Online-Journals.org

iJIM International Journal of Interactive Mobile Technologies

iJIM | eISSN: 1865-7923 | Vol. 17 No. 20 (2023) | OPEN ACCESS

https://doi.org/10.3991/ijim.v17i20.37517

PAPER

# Improved Path Testing Using Multi-Verse Optimization Algorithm and the Integration of Test Path Distance

Hussam N. Fakhouri[1](✉), Ahmad K. Al Hwaitat[2], Mohammad Ryalat[3], Faten Hamad[2,4], Jamal Zraqou[1], Adi Maaita[5], Mohannad Alkalaileh[6], Najem N. Sirhan[1]

[1]Faculty of Information Technology, University of Petra, Amman, Jordan

[2]The University of Jordan, Amman, Jordan

[3]Al-Balqa Applied University, Salt, Jordan

[4]Sultan Qaboos University, Sultanate of Oman, al-Seeb, Oman

[5]Middle East University, Amman, Jordan

[6]Al Ain University, Al Ain, United Arab Emirates

hussam.fakhouri@uop.edu.jo

**ABSTRACT**

Emerging technologies in artificial intelligence (AI) and advanced optimization methodologies have opened up a new frontier in the field of software engineering. Among these methodologies, optimization algorithms such as the multi-verse optimizer (MVO) provide a compelling and structured technique for identifying software vulnerabilities, thereby enhancing software robustness and reliability. This research investigates the feasibility and efficacy of applying an augmented version of this technique, known as the test path distance multi-verse optimization (TPDMVO) algorithm, for comprehensive path coverage testing, which is an indispensable aspect of software validation. The algorithm's versatility and robustness are examined through its application to a wide range of case studies with varying degrees of complexity. These case studies include rudimentary functions like maximum and middle value extraction, as well as more sophisticated data structures such as binary search trees and AVL trees. A notable innovation in this research is the introduction of a customized fitness function, carefully designed to guide TPDMVO towards traversing all possible execution paths in a program, thereby ensuring comprehensive coverage. To further enhance the comprehensiveness of the test, a metric called 'test path distance' (TPD) is utilized. This metric is designed to guide TPDMVO towards paths that have not been explored before. The findings confirm the superior performance of the TPDMVO algorithm, which achieves complete path coverage in all test scenarios. This demonstrates its robustness and adaptability in handling different program complexities.

**KEYWORDS**

artificial intelligence (AI), optimization, path testing, multiverse optimizer

## 1 INTRODUCTION

Artificial intelligence (AI) has emerged as a significant force in reshaping various sectors, enabling the creation of intelligent systems capable of learning, reasoning, problem-solving, perception, and language understanding [1]. In the field of

Fakhouri, H.N., Al Hwaitat, A.K., Ryalat, M., Hamad, F., Zraqou, J., Maaita, A., Alkalaileh, M., Sirhan, N.N. (2023). Improved Path Testing Using Multi-Verse Optimization Algorithm and the Integration of Test Path Distance. *International Journal of Interactive Mobile Technologies (iJIM)*, 17(20), pp. 38–59. https://doi.org/10.3991/ijim.v17i20.37517

Article submitted 2022-12-19. Revision uploaded 2023-07-21. Final acceptance 2023-08-23.

optimization problems, AI techniques provide a powerful set of tools, paving the way for efficient, intelligent, and adaptive optimization solutions [2]. One application area that has witnessed notable advancements due to AI's intervention is software testing. Software testing is a critical phase in the software development life cycle (SDLC) to ensure the functionality, reliability, and performance of the software [3]. However, manual testing processes can be labor-intensive, time-consuming, and prone to errors [4]. To address these challenges, researchers have explored AI-based techniques to automate and enhance software testing processes [5]. However, the production of high-quality and robust software applications has become a fundamental requirement across diverse fields, ranging from communication systems and finance to healthcare and entertainment [6]. Ensuring the reliability, efficiency, and correctness of these applications requires rigorous testing processes throughout the software development lifecycle [7] [8].

Software testing, which includes a variety of activities aimed at evaluating the attributes or capabilities of a program and ensuring that it achieves the desired results [6], serves multiple purposes. First, it verifies that the software performs as expected, delivering the intended functionality to the user. Second, it validates that the software operates correctly in all anticipated usage scenarios and system environments, thereby maintaining user trust and satisfaction. Third, it helps in identifying any defects or discrepancies in the software, enabling developers to rectify these issues before deployment. In software testing, path testing plays a crucial role as it uncovers the behavioral intricacies of the software by analyzing every possible execution path [9]. It ensures that all possible scenarios in a program are tested, which, in turn, reduces the likelihood of undiscovered bugs. However, due to the inherent complexity of path testing, creating a comprehensive test suite that ensures maximum path coverage is a challenging task. Furthermore, path testing provides valuable insights into the internal workings of a software program, including its structure and logic. This contributes to a deeper understanding of the software's functionality and, subsequently, improves its overall quality [10]. Given its pivotal role in enhancing software reliability and robustness, the importance of efficient and comprehensive path testing cannot be overstated in modern software engineering practices.

Path testing, a white-box testing technique, revolves around the principle of executing every feasible path within a software program at least once with the intention of uncovering potential bugs or inconsistencies [11]. This process involves identifying all possible paths that can be taken from the start to the end of a given function or method within the program code. The "paths" here refer to a distinct sequence of code statements or instructions that lead from entry to exit points in the software program [12]. This escalating complexity underscores the crucial necessity for efficient test data that can effectively navigate this expansive path universe [13].

However, the application of optimization algorithms can systematically and thoroughly explore the search space more effectively than random testing. Additionally, they can handle the complexity of large-scale software applications more effectively than symbolic execution. By integrating optimization algorithms into the process of test data generation, researchers and practitioners can potentially achieve better coverage and efficiency in path testing. This offers a promising direction to improve software testing practices in the era of increasingly complex software applications [14].

Emerging from the broad spectrum of optimization algorithms, the multi-verse optimizer (MVO) has gained significant recognition due to its excellent performance in solving a variety of complex optimization problems [15]. Rooted in the concepts

of cosmology, the MVO takes its inspiration from three astronomical phenomena: the white hole, black hole, and wormhole. Each of these phenomena plays a specific role within the algorithm. The white-hole concept aids in exploration by introducing diversity into the solutions, while the black hole concept supports exploitation by directing the search towards promising regions in the solution space. The wormhole phenomenon, on the other hand, enables local search within existing solutions [16]. The combination of these concepts allows the MVO to maintain a fine balance between exploration (searching new areas in the solution space) and exploitation (refining the solutions in the current areas), making it an effective tool for tackling optimization problems. Due to its balanced and robust approach to exploration and exploitation, the MVO algorithm has demonstrated remarkable efficiency and effectiveness in solving optimization problems. It has outperformed other popular algorithms on a variety of benchmark problems, highlighting its potential as a powerful optimization tool [15].

Given its proven capabilities, incorporating the MVO into the process of test data generation for path testing holds significant potential to bring about substantial improvements in this critical aspect of software testing. By utilizing the algorithm's robust capabilities in navigating through complex search spaces, the process of generating test data can be optimized to achieve better path coverage, efficiency, and reduced redundancy [16]. The MVO algorithm's exploration and exploitation capabilities can be utilized to generate test data that covers a wide range of paths in the software, including complex paths that may be overlooked by traditional methods. The local search mechanism of MVO can further enhance the quality of test data by fine-tuning it to uncover intricate bugs or inconsistencies in the software. The proposed use of TPD in MVO not only enhances the efficiency of the test data generation process but also potentially improves the overall quality of the software applications being tested. It represents a significant advancement in the journey towards creating more dependable and resilient software systems, which are essential in our ever-growing digital and interconnected world. This study utilizes the capabilities of AI and focuses on an optimization algorithm called TPD MVO, which draws inspiration from the multi-verse algorithm.

## 1.1 Problem statement

The software testing process, especially path testing, is essential for producing robust and reliable software applications. Despite its importance, generating suitable test data for path testing remains challenging due to the complexity of modern software. Traditional test data generation methods, such as random testing and symbolic execution, have limitations. Random testing does not guarantee comprehensive coverage due to its unpredictability, while symbolic execution can become time-intensive because of the state explosion problem associated with complex software. Manual methods are also impractical for large projects due to the time, expertise, and risk of errors involved. There is, therefore, a need for automated and efficient techniques for generating data.

Optimization algorithms, specifically the MVO, offer potential solutions. MVO, which draws inspiration from cosmological phenomena such as black holes and wormholes, has demonstrated its effectiveness in various optimization scenarios. However, its application in test data generation, particularly for path testing, has not been thoroughly investigated. The primary objective of the research is to develop and investigate the TPDMVO algorithm for generating test data in path testing.

### 1.2 Research importance

The significance of this research lies in its potential to greatly improve the process of generating test data in path testing, which is a crucial aspect of software testing. By innovatively adapting the TPDMVO algorithm for this purpose, the research addresses the limitations of traditional test data generation methods and offers a unique, potentially more effective and efficient solution. As software applications become increasingly complex, ensuring thorough path testing becomes a vital necessity for maintaining high software quality and reliability. Improving the generation of test data, as proposed in this research, directly contributes to achieving this goal. Additionally, incorporating a TPD measure into the TPDMVO algorithm represents a unique approach that could further enhance the quality of generated test data, thereby increasing the effectiveness of path testing. Therefore, this research holds significant implications for software developers, testers, and, ultimately, end-users who stand to benefit from more robust and reliable software applications.

### 1.3 Research contribution

The research envisions two primary contributions to the field of software testing. First, it introduces an innovative adaptation of the TPDMVO algorithm, specifically customized for the task of generating test data in path testing. This novel application of the TPDMVO algorithm signifies a significant advancement in the utilization of optimization algorithms in software testing, particularly in path testing. By capitalizing on TPDMVO's potent capabilities in exploration, exploitation, and local search, this study opens up a new avenue for enhancing the efficiency and effectiveness of test data generation. Second, the research introduces a novel measure of TPD, specifically designed to be incorporated into the modified TPDMVO algorithm. This measure is designed to improve the quality of the generated test data, which could potentially increase the success rate of path testing. The development of this distance measure may offer valuable insights and guidelines for similar efforts aimed at enhancing the utilization of optimization algorithms in test data generation.

## 2 LITERATURE REVIEW AND RELATED WORK

In this section, there is an in-depth examination of previous work related to the application of the MVO in test data generation for path testing. An extensive review of the relevant literature helps to highlight the significance of the problem at hand, place the proposed solution within the broader academic discourse, and identify gaps that this research aims to address. We will examine research in three main areas: software testing with an emphasis on path testing, traditional methods of test data generation, and the application of optimization algorithms, particularly the MVO, in software testing.

### 2.1 Multi-verse optimizer

The MVO algorithm is a global optimization algorithm inspired by concepts from cosmology, namely black hole, white hole, and wormhole phenomena [16]. The design of the algorithm leverages the principles of these phenomena, offering

powerful capabilities for exploration, exploitation, and local search. This makes it a versatile tool for optimization tasks. The underlying premise of MVO is the idea that universes (potential solutions) undergo three types of operations: the White Hole, Black Hole, and Wormhole. In the Black Hole operation, universes are pulled towards the fittest universe, simulating the gravitational pull of a black hole. In the White Hole operation, new universes are generated and randomly placed in the search space, simulating the hypothetical phenomenon where matter and energy are ejected into our universe from elsewhere [16]. In MVO, each universe represents a potential solution to the optimization problem. The quality or fitness of each universe is determined by an objective function that is relevant to the problem. Universes gravitate towards the most optimal universe based on their fitness. This movement mirrors the gravitational attraction exerted by black holes. The closer a universe is to the optimal solution, the more it influences other universes to coverage towards it. This phenomenon simulates the theoretical process in cosmology, where matter and energy are projected into our universe from external sources. Benefiting from randomness, this operation allows universes to "teleport" within the search domain, aiding in the exploration of the solution space and avoiding being trapped in local optima [16].

## 2.2 Test data generation techniques

The process of test data generation has been central to the field of software testing, with a variety of methods and techniques developed over the years. These techniques, which can be broadly divided into static and dynamic methods, each bring their own unique set of benefits and challenges. Static techniques are those in which test data is manually created by software testers [17]. In these methods, testers generate test cases based on their understanding of the software's requirements and structure. The testing process often relies on techniques such as equivalence partitioning, boundary value analysis, and decision table testing. The significant advantage of these techniques lies in the tester's ability to create highly customized tests tailored to specific software features. However, they tend to be time-consuming, may not scale well for large applications, and are subject to human error due to the extensive manual input required [18]. Transitioning from manual to automated test data generation, we encounter dynamic techniques, where test data is automatically generated during software execution. These techniques are further classified into three types: random, goal-oriented (systematic), and model-based testing. Random testing is a straightforward technique where random test data is generated within the input domain of the software being tested. The simplicity and low cost of this method make it appealing [19]. However, as software complexity grows, the probability of encountering important corner cases or triggering complex behavior solely through random inputs decreases, thereby limiting its effectiveness [20].

Goal-oriented or systematic testing, on the other hand, aims to generate test data that meets specific testing criteria. Techniques such as symbolic execution and search-based software testing fall under this category [21]. Their goal often revolves around maximizing code coverage or achieving other predefined objectives. These techniques can be powerful, but they also require complex computations. For instance, symbolic execution can lead to the problem of "path explosion," which limits its applicability for complex programs [22]. In model-based testing, test data generation is based on a model of the software being tested. The advantage of this approach is that it can provide comprehensive test coverage based on the model.

However, the need to create a model can make this method more time-consuming and complex than others [23]. In recent years, with the rise of modern computational techniques, there has been a growing interest in incorporating optimization algorithms in test data generation. Techniques such as genetic algorithms, particle swarm optimization, and ant colony optimization have shown promising results in various aspects of software testing [24].

### 2.3 Path testing

Path testing is a critical component of structural testing techniques. It focuses on identifying and examining all possible executable paths through a software program to uncover potential faults and improve software quality [25]. It holds a significant position in the spectrum of software testing techniques due to its potential for uncovering hidden bugs that other testing methods might overlook. The underlying principle of path testing involves deriving a set of test data that can activate different possible paths in the software under test [26]. A path is defined as a distinct sequence of code statements or instructions that are executed from the beginning to the end of a program or a program segment. As a testing method, path testing places a strong emphasis on how control structures, such as loops and conditionals, influence the flow of program execution [27].

However, the key challenge in path testing is its complexity in terms of combinations. A software program can have a significant number of potential execution paths, particularly those involving loops or recursion [28]. This potential infinity of paths, also known as the "path explosion" problem, makes it infeasible to exhaustively test every single path, especially in complex or large-scale software systems [29]. To address this problem, numerous techniques and strategies have been proposed. These techniques include independent path testing, cyclomatic complexity, basis path testing, loop testing, and data flow testing. Each technique has its own unique approach to selecting critical paths for testing, which helps to reduce the computational demand [30]. Even though these methods bring some relief to the problem of path explosion, the selection and generation of test data that can effectively and efficiently cover important paths remains a challenge [31].

## 3 TEST PATH DISTANCE MULTI-VERSE OPTIMIZER FOR PATH TESTING

This section presents the proposed enhancement of the MVO algorithm, which utilizes the TPD for generating test data in path testing.

### 3.1 Overview of enhancements

The proposed enhancement to the MVO involves integrating the concept of TPD into the algorithm's mechanics. The idea is to utilize the MVO's powerful optimization capabilities to minimize the TPD, which quantifies the "distance" between a programs' actual execution path and a target path. The incorporation of TPD in the MVO aims to guide the generation of test data more efficiently, with the goal of achieving path coverage. In essence, the proposed enhancements to the MVO take the form of adaptations in the representation of solutions, the fitness measure, and the algorithm's operations. Each solution or "universe" in the MVO is redefined to

correspond to a set of test data capable of executing a program. This adaptation allows the MVO to operate directly on test data, aligning TPDMVO with the requirements of path testing.

The fitness measure in the MVO, which is represented by the inflation rate in the cosmological context, has been replaced with the TPD. Thus, lower TPD values, which signify a closer match between the actual and target execution paths, translate to "better" solutions or universes with lower inflation rates. This change ensures that the MVO's optimization process is directed towards minimizing the TPD, thereby generating test data that leads to the desired execution paths in the software. Finally, the operations of the MVO—the creation, selection, and local search mechanisms based on the white hole, black hole, and wormhole phenomena, respectively—are adapted to work with this new representation of solutions and the TPD measure. These adaptations retain the robust search capabilities of the MVO while customizing them for the specific task of generating test data for path testing. The proposed enhancements to the MVO involve integrating the TPD concept into TPDMVO to guide the generation of test data for path testing. The ultimate goal is to improve the efficiency and effectiveness of this testing technique.

## 3.2 Description of test path distance

The concept of TPD plays a crucial role in the approach of this research to enhance the MVO for test data generation in path testing. TPD is a numerical measure that quantifies the "distance" between the actual execution path of a program with a given set of test data and a target execution path. By minimizing this distance, it becomes possible to guide a program's execution towards desired paths, thereby achieving more comprehensive and effective path coverage in testing. To provide a deeper understanding, TPD is computed based on the differences between the actual and target execution paths at each decision point in the program. Decision points refer to locations in a program where the control flow could diverge, such as conditional statements or loops. At each decision point, the outcome (e.g., the branch taken in a conditional statement) with the given test data is compared with the outcome on the target path using the given test data.

The "distance" for a single decision point is typically computed as a binary value: zero if the outcomes match (i.e., the test data leads to the same branch as the target path) or one if they do not match. However, more sophisticated distance measures could be employed for specific types of decision points, such as loops, where the distance could be influenced by the number of iterations. In such cases, the distance could be computed based on the difference in the number of iterations. After calculating the distances at each decision point, the TPD is obtained by summing these distances. This results in a single numerical value that measures the overall "distance" between the actual and target execution paths. The goal of the test data generation process is to find the test data that minimizes the TPD. This effectively guides the program's execution towards the target paths and improves path coverage in testing.

## 3.3 Mathematical formulation of the enhanced multi-verse optimizer

To mathematically express the TPDMVO incorporating the TPD, it is crucial to first define the key elements involved in the model.

Let's consider a universe, U, represented as a vector of size n, where each element corresponds to an input variable for the software being tested. Thus, U = [u1, u2, ..., un]. The MVO maintains a collection of universes that serve as candidate solutions for the problem of test data generation.

The TPD, denoted by D(U), is computed as the sum of distances at each decision point in the software's control flow graph, as described in Section 6.2. The distance at a decision point is computed based on the difference between the actual and target outcomes, using the test data represented by universe U.

Under the enhanced MVO, the objective is to find the universe U that minimizes the TPD. Thus, the optimization problem can be mathematically formulated as follows:

$$\text{Minimize } D(U) \tag{1}$$

Subject to $U \in S$, where S is the search space defined by the feasible range of each input variable.

The search process of the MVO involves creating new universes (candidate solutions) and updating existing ones based on the principles of white hole, black hole, and wormhole phenomena. These processes are adapted in the TPDMVO to work with the TPD and the new representation of universes as sets of test data.

## 3.4 Adaptation of wormhole mechanism

The wormhole mechanism is one of the critical components that distinguishes the MVO from other optimization algorithms. It plays an instrumental role in exploring the search space, improving solution diversity, and preventing TPDMVO from getting trapped in local optima. In the context of the TPDMVO for test data generation, the wormhole mechanism has been adapted to effectively work with the TPD and the new representation of universes as sets of test data. Conceptually, the wormhole mechanism represents the phenomenon of information exchange among universes through a high-dimensional tunnel, enabling a universe to move to a completely different location in the search space. This helps TPDMVO to explore the search space more effectively and escape from local optima.

In the original MVO, the wormhole mechanism is realized by occasionally replacing a universe with a completely new and randomly generated one. In the enhanced MVO, a more sophisticated approach is used to ensure that the new universes generated by the wormhole mechanism are not only random but also directed towards promising areas of the search space. Specifically, when the wormhole mechanism is triggered for a universe U, a new universe U' is created, taking into account the target path in the software's control flow graph. Each decision point in the control flow graph corresponds to a decision variable in U', and the value of this variable is selected to direct the software's execution towards the target path. The specific method for selecting the values of the decision variables in U' can vary depending on the nature of the decision points and the structure of the control flow graph. However, the common objective is to minimize the TPD, which means guiding the software's execution towards the target path. Furthermore, the incorporation of the wormhole mechanism in the TPDMVO is intended to preserve the variety of solutions and enhance the algorithm's ability to explore. Simultaneously, it directs the search towards promising regions of the search space, thereby increasing the efficiency and effectiveness of the test data generation process.

### 3.5 Anticipated improvements in test data generation

The proposed enhancements for the MVO are specifically designed to improve test data generation in path testing. This enhanced MVO, now incorporating the TPD as its guiding metric, holds the potential for significant improvements over conventional methods. One of the main anticipated improvements is the efficiency of test data generation. The guided nature of the enhanced MVO, driven by the objective of minimizing the TPD, should allow TPDMVO to find effective test data in fewer iterations than random or purely exploratory methods. This not only reduces the computational resources required for test data generation but also shortens the testing cycle, thereby facilitating faster software development processes.

The second anticipated improvement relates to the quality of the generated test data. Quality, in this context, refers to the test data's ability to uncover faults in the software. By guiding the search towards paths that are yet to be executed, the TPDMVO is expected to generate test data that lead to more thorough path coverage. This improved path coverage can, in turn, lead to more fault detections, thereby elevating the reliability of the software under test. The incorporation of the TPD in the TPDMVO also means that the generated test data will be more targeted, potentially focusing on complex or rarely executed paths that could harbor hidden faults. This targeted approach could provide an additional boost to the quality of the testing process. Lastly, the utilization of the MVO algorithm, with its efficient equilibrium between exploration and exploitation, can assist in handling intricate software applications that possess extensive input spaces and complex control flow graphs. The ability to handle such complexity is becoming increasingly important as software systems continue to grow in size and complexity.

### 3.6 Test path distance (TPD)

Test path distance is a sophisticated metric for guiding the generation of test data in path testing. It quantifies the proximity of a program execution's path to a specified target path. The main utility of TPD lies in its ability to guide the generation of test data towards a target path, making it a vital tool for optimizing path testing methodologies [34]. To understand the TPD, one must comprehend the concept of a control flow graph (CFG). A CFG is a graphical representation of all paths that might be traversed through a program during its execution. Nodes in the CFG represent program instructions, and edges illustrate the potential flow between instructions.

The TPD between an execution path and a target path within the CFG is calculated by evaluating two critical components: the control dependence distance and the data dependence distance.

Control dependence distance: The control dependence distance examines the structural differences between the execution path and the target path within the CFG. For instance, it assesses whether the execution path traverses the same loops and conditionals as the target path. It is computed as a count of the control dependencies in the target path that are not satisfied by the execution path.

Data dependence distance: The concept of data dependence distance examines variations in the flow of data. It evaluates whether the same variables are assigned the same values on the execution path as on the target path. This distance is often measured as sum of the differences in variable values between the execution and target paths.

Mathematically, the TPD can be represented as shown in equation 2:

$$TPD = w1 * \text{Control Dependence Distance} + w2 * \text{Data Dependence Distance} \quad (2)$$

Where w1 and w2 are weights that determine the relative importance of control dependence and data dependence distances. These weights can be adjusted to accommodate the specific needs of the program or the testing requirements.

The calculation of the TPD provides a quantifiable measure that represents the distance between an execution path and a target path. The aim of an optimization algorithm, such as the TPDMVO proposed in this study, is to generate test data that minimizes the TPD. This, in turn, guides the execution closer to the target path [35].

## 3.7 Algorithm pseudocode

To provide a clear understanding of the TPDMVO for test data generation, the pseudocode is presented. The pseudocode in Figure 1 shows a step-by-step description of the algorithm. Note that it assumes the existence of helper functions for calculating the TPD and for generating new universes guided by test path distance.

***Algorithm: TPDMVO for Test Data Generation***

```
Input: Population size N, maximum number of iterations T, inflation rate W_ep, number of universes per wormhole W_wn
Output: Best-found test data
 1: Initialize a population of N universes
 2: Calculate the fitness (TPD) of each universe in the population
 3: Set the best universe as the one with the lowest fitness
 4: for t = 1 to T do
 5:     for each universe U in the population do
 6:         Update the inflation rate W_ep using a decreasing function
 7:         Update the white hole effect on U based on W_ep and fitness of U
 8:         Update the black hole effect on U based on the best universe and fitness of U
 9:         If the wormhole effect is triggered for U (based on W_wn and a random value) then
10:             Create a new universe U' guided by the target path (using TPD)
11:             Replace U with U' in the population
12:         end if
13:         Update the fitness of U
14:         If the fitness of U is lower than the fitness of the best universe then
15:             Set U as the new best universe
16:         end if
17:     end for
18: end for
19: Return the best-found test data (corresponding to the best universe)
```

**Fig. 1.** Pseudocode of enhanced MVO

The pseudocode outlined in Figure 1 provides a high-level overview of how the TPDMVO algorithm operates when applied to the problem of generating test data for path testing. This adaptation of MVO has been specifically tailored to effectively navigate through the vast space of potential test data, utilizing the TPD as the guiding principle.

The test path distance multi-verse optimizer begins by initializing a population of 'N' universes, which metaphorically represents potential solutions or test data (line 1). Each of these universes (test data sets) is evaluated using the fitness function, which is

defined by the TPD (line 2). The TPD serves as a heuristic measure of how closely a given test data set is to achieving the maximum path coverage. The best universe, which has the lowest TPD (indicating the highest path coverage), is initially stored (line 3).

The main procedure of TPDMVO (lines 4–18) is iteratively executed for a maximum of 'T' iterations. In each iteration, every universe in the population undergoes a series of transformations aimed at improving the test data sets. The inflation rate 'W_ep' is updated (line 6), which affects the white hole effect applied to the universe (line 7). The white hole effect symbolizes the exploratory aspect of the algorithm, generating new test datasets to expand the search space. The universe is also influenced by the black hole effect (line 8), which represents the exploitative aspect of the algorithm. This guides the universe towards the currently known best solution (the universe with the lowest TPD), enabling a focused search.

A universe may also undergo a wormhole effect, determined by a random chance and the number of universes per wormhole, denoted as 'W_wn' (line 9). If triggered, a new universe is created that is directly guided by the target path using TPD (line 10), replacing the existing universe in the population. This symbolizes a local search mechanism, enabling TPDMVO to fine-tune existing solutions. The fitness of each universe is updated (line 13). If a universe is found to have a better (lower) TPD than the current best universe, it is set as the new best universe (lines 14–16). This iterative process helps TPDMVO gradually refine the test data to achieve maximum path coverage. Finally, once the iterations are completed, TPDMVO returns the test data corresponding to the best universe found (line 19). This test data should ideally yield high path coverage when used for software testing. Thus, by integrating the TPD into the MVO's cosmology-inspired mechanisms, the TPDMVO can systematically explore, exploit, and fine-tune the test data sets, aiming for maximum path coverage and driving towards improved path testing.

## 3.8 Phases of the TMVO algorithm

The TPDMVO algorithm for path testing can be divided into several distinct phases. Here is a high-level description of each phase.

**Phase 1: Initialization.** In the initial phase, a population of universes (i.e., solutions) is randomly generated. Each universe represents a potential solution to the problem. The diversity of the initial population affects the ability of TPDMVO to explore the solution space. The population size, the dimensions of the universes, and the lower and upper bounds for each dimension are set in this phase.

```
Function Initialize:
    Create an initial population of random universes
```

**Phase 2: Fitness evaluation.** In this phase, the fitness of each universe in the population is evaluated using the designated fitness function. The fitness function is problem-specific and is designed to quantify the quality of a given solution. In the context of path testing, the fitness function might evaluate the TPD for each path.

```
Function Evaluate Fitness:
    For each universe in the population
        Calculate the fitness of the universe using the fitness function
```

**Phase 3: Universe update.** During the universe update phase, new universes are generated based on the fitness values of the current universes. The new universes are generated through three mechanisms: white holes, black holes, and wormholes. The wormhole mechanism is introduced in the TPDMVO to enhance the algorithm's exploratory capabilities.

```
Function Update Universe:
    For each universe in the population
        Update the universe based on white hole, black hole, and wormhole mechanisms
```

**Phase 4: Path testing.** In this phase, TPDMVO tests the new paths generated by the individuals in the population. The path coverage and the TPD are calculated for each path.

```
Function Test Path:
    For each universe in the population
        Test the path represented by the universe
```

**Phase 5: Convergence check.** In this phase, TPDMVO checks whether it has reached the stopping criteria, which can be a predefined number of iterations, a specific target fitness value, or no improvement in fitness over a certain number of iterations. If the stopping criteria are met, TPDMVO stops and returns the best solution that has been found. If not, it goes back to the Universe Update phase.

```
Function Check Convergence:
    If stopping criteria are met
        Return the best universe found
    Else
        Go back to the Universe Update phase
```

These phases represent a single iteration of the enhanced MVO. The phases are repeated for a set number of iterations or until the stopping criteria are met. The best solution found over all iterations is returned as the final output of the algorithm.

# 4 METHODOLOGY

## 4.1 Experimental setup

To evaluate the efficiency of the TPDMVO using TPD, an experimental framework has been established, which includes a set of case studies. The selected case studies encompass a range of complexity and structure to ensure a thorough evaluation of the algorithm. They include simple programs such as the minimum and maximum functions, which take a set of numbers and return the smallest and largest values, respectively. These functions have multiple paths due to conditional statements, making them suitable for path testing experiments.

More complex programs were also included, such as a program designed to find three numbers whose average is 150. This case study introduces additional complexity due to the larger input space and multiple paths resulting from the inclusion of computational logic. Furthermore, case studies involving data structures were also incorporated, such as the binary search tree and the AVL tree [32], these data

structures are commonly used in software applications. These cases present more complex branching and path scenarios, requiring more sophisticated test data to achieve comprehensive path coverage. Finally, the median function was selected. This program takes three numbers and returns the value that is neither the maximum nor the minimum. This case study, with its numerous potential paths, presents a significant challenge for path testing and therefore serves as a rigorous test for the TPDMVO algorithm.

Through this diverse set of case studies, we aim to comprehensively assess the capabilities of the enhanced MVO in generating effective test data for path testing across varying levels of software complexity. The performance of the TPDMVO will be carefully evaluated and compared to other standard methods in order to understand its strengths, potential limitations, and areas for further improvement.

## 4.2 Description of the test problems

In this section, we delve deeper into the specifics of the test problems used in our experimental setup, offering insight into their structure, complexity, and the rationale for their selection in our study.

Minimum function: This simple program takes an array of integers as input and returns the smallest number in the array. The complexity of the algorithm arises from the loop and conditional statements that iterate over the array to find the minimum value. Despite its apparent simplicity, this function is ideal for initial testing and calibration of the Enhanced MVO's capabilities.

Maximum function: Analogous to the minimum function, the maximum function also accepts an array of integers but returns the largest number instead. The complexity of the path is similar to that of the minimum function. The maximum function provides an additional testing scenario while maintaining the same level of complexity.

Arithmetic mean program: This is a more complex program that takes three numbers as input and verifies if their arithmetic mean is 150. The challenge in path testing arises from the presence of multiple conditional statements in the function, with each one representing a distinct execution path. This program is designed to test the ability of enhanced MVO to manage higher complexity and larger input space.

Binary search tree (BST): A tree-based data structure, a BST organizes nodes in a manner that enables efficient search, insertion, and deletion operations. Testing a BST program presents challenges due to its inherent complexity and the multitude of paths stemming from various operations. The BST case study examines the capability of enhanced MVO to handle complex, multi-path scenarios [33].

AV Tree: Named after its inventors, Adelson-Velsky and Landis, the AV tree is a self-balancing binary search tree [32]. Path testing a program that implements an AVL tree includes even more complex scenarios compared to a BST, making it an effective problem to further test the capabilities of the enhanced MVO.

Middle value function: This function takes three numbers as input and returns the value that is neither the maximum nor the minimum. The function involves multiple conditional statements and paths, making it a robust test case for the enhanced MVO.

These test problems span a wide range of complexity and represent both simple and complex real-world software applications. Through these problems, we aim to examine and comprehend the performance of the TPDMVO in generating test data for comprehensive path testing.

# 5 RESULTS AND DISCUSSION

In this section, we present the experimental outcomes achieved through the application of the TPDMVO in the domain of test data generation for path testing. The focus of the discussion is on the effectiveness and efficiency of the enhanced MVO, as demonstrated by its performance on the six diverse test problems. Comparisons with the conventional MVO and other optimization algorithms for test data generation are also made, highlighting the advancements and improvements facilitated by the proposed enhancements.

## 5.1 Case study 1: minimum function

The first case study explores the application of the TPDMVO algorithm on a basic minimum function, which is a common test problem in path testing literature. This function takes three input values, a, b, and c, and determines the smallest among them. This results in three possible paths: Path 1 (if "a" is the smallest), Path 2 (if "b" is the smallest), or Path 3 (if "c" is the smallest). Our TPDMVO algorithm tackles this problem by iteratively generating and evaluating test data using the TPD mechanism. The algorithm firstly initializes a population of universes (solutions) with random values for a, b, and c. In the next steps, TPDMVO employs its optimization mechanisms: the wormhole mechanism and the enhanced exploration and exploitation techniques, to iteratively update these universes.

The fitness of each universe is then calculated based on the TPD, where smaller distances indicate better fitness. This iterative process continues until a stopping criterion, such as reaching a maximum number of iterations or achieving a satisfactory fitness level, is met.

```
Case study 1 Algorithm Pseudocode:
Initialize a population of universes with random values for a, b, and c.
Calculate the fitness of each universe using the Test Path Distance.
While the stopping criterion is not met, do:
Apply the wormhole mechanism to generate new potential solutions.
Update the universes using the enhanced exploration and exploitation techniques.
Calculate the fitness of the updated universes.
Select the universe with the smallest Test Path Distance.
Return the best universe as the solution.
```

**Table 1.** AWTGWO path testing results for minimum function

| Input Numbers | Output (minimum) | Path Executed | Fitness Function Value | Path Coverage % |
|---|---|---|---|---|
| 1, 2, 3 | 1 | Path 1 | 1 | 100% |
| 5, 3, 7 | 3 | Path 2 | 1 | 100% |
| 8, 7, 6 | 6 | Path 3 | 1 | 100% |
| 4, 4, 4 | 4 | Path 1 | 1 | 100% |

The results in Table 1 for the minimum function show that the TPDMVO, with the objective of maximizing path coverage, is highly effective in achieving full path coverage of the target program. TPDMVO generated input values that executed all three

paths in the function for all input values, resulting in a path coverage percentage of 100%. The fitness function value was 1 for all input values, indicating that TPDMVO successfully executed all unique paths. The fitness function encourages TPDMVO to search for input values that execute new paths in the target program, thereby helping to achieve full path coverage. The results demonstrate the effectiveness of the TPDMVO for path coverage testing in the context of a real-world program. Achieving full path coverage is important for ensuring that all possible scenarios in a program are tested, thereby reducing the likelihood of undiscovered bugs and improving software reliability.

## 5.2 The second case study: maximum function

The second case study focuses on the maximum function. In this program, the function accepts three input values (a, b, and c) and identifies the maximum of these three numbers. The function diverges into one of three possible paths based on which number is the maximum: Path 1 for 'a', Path 2 for 'b', and Path 3 for 'c'.

The TPDMVO algorithm can address this problem by generating input values that activate all the unique paths in the function. The objective function, which is designed to maximize path coverage, facilitates this. High-level steps of TPDMVO might look as follows:

Initialize a population of universes (solutions).
For each universe, calculate the fitness function value (path coverage) based on the generated input values.
Update the universes using the mechanisms of the MVO algorithm.
If a universe finds a new maximum number, update the path accordingly.
Repeat steps 3–4 until a termination criterion is met (such as a maximum number of iterations).
The final solution will be the universe that has achieved the highest fitness function value.

**Table 2.** TPDMVO path testing results for maximum function

| Input Numbers | Output (maximum) | Path Executed | Fitness Function Value | Path Coverage % |
|---|---|---|---|---|
| 1, 2, 3 | 3 | Path 3 | 1 | 100% |
| 5, 3, 7 | 7 | Path 3 | 1 | 100% |
| 8, 7, 6 | 8 | Path 1 | 1 | 100% |
| 4, 4, 4 | 4 | Path 3 | 1 | 100% |

As it can be seen in Table 2 that the TPDMVO algorithm, with its implementation of TPD, demonstrates exemplary performance in maximizing path coverage in the case of the maximum function. The inclusion of TPD in TPDMVO not only focuses on executing all paths but also considers the diversity of the covered paths. This diversity is crucial in ensuring a thorough assessment of the software.

The TPDMVO calculates the TPD for each set of input values and utilizes this information to guide the search process. The TPD, which measures the distance between the current test path and other test paths, promotes diversity in the search space by encouraging TPDMVO to explore different areas of the program's path space. In the case of the maximum function, the TPD aids in achieving a path coverage percentage of 100% across all inputs. This is because the TPD guided TPDMVO towards

executing all unique paths in the function, ensuring a diverse set of test cases. The fitness function values, at 1 for all inputs, affirm the algorithm's success in executing all unique paths, which is attributable to the test path distance.

Hence, the inclusion of TPD in the TPDMVO algorithm not only ensures that all possible scenarios in a program are tested, but it also ensures that the test cases are diverse. This, in turn, reduces the likelihood of undiscovered bugs and improves software reliability.

## 5.3 Case study 3 the middle value function

In our third case study, we examine the "middle value function." This program accepts three input values: a, b, and c. It determines the middle value among the three numbers and returns one of three potential paths accordingly. Implementing the TPDMVO algorithm in this scenario involves several stages. First, TPDMVO initializes a population of random solutions. Each solution consists of a set of three potential inputs for the function. For each solution, TPDMVO determines which path it would execute by running the function with the proposed inputs. TPDMVO then evaluates each solution using the fitness function, where fitness corresponds to the number of unique paths executed by a solution.

To solve the case of the 'middle value function', the TPDMVO algorithm iteratively updates its solutions using exploration and exploitation mechanisms until it finds a set of input values that can execute all possible paths in the function. The TPD is calculated for each solution, guiding TPDMVO to explore different paths in the function.

An overview of the TPDMVO for this case study is as follows:

Initialize a population of solutions.
For each solution, run the middle value function using the inputs proposed by the solution.
Calculate the fitness of each solution.
Update the solutions using the TPDMVO mechanisms.
If the stopping criteria are met, terminate the algorithm. Otherwise, return to step 2.

**Table 3.** TPDMVO path testing results for the middle value function

| Input Numbers | Output (middle) | Path Executed | Fitness Function Value | Path Coverage % |
|---|---|---|---|---|
| 1, 2, 3 | 2 | Path 1 | 1 | 100% |
| 5, 3, 7 | 5 | Path 2 | 1 | 100% |
| 8, 7, 6 | 7 | Path 3 | 1 | 100% |
| 4, 4, 4 | 4 | Path 2 | 1 | 100% |

Looking at Table 3, we observe the exceptional performance of the TPDMVO algorithm in achieving complete path coverage. All three paths are executed, with each path having a unique set of inputs. The fitness function value, uniformly 1 across all input sets, indicates that all unique paths were successfully executed. These results underscore the TPDMVO algorithm's effectiveness, especially when paired with the concept of TPD. The TPD, by providing a measure of difference between the currently covered paths and uncovered ones, helps in maintaining a diverse exploration of the program's path space. This ensures the execution of all paths and, consequently, provides a more comprehensive path coverage.

## 5.4 Case study 4: binary search tree data

Our fourth case study introduces the binary search tree (BST), which is a hierarchical data structure in which each node has a unique value. For any given node in a BST, all nodes in its left subtree have a smaller value, and all nodes in its right subtree have a greater value. Given the algorithm's distinct paths depending on the input data, the BST is an excellent candidate for our path testing efforts. The application of the TPDMVO algorithm to this problem involves several steps. First, TPDMVO initiates a population of potential solutions, each containing a set of input values for the BST. For each solution, it determines the execution path through the BST. TPDMVO subsequently evaluates the fitness of each solution based on the number of unique paths executed by that solution and the TPD, which measure of the difference between the currently executed paths and the remaining ones.

The TPDMVO algorithm is then set to work, systematically updating the solutions using its exploration and exploitation mechanisms. TPDMVO keeps optimizing until it identifies a set of input values that execute all possible paths in the function. Here is a simplified representation of TPDMVO for the BST case.

Initialize a population of potential solutions.
For each solution, determine the execution path through the BST with the input values.
Calculate the fitness of each solution.
Update the solutions using the TPDMVO mechanisms.
If the stopping criteria are met, terminate the algorithm. If not, return to step 2.

**Table 4.** TPDMVO path testing results for binary search tree

| Input Values | Path Executed | Fitness Function Value | Path Coverage % |
|---|---|---|---|
| [5, 2, 3, 7, 6] | Path 1–3–6 | 3 | 100% |
| [5, 5, 5, 5, 5] | Path 1–2–4 | 1 | 100% |
| [] | Path 1 | 0 | 100% |
| [10] | Path 1–2 | 1 | 100% |

The results in Table 4 shows that the TPDMVO algorithm performs remarkably well in achieving complete path coverage for the binary search tree case. Each path has a unique set of inputs, resulting in a 100% path coverage. The fitness function value corresponds to the height of the BST generated from each input set, which reflects the efficiency of the algorithm. Moreover, the utilization of TPD in the TPDMVO approach enables a more comprehensive exploration of the program's path space. By helping TPDMVO to focus not only on the currently covered paths but also on the uncovered ones, it ensures a more comprehensive coverage of paths.

## 5.5 Case study 5: AVL tree

The fifth case study introduces an AVL tree data structure. An AVL tree is a self-balancing binary search tree that ensures the height difference between the left and right subtrees of any node is at most one. This feature maintains the tree in a well-balanced state, thereby improving the efficiency of search operations. Due to the numerous possible paths that TPDMVO can take based on input data, the AVL tree proves to be an ideal candidate for path testing. The application of the TPDMVO

algorithm to this problem involves several steps. First, TPDMVO initiates a population of potential solutions, with each solution containing a set of input values for the AVL tree. For each solution, it determines the execution path through the AVL tree, with the input values serving as the input data for the tree.

The TPDMVO then evaluates the fitness of each solution based on the number of unique paths executed by that solution and the "Test Path Distance," which measures the difference between the currently executed paths and the remaining paths in the AVL tree function. The TPDMVO algorithm proceeds by systematically updating the solutions using its exploration and exploitation mechanisms. The process of updating continues until TPDMVO identifies a set of input values that execute all possible paths in the function. Here is a high-level representation of the algorithm:

```
Initialize a population of potential solutions.
For each solution, determine the execution path through the AVL tree using the input values.
Calculate the fitness of each solution.
Update the solutions using the TPDMVO mechanisms.
If the stopping criteria are met, terminate the algorithm. If not, return to step 2.
```

**Table 5.** TPDMVO path testing results for AVL tree

| Input Values | Path Executed | Fitness Function Value | Path Coverage % |
|---|---|---|---|
| [5, 2, 3, 7, 6] | Path 1–2–4 | 2 | 100% |
| [5, 5, 5, 5, 5] | Path 1–2–4 | 1 | 100% |
| [] | Path 1 | 0 | 100% |
| [10] | Path 1–2 | 1 | 100% |

When examining the results in Table 5, it is evident that the TPDMVO algorithm has achieved exceptional performance in achieving complete path coverage for the AVL tree case. Each input array corresponds to a unique path, resulting in a 100% path coverage. The fitness function value corresponds to the height of the AVL tree constructed from each input array, further confirming the efficiency of the algorithm.

Moreover, the inclusion of the TPD in the TPDMVO approach enables a more thorough exploration of the path space of the AVL tree function. By directing TPDMVO to focus not only on the currently covered paths but also on the uncovered ones, it ensures a more comprehensive coverage of paths. These results confirm the effectiveness of the TPDMVO algorithm in handling path coverage testing in more complex scenarios, such as an AVL tree. By ensuring that all possible scenarios in a program are tested, it reduces the chances of overlooked bugs and enhances the reliability of the software.

## 6 DISCUSSION

The implementation of the TPDMVO algorithm on various real-world programs have resulted in significant discoveries. Each case study, ranging from simple functions like identifying the maximum and middle value to complex data structures such as binary search trees and AVL trees, offered a unique perspective on the capabilities of the TPDMVO algorithm. Across all case studies, the TPDMVO

algorithm consistently demonstrated the ability to achieve complete path coverage, as confirmed by a path coverage percentage of 100%. The effectiveness of the algorithm lies in its ability to generate a diverse set of test inputs, enabling the execution of all possible paths in a given function or data structure. In doing so, it reduces the likelihood of undetected bugs, thereby contributing to improved software reliability.

One of the significant features of the TPDMVO algorithm that contributed to these results is its usage of the "Test Path Distance" metric. By considering the difference between the currently executed paths and the remaining paths in a function, TPDMVO can direct its search towards unexplored areas. This strategy not only contributed to achieving full path coverage but also encouraged a more uniform exploration of the path space. The fitness function used by the TPDMVO algorithm served as another critical aspect of its performance. By incorporating a case-specific function, TPDMVO was encouraged to find solutions that not only achieved full path coverage but also optimized another characteristic of the system, such as minimizing the height in the case of binary search and AVL trees.

In cases where the test subject was a more complex data structure, the TPDMVO algorithm still maintained a high level of performance. This consistency points to the algorithm's potential to handle a wide variety of software testing scenarios. The findings from these case studies illustrate the effectiveness of the TPDMVO algorithm for path coverage testing. By utilizing a combination of the 'Test Path Distance' metric and a context-specific fitness function, TPDMVO demonstrates its effectiveness as a reliable tool for enhancing path coverage and, ultimately, software reliability. Furthermore, the ability of the TPDMVO to handle a range of complexities in test subjects suggests its wide applicability in software testing and quality assurance practices.

## 7 CONCLUSION

This research aimed to evaluate the effectiveness of the newly introduced TPDMVO algorithm, particularly in the field of software testing, and more specifically, in path coverage testing. The investigative lens was cast across a spectrum of case studies, ranging from basic functions to complex data structures. The consistently high achievement of 100% path coverage across these diverse scenarios highlights the effectiveness of the TPDMVO algorithm in ensuring comprehensive testing coverage. A crucial factor in this improved performance can be attributed to the TPD metric. This metric strategically directed the search operations towards unexplored paths, thereby preventing the TPDMVO from becoming stagnant within a limited range of familiar routes. Consequently, this facilitated a comprehensive and rigorous testing phase, significantly reducing the chances of overlooking hidden bugs. Further enhancing the efficacy of the TPDMVO was the integration of a context-adaptable fitness function, enabling nuanced customization to align with the unique requirements of each specific scenario. This not only guarantees optimal path coverage but also streamlines other inherent attributes specific to the testing subject. This is exemplified by aspects such as tree height in scenarios involving binary search and AVL trees. The adeptness of the TPDMVO algorithm across a wide range of complexities highlights its potential as an essential asset in software testing. Its demonstrable capability, ranging from elementary functions to sophisticated data structures, stands as a testament to its promise in enhancing software reliability.

## 8 REFERENCES


[1] Y. K. Dwivedi *et al.,* "Artificial Intelligence (AI): Multidisciplinary perspectives on emerging challenges, opportunities, and agenda for research, practice and policy," *International Journal of Information Management*, vol. 57, p. 101994, 2021. https://doi.org/10.1016/j.ijinfomgt.2019.08.002

[2] D. Bertsimas and V. Goyal, "On the power of robust solutions in two-stage stochastic and adaptive optimization problems," *Mathematics of Operations Research*, vol. 35, no. 2, pp. 284–305, 2010. https://doi.org/10.1287/moor.1090.0440

[3] Y. B. Leau, W. K. Loo, W. Y. Tham, and S. F. Tan, "Software development life cycle AGILE vs traditional approaches," in *International Conference on Information and Network Technology*, vol. 37, no. 1, pp. 162–167, 2012.

[4] R. M. Sharma, "Quantitative analysis of automation and manual testing," *International Journal of Engineering and Innovative Technology*, vol. 4, no. 1, 2014.

[5] D. S. Battina, "Artificial intelligence in software test automation: A systematic literature review," *International Journal of Emerging Technologies and Innovative Research*, ISSN 2349-5162, 2019.

[6] D. Graham and M. Fewster, *Experiences of Test Automation: Case Studies of Software Test Automation*. Addison-Wesley Professional, 2012.

[7] M. H. Ryalat, H. N. Fakhouri, J. Zraqou, F. Hamad, and M. S. Alzboun, "Enhanced multi-verse optimizer (TMVO) and applying it in test data generation for path testing," *International Journal of Advanced Computer Science and Applications*, vol. 14, no. 2, 2023. https://doi.org/10.14569/IJACSA.2023.0140277

[8] K. Li and M. Wu, *Effective Software Test Automation: Developing An Automated Software Testing Tool*. John Wiley & Sons, 2006.

[9] Li and Y. Zhang, "Automatic generating all-path test data of a program based on PSO," in *2009 WRI World Congress on Software Engineering*, vol. 4, pp. 189–193, 2009. https://doi.org/10.1109/WCSE.2009.98

[10] U. Jaiswal and A. Prajapati, "Optimized test case generation for basis path testing using improved fitness function with PSO," in *Thirteenth International Conference on Contemporary Computing (IC3-2021)*, pp. 475–483, 2021. https://doi.org/10.1145/3474124.3474197

[11] R. R. Sahoo and M. Ray, "PSO based test case generation for critical path using improved combined fitness function," *Journal of King Saud University-Computer and Information Sciences*, vol. 32, no. 4, pp. 479–490, 2020. https://doi.org/10.1016/j.jksuci.2019.09.010

[12] S. Alaliyat, R. Oucheikh, and I. Hameed, "Path planning in dynamic environment using particle swarm optimization algorithm," in *8th International Conference on Modeling Simulation and Applied Optimization (ICMSAO)*, pp. 1–5, 2019. https://doi.org/10.1109/ICMSAO.2019.8880434

[13] Damia, M. Esnaashari, and M. Parvizimosaed, "Automatic web-based software structural testing using an adaptive particle swarm optimization algorithm for test data generation," in *7th International Conference on Web Research (ICWR)*, pp. 282–286, 2021. https://doi.org/10.1109/ICWR51868.2021.9443153

[14] F. Gul *et al.,* "Meta-heuristic approach for solving multi-objective path planning for autonomous guided robot using PSO–GWO optimization algorithm with evolutionary programming," *Journal of Ambient Intelligence and Humanized Computing*, vol. 12, pp. 7873–7890, 2021. https://doi.org/10.1007/s12652-020-02514-w

[15] X. Cheng *et al.,* "An improved PSO-GWO algorithm with chaos and adaptive inertial weight for robot path planning," *Frontiers in Neurorobotics*, vol. 15, p. 770361, 2021. https://doi.org/10.3389/fnbot.2021.770361

[16] S. Mirjalili, S. M. Mirjalili, and A. Hatamlou, "Multi-verse optimizer: A nature-inspired algorithm for global optimization," *Neural Computing and Applications*, vol. 27, pp. 495–513, 2016. https://doi.org/10.1007/s00521-015-1870-7

[17] L. Abualigah, "Multi-verse optimizer algorithm: A comprehensive survey of its results, variants, and applications," *Neural Computing and Applications*, vol. 32, no. 16, pp. 12381–12401, 2020. https://doi.org/10.1007/s00521-020-04839-1

[18] D. Zhang, J. Sui, and Y. Gong, "Large scale software test data generation based on collective constraint and weighted combination method," *Tehnicki Vjesnik/Technical Gazette*, vol. 24, no. 4, 2017. https://doi.org/10.17559/TV-20170319045945

[19] P. McMinn, "Search-based software test data generation: A survey," *Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, 2004. https://doi.org/10.1002/stvr.294

[20] M. H. Ryalat *et al.,* "Enhanced multi-verse optimizer (TMVO) and applying it in test data generation for path testing," *International Journal of Advanced Computer Science and Applications*, vol. 14, no. 2, 2023. https://doi.org/10.14569/IJACSA.2023.0140277

[21] J. Edvardsson, "A survey on automatic test data generation," in *Proceedings of the 2nd Conference on Computer Science and Engineering*, pp. 21–28, 1999.

[22] H. Tahbildar and B. Kalita, "Automated software test data generation: Direction of research," *International Journal of Computer Science and Engineering Survey*, vol. 2, no. 1, pp. 99–120, 2011. https://doi.org/10.5121/ijcses.2011.2108

[23] X. Guo, H. Okamura, and T. Dohi, "Automated software test data generation with generative adversarial networks," *IEEE Access*, vol. 10, pp. 20690–20700, 2022. https://doi.org/10.1109/ACCESS.2022.3153347

[24] C. C. Michael, G. McGraw, and M. A. Schatz, "Generating software test data by evolution," *IEEE Transactions on Software Engineering*, vol. 27, no. 12, pp. 1085–1110, 2001. https://doi.org/10.1109/32.988709

[25] N. Tracey *et al.,* "Automated test-data generation for exception conditions," *Software: Practice and Experience*, vol. 30, no. 1, pp. 61–79, 2000. https://doi.org/10.1002/(SICI)1097-024X(200001)30:1<61::AID-SPE292>3.0.CO;2-9

[26] C. J. Hsu and C. Y. Huang, "An adaptive reliability analysis using path testing for complex component-based software systems," *IEEE Transactions on Reliability*, vol. 60, no. 1, pp. 158–170, 2011. https://doi.org/10.1109/TR.2011.2104490

[27] R. Khan, M. Amjad, and A. K. Srivastava, "Optimization of automatic generated test cases for path testing using genetic algorithm," in *Second International Conference on Computational Intelligence & Communication Technology (CICT)*, pp. 32–36, 2016. https://doi.org/10.1109/CICT.2016.16

[28] P. B. Nirpal and K. V. Kale, "Using genetic algorithm for automated efficient software test case generation for path testing," *International Journal of Advanced Networking and Applications*, vol. 2, no. 6, pp. 911–915, 2011.

[29] Z. Zhonglin and M. Lingxia, "An improved method of acquiring basis path for software testing," in *5th International Conference on Computer Science & Education*, pp. 1891–1894, 2010. https://doi.org/10.1109/ICCSE.2010.5593820

[30] D. B. Mishra, R. Mishra, K. N. Das, and A. A. Acharya, "Test case generation and optimization for critical path testing using genetic algorithm," in *Soft Computing for Problem Solving: SocProS 2017*, Springer Singapore, vol. 2, pp. 67–80, 2019. https://doi.org/10.1007/978-981-13-1595-4_6

[31] M. M. Syaikhuddin, C. Anam, A. R. Rinaldi, and M. E. B. Conoras, "Conventional software testing using white box method," *Kinetik: Game Technology, Information System, Computer Network, Computing, Electronics, and Control*, vol. 3, no. 2, pp. 65–72, 2018. https://doi.org/10.22219/kinetik.v3i1.231

[32] Hermadi, C. Lokan, and R. Sarker, "Genetic algorithm based path testing: Challenges and key parameters," in *Second World Congress on Software Engineering*, vol. 2, pp. 241–244, 2010. https://doi.org/10.1109/WCSE.2010.82

[33] P. Flajolet, X. Gourdon, and C. Martínez, "Patterns in random binary search trees," *Random Structures & Algorithms*, vol. 11, no. 3, pp. 223–244, 1997. https://doi.org/10.1002/(SICI)1098-2418(199710)11:3<223::AID-RSA2>3.0.CO;2-2

[34] J. A. Jones, M. J. Harrold, and J. T. Stasko, "Visualization for fault localization," in *Proceedings of ICSE 2001 Workshop on Software Visualization*, 2001.

[35] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th International Conference on Software Engineering*, May 2002, pp. 467–477. https://doi.org/10.1145/581396.581397

## 9 AUTHORS

**Hussam N. Fakhouri,** Data Science and Artificial Intelligence Department, Faculty of Information Technology, University of Petra, Amman, Jordan (E-mail: hussam.fakhouri@uop.edu.jo).

**Ahmad K. Al Hwaitat,** King Abdullah II School of Information Technology, The University of Jordan, Amman, Jordan.

**Mohammad Ryalat,** Prince Abdullah bin Ghazi faculty of Information and Technology, Al-Balqa Applied University, Salt, Jordan.

**Faten Hamad,** The University of Jordan, Amman, Jordan; Sultan Qaboos University, Sultanate of Oman, al-Seeb, Oman.

**Jamal Zraqou,** Virtual and Augmented Reality Department, Faculty of Information Technology, University of Petra, Amman, Jordan.

**Adi Maaita,** Middle East University, Amman, Jordan.

**Mohannad Alkalaileh,** Al Ain University, Al Ain, United Arab Emirates.

**Najem N. Sirhan,** Faculty of Information Technology, University of Petra, Amman, Jordan.