des.* **2018**, *39*, 3728–3733. [CrossRef]
8. Sun, W.; Jiang, J.; Nie, C. Configurable Hybrid Algorithm for Combinatorial Test Suit Generation. *Comput. Sci.* **2011**, *38*, 130–135+160.
9. Kuhn, D.R.; Reilly, M.J. An Investigation of the Applicability of Design of Experiments to Software Testing. In Proceedings of the 27th Annual NASA Goddard/IEEE Software Engineering Workshop, Greenbelt, MD, USA, 5–6 December 2002; pp. 91–95. [CrossRef]
10. Caponetto, R.; Fortuna, L.; Fazzino, S.; Xibilia, M.G. Chaotic sequences to improve the performance of evolutionary algorithms. *IEEE Trans. Evol. Comput.* **2003**, *7*, 289–304. [CrossRef]
11. Demir; Burak, F.; Tuncer, T.; Kocamaz, A.F. A chaotic optimization method based on logistic-sine map for numerical function optimization. *Neural Comput. Appl.* **2020**, *32*, 14227–14239. [CrossRef]
12. Zhao, X.; Fang, Y.; Liu, L.; Xu, M.; Li, Q. A covariance-based Moth–flame optimization algorithm with Cauchy mutation for solving numerical optimization problems. *Appl. Soft Comput.* **2022**, *119*, 108538. [CrossRef]
13. Mao, Q.; Zhang, Q. Improved sparrow algorithm combining Cauchy mutation and opposition-based learning. *J. Front. Comput. Sci. Technol.* **2021**, *15*, 1155–1164.
14. Mao, C.; Yu, X.; Xue, Y. Algorithm Design and Empirical Analysis for Particle Swarm Optimization-Based Test Data Generation. *J. Comput. Res. Dev.* **2014**, *51*, 824–837.
15. Lee, C.C.; Friedman, J. Requirements Modeling and Automated Requirements-Based Test Generation. *SAE Int. J. Aerosp.* **2013**, *6*, 607–615. [CrossRef]
16. Son, H.S.; Kim, R.Y.C.; Park, Y.B. Test Case Generation from Cause-Effect Graph Based on Model Transformation. In Proceedings of the 2014 International Conference on Information Science & Applications (ICISA), Seoul, Republic of Korea, 6–9 May 2014; pp. 1–4. [CrossRef]
17. Chung, I. CEGPairGen: An automated tool for generating pairwise tests from cause-effect graphs. *Int. J. Softw. Eng. Its Appl.* **2015**, *9*, 53–66.
18. Akram, K.; Rafe, V. An Optimal Solution for Test Case Generation Using ROBDD Graph and PSO Algorithm. *Qual. Reliab. Eng. Int.* **2016**, *32*, 2263–2279.
19. Woo, J.; Young, K. Automatic Generation Mechanism of Cause-Effect Graph with Informal Requirement Specification Based on the Korean Language. *Appl. Sci.* **2021**, *11*, 11775. [CrossRef]
20. Huang, M.; Zhang, C.; Liang, X. Software Test Cases Generation Based on Improved Particle Swarm Optimization. In Proceedings of the 2nd International Conference on Information Technology and Electronic Commerce, Dalian, China, 20–21 December 2014; pp. 52–55. [CrossRef]
21. Sheng, Y.; Wei, C.; Jiang, S. Constraint Test Cases Generation Based on Particle Swarm Optimization. *Int. J. Reliab. Qual. Saf. Eng.* **2017**, *24*, 1750021. [CrossRef]
22. Prasad, M.L.; Sastry, J.K.R.; Mallikarjuna, B.; Sitaramulu, V.; Srinivasulu, C.; Naib, B.B. Development of a Programmed Generation of T-Way Test cases Using an Improved Particle Swarm Optimization Strategy. In Proceedings of the 2022 2nd International Conference on Advance Computing and Innovative Technologies in Engineering (ICACITE), Greater Noida, India, 28–29 April 2022; pp. 1394–1399. [CrossRef]
23. Wang, K.; Zhu, Y.; Li, G.; Wang, J.; Liu, Z. Test Case Generation Method Based on Particle Swarm Optimization Algorithm. In Proceedings of the SPIE 12721, Second International Symposium on Computer Applications and Information Systems (ISCAIS 2023), Hangzhou, China, 26 June 2023. [CrossRef]
24. Lipsky, A.M.; Greenland, S. Causal Directed Acyclic Graphs. *JAMA* **2022**, *327*, 1083–1084. [CrossRef]
25. Law, G.R.; Green, R.; Ellison, G.T.H. Confounding and Causal Path Diagrams. In *Modern Methods for Epidemiology*; Tu, Y.K., Greenwood, D., Eds.; Springer: Dordrecht, The Netherlands, 2012. [CrossRef]
26. Cohen, D.M.; Dalal, S.R.; Fredman, M.L.; Patton, G.C. The AETG system: An approach to testing based on combinatorial design. *IEEE Trans. Softw. Eng.* **1997**, *23*, 437–444. [CrossRef]

27. Page, A.; Johnston, K.; Rollison, B. Functional Testing Techniques. In *How We Test Software at Microsoft*; Microsoft Press: Unterschleissheim, Germany, 2009; pp. 100–113.
28. Cohen, D.M.; Dalal, S.R.; Parelius, J.; Patton, G.C. The Combinatorial Design Approach to Automatic Test Generation. *IEEE Softw.* **1996**, *13*, 83–88. [CrossRef]
29. Kendall, D.; Rényi, A. *Foundations of Probability*; Cambridge University Press: Cambridge, UK, 1971.
30. Katona, G.O.H. Two applications (for search theory and truth functions) of Sperner type theorems. *Period. Math. Hung.* **1973**, *3*, 19–26. [CrossRef]
31. Kleitman, D.J.; Spencer, J. Families of k-independent sets. *Discret. Math.* **1973**, *6*, 255–262. [CrossRef]
32. Choi, S.; Kim, H.K.; Oh, D.Y. Structures and lower bounds for binary covering arrays. *Discret. Math.* **2011**, *312*, 2958–2968. [CrossRef]
33. Jenkins, B. Jenny Test Tool, Online Tool. 2016. Available online: https://burtleburtle.net/bob/math/jenny.html (accessed on 8 December 2023).
34. Lei, Y.; Kacker, R.; Kuhn, D.R.; Okun, V.; Lawrence, J. IPOG: A General Strategy for T-Way Software Testing. In Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07), Tucson, AZ, USA, 26–29 March 2007; pp. 549–556. [CrossRef]
35. Czerwonka, J. Pairwise Testing in Real World Practical Extensions to Test Case Generators. In Proceedings of the 24th Annual Pacific Northwest Software Quality Conference, Portland, OR, USA, 9–11 October 2006.
36. Alazzawi, A.K.; Md Rais, H.; Basri, S.B. Artificial Bee Colony Algorithm for t-Way Test Suite Generation. In Proceedings of the 2018 4th International Conference on Computer and Information Sciences (ICCOINS), Kuala Lumpur, Malaysia, 13–14 August 2018; pp. 1–6.
37. Htay, K.M.; Othman, R.R.; Amir, A.; Alkanaani, J.M. Gravitational search algorithm based strategy for combinatorial t-way test suite generation. *J. King Saud Univ. Comput. Inf. Sci.* **2021**, *34*, 4860–4873.
38. Hassan, A.; Abdullah, S.; Zamli, K.; Razali, R. Whale Optimization Algorithm Strategies for Higher Interaction Strength T-Way Testing. *Comput. Mater. Contin.* **2022**, *73*, 2057–2077. [CrossRef]
39. Medikonda, B.S.; Ramaiah, P.S.; Gokhale, A.A. FMEA and Fault Tree based Software Safety Analysis of a Railroad Crossing Critical System. *Glob. J. Comput. Sci. Technol.* **2011**, *11*, 57–62.
40. Krupalija, E.; Cogo, E.; Bećirović, Š.; Prazina, I.; Hodižić, K.; Bešić, I. Forward-Propagation Approach for Generating Feasible and Minimum Test Case Suites From Cause-Effect Graph Specifications. *IEEE Access* **2022**, *10*, 124545–124562. [CrossRef]
41. Zamli, K.Z.; Mohammad, F.J.K.; Mohammed, I.Y.; Nor Ashidi, M.I.; Rusli, A. Design and implementation of a t-way test data generation strategy with automated execution tool support. *Inf. Sci.* **2011**, *181*, 1741–1758. [CrossRef]
42. Krupalija, E.; Šeila, B.; Irfan, P.; Emir, C.; Ingmar, B. New Graphical Software Tool for Creating Cause-Effect Graph Specifications. *J. Commun. Softw. Syst.* **2022**, *18*, 311–322. [CrossRef]
43. Karaboğa, D. *An Idea Based on Honey Bee Swarm for Numerical Optimization*; Erciyes University: Kayseri, Türkiye, 2005; Volume 200, pp. 1–10.
44. Yang, X.-S. Firefly Algorithms for Multimodal Optimization. In *Symposium on Stochastic Algorithms: Foundations and Applications*; Springer: Berlin/Heidelberg, Germany, 2009.
45. Mirjalili, S.; Mirjalili, S.M.; Lewis, A. Grey Wolf Optimizer. *Adv. Eng. Softw.* **2014**, *69*, 46–61. [CrossRef]
46. Mirjalili, S.M.; Andrew, L. The Whale Optimization Algorithm. *Adv. Eng. Softw.* **2016**, *95*, 51–67. [CrossRef]
47. Scellato, S.; Fortuna, L.; Frasca, M.; Gómez-Gardeñes, J.; Latora, V. Traffic optimization in transport networks based on local routing. *Eur. Phys. J. B* **2010**, *73*, 303–308. [CrossRef]


**Disclaimer/Publisher's Note:** The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.