Modern Applied Science; Vol. 12, No. 7; 2018
ISSN 1913-1844 E-ISSN 1913-1852
Published by Canadian Center of Science and Education

Using Artificial Bee Colony Algorithm for Test Data Generation and Path Testing Coverage

Faten Hamad[1]

[1] School of educational sciences, the university of Jordan, amman, Jordan

Correspondence: Faten Hamad, School of Educational Sciences, The University of Jordan, Amman, Jordan. E-mail: f.hamad@ju.edu.jo

Received: January 3, 2018 Accepted: May 18, 2018 Online Published: June 30, 2018

doi:10.5539/mas.v12n7p99 URL: https://doi.org/10.5539/mas.v12n7p99

Abstract

Software testing is a significant stage in software development lifecycle. There are different sorts of' structural software testing methodologies that may be generally utilized and moved forward through enhancing the traverse of all of the conceivable code software paths. The interest for automating data testing is growing; however, manual testing strategies utilization would be expensive and costly. Heuristic measure is being applied to; detect how better the result might be (solution fitness); result development mechanism; and suitableness criteria with stop search mechanism depending on wither a result is found or not. Testing experience could be exploited for finding a solution to the optimization problem by utilizing Meta heuristic procedures. The presented approach might have been tested for five programs to demonstrate the distinctive tests issues. This paper proposes an automatic test data generation approach that use artificial bee colony algorithm for software structural testing, particularly, path testing. This is brought on moving the centralization of data generation testing, as opposed to the automation of the whole testing operation. It executes artificial bee colony algorithm by creating testing data for the criteria of path coverage testing, and then applying the strategy to a group of test programs.

Keywords: software testing, heuristic measure, automating data testing

1. Introduction

Optimization can be characterized as the best probability search and expected solutions that can be provided to solve a problem. There would be number of optimization mechanisms, that are both conventional also meta-heuristics. Those conventional techniques are gradient based that are exactly quicker over convergence; however, they are not appropriate for non differentiable and unpredictable multimodal functions. These mechanisms have confinements for finding the global optimal be it get stuck into local optimal value as they begin with only one point. There would be a lot of search techniques to solve this problem but they are slower and need exponential time (Alauddin, 2016). The most common method is Meta heuristic optimization techniques.

Meta heuristic is getting very common in the most recent two decades. For example, Genetic Algorithm, multi verse and Particle Swarm Optimization, are well known to computer researchers from various fields. large amount of theoretical science, from different study fields have been applying optimization. Meta heuristic is very popular due to its simplicity, flexibility, derivation of free techniques, and local optima shirking (Mirjalili, 2014). White box criteria software testing contains a group of test cases which increase these criteria an experiment will be an indication that calls those test functions with specific input group values. Those driver afterward make ~~a~~ comparisons of the between the output and the relied one. Utilizing know workable inputs will be infeasible so that their number may be limitless. Therefore software testing automation trying in this setting comprises about naturally discovering the littlest group of those inputs so test criterion may be expanded. (Arcuri, 2012).

Black box and white box testing both are testing methodologies. White box testing (structural testing) is a procedure that uncovers inside instructions and is applied from programs throughout test runs. In structural test, the principle destination will be achieved by testing every code path for specific test information inputs. Picking diverse control flow path to test is very important as a result of vast number of paths consequence vast number for test succession which is challenging to be performed. There are a large number of problem viewing paths over software testing like discovering paths to cover system module, Prioritize those paths, Produce test data for every path and assess test result (Biswas et.al, 2015).

The algorithm for choosing the critical path necessities will Figure out the most punctual and most recent occurring event time, and also the most punctual and start time about all activities. That algorithm execution will be error prone and wasteful for programs with complex variables. Critical paths determination of the source codes is executed by the algorithm where every path matrix will be created by resolving the nearness AOE matrix system depends on those source codes.

Test cases creation for critical path coverage testing depends on the idea about Linear Coded Sequence. Through all critical paths the effectiveness of the creation for them might be enhanced at the same time, it will be suitable with design the test cases with different testing tools ~~aid~~ which might give the investigation about LCSAJ to those tested program (Zhi, 2014).

This paper is organized as follows. In section 2 Related Work is presented. Software testing, testing types and Software Path Coverage Test is described in section 3. Multi-Verse Optimizer are reported in section 4. Testing ABC on the CEC2005 Benchmarks functions is presented in section 5, Experiment Results and Analysis is being described in section 6. The paper ends results of case studies are discussed in section 7.

2. Related Work

The Artificial Bee Colony Algorithm (ABC) is a swarm based meta-heuristic algorithm, that was introduced by Karaboga in 2005 to solve problems related to numerical optimization. The algorithm was developed based on the intelligent behavior of the honey bees' foraging process (Karaboga, 2009). Biswas (2015) introduced ant colony optimization (ACO) based algorithm which produce group of' ideal paths in form of what's more prioritize the paths. Additionally, those methodology results test data grouping inside the area to utilize similarly as input of' the created paths. Suggested techniques ensure full coverage of software for least redundancy. Kun et. Al (2016) adopted a propagation error model should describe those methodology for the evolution of defects, What's more fault injection technique that will present seed defect. Seed defect is being activated and relating possibility defects that are prompted with test case design. This technique utilizes intelligent algorithms that should permanently design test cases to spread paths about seed defects, furthermore propagation paths, so all the the undetected and associated defects could be distinguished in propagation paths.

Kaur et. al. (2016) presented testing utilizing models that may be an incoming methodology for using the idea of black box testing. Furthermore model completeness that will be used; methodology for design/model system under testing with increase precisions. It might make joined of the Model based testing traditional techniques trying filling those loopholes and gabs that appears through design and test stages. In this methodology we might see that front end and in addition system structure together. The methodology will demonstrate with make guaranteeing for enhancing testing Furthermore nature of test instances.

Jain et. al. (2016) recommends the utilization of Aspect Oriented Programming (AOP) for the reason for crawling inside those program's modules without changing their source code and component experiments the place that are bugs being suspected. AOP might be catching alternately that's only execution focuses in the system utilizing pointcuts and moreover it might be composed to embed important code at each execution points for testing reason. Suitability has been analyzed from utilizing viewpoints for composing testing codes what are more performing different sorts of software testing. Zhang et. Al accompanied with software cybernetics, Dynamic Random Testing might have been recommended to enhance those conventional random testing and random partition testing methodologies. On the support for Dynamic Random testing which may be proposed for further enhancement the viability of the testing Dissimilar to those first it takes privileges about additional historical testing information with the estimation of defects identification rate for every sub domain in real time something like that Concerning illustration to settle on those upgrade for trying profile that's more sensible.

Danilov et.al. recommends that utilization for model for usability quality estimation for every module enhances accuracy of test procedure. Privileges of recommended models, as opposed with well known, may be that they permit thought about debugging test efficiencies, models deals with Different degrees of certainty for various tests. This let us will expand those software dependability characteristic by Creating test that detect errors with most probabilities. Models permit us to pick the best programming test system In light of accessible fundamental resources, effort. Ideally calendar assignments to various test and advancement group.

Pina et.al. presents Tedsuto, testing structure for DSU, alongside with usage of it to Rubah, An instance of symbolization Java based DSU framework. Tedsuto utilize system tests formed for the old What's more new forms of the updateable software, deliberately tests if a progressive upgrade may bring about a failed of test. Altogether frequently this methodology is completely automated, where in other percentage cases percentage manual annotations would be required. to assess the efficacy of Tedsuto's, dynamic updates is being applied before produced to the H2 SQL database server and the CrossFTP server a real world system with multi threaded

frameworks. Hoffmann et. Al (2016) presents idiosyncrasies for automotive embedded software and demonstrates those advantages of leveraging typical data on produce test cases. Examine Furthermore analyze separate strategies to generating test instances. Assess constantly on every methodology observationally once related, real world programs.

3. Software Testing

Software testing is the process of researching, testing a software product that aims to verify the correspondence between the actual behavior of a program and its expected behavior on a finite set of tests selected in a certain way (ISO / IEC TR 19759: 2005) [1].

3.1 Test Definitions

At different times and in various sources testing was given various definitions, including:

the process of executing the program for the purpose of finding errors [2]; an intellectual discipline aimed at obtaining reliable software without undue effort to verify it [3]; technical research program to obtain information about its quality in terms of a certain range of stakeholders (S. Kaner); checking the correspondence between the actual behavior of the program and its expected behavior on a finite set of tests performed in a certain way [1]; the process of monitoring the implementation of the program in special conditions and making on this basis an assessment of any aspects of its work [4]; a process aimed at identifying situations in which the behavior of the program is incorrect, undesirable or not conforming to the specification [5]; a process that contains all the life cycle activities, both dynamic and static, relating to the planning, preparation and evaluation of the software product and the related work results in order to determine that they meet the described requirements, to show that they are suitable for the stated purposes and to determine defects [6]; (Kosindrdecha, 2010).

3.2 History of Software Testing

The first software systems were developed as part of research programs or programs for the needs of defense ministries. The testing of such products was strictly formalized with the recording of all test procedures, test data, and the results obtained. Testing was allocated in a separate process, which began after the end of the encoding, but it was usually performed by the same personnel.

In the 1960s, much attention was paid to "exhaustive" testing, which should be done using all paths in the code or all possible input data. It was noted that in these conditions, a complete testing of software is impossible, because, firstly, the number of possible input data is very large, secondly, there are many ways, thirdly, it is difficult to find problems in architecture and specifications. For these reasons, "exhaustive" testing was rejected and found to be theoretically impossible.

In the early 1970s, software testing was referred to as "a process aimed at demonstrating the correctness of the product" or as "activity to confirm the correctness of the software". In the emerging software engineering software verification was listed as "proof of correctness". Although the concept was theoretically promising, in practice it took a long time and was not comprehensive enough. It was decided that proof of correctness is an ineffective method for testing software. However, in some cases, the demonstration of correct operation is also used today, for example, acceptance tests. In the second half of the 1970s, testing was presented as executing a program with the intention of finding errors, rather than proving that it worked. A successful test is a test that detects previously unknown problems. This approach is directly opposite to the previous one. These two definitions are a "testing paradox", based on two opposing statements: on the one hand, testing allows you to verify that the product works well, and on the other hand, identifies errors in the programs, indicating that the product does not work. The second purpose of testing is more productive in terms of quality improvement, since it does not allow to ignore the shortcomings of the software.

In the 1980s, testing was expanded with the notion of defect prevention. Design tests - the most effective of the known methods of preventing errors. At the same time, the idea began to be expressed that a testing methodology was needed, in particular that testing should include checks throughout the development cycle, and this should be a controlled process. During testing, it is necessary to check not only the collected program, but also the requirements, code, architecture, tests themselves. "Traditional" testing, which existed before the early 1980s, applied only to a compiled, ready-made system (now this is usually called system testing), but in the future testers began to get involved in all aspects of the development life cycle. This allowed us to find problems in the requirements and architecture earlier and thereby reduce the time and budget for development. In the mid-1980s, the first tools for automated testing appeared. It was assumed that the computer will be able to perform more tests than a person, and will do it more reliably. Initially, these tools were extremely simple and did not have the ability to write scripts in scripting languages.

In the early 1990s, the concept of "testing" began to include planning, designing, creating, maintaining and executing tests and test environments, and this meant a transition from testing to quality assurance covering the entire software development cycle. At this time, various software tools begin to appear to support the testing process: more advanced automation environments with the ability to create scripts and generate reports, a test management system, software for stress testing. In the mid-1990s, with the development of the Internet and the development of a large number of web applications, it became particularly popular to receive "flexible testing" (similar to flexible programming methodologies).

In the 2000s, an even broader definition of testing emerged, when the concept of "business technology optimization" was added to it [the source was not specified 1304 days]. The main approach is to assess and maximize the importance of all stages of the software development life cycle to achieve the required level of quality, performance, availability. (Trivedi, 2012).

Levels of testing

Component testing - testing the lowest possible component for testing, for example, a separate class or function. Often, components are tested by software developers.

Integration testing - interfaces are tested between components, subsystems or systems. If there is a time reserve at this stage, testing is carried out iteratively, with a gradual connection of subsequent subsystems.

System testing - the integrated system is tested for its compliance with requirements.

Alpha testing is an imitation of real work with the system by full-time developers, or real work with the system by potential users / customers. Most often, alpha testing is carried out at an early stage of product development, but in some cases it can be used for a finished product as an internal acceptance test. Sometimes alpha testing is performed under a debugger or using an environment that helps to quickly identify the errors found. The detected errors can be passed on to testers for additional investigation in an environment similar to the one in which the program will be used.

Beta testing - in some cases, the distribution of the preliminary version (in the case of proprietary software sometimes with limitations in functionality or working time) is performed for some larger group of individuals in order to ensure that the product contains few errors. Sometimes beta testing is performed in order to get feedback about the product from its future users.

Often for free and open source software, the alpha testing phase is characterized by the functional content of the code, and beta testing is the error correction stage. At the same time, as a rule, at each stage of development, intermediate results of work are available to end users.

Static and dynamic testing

The techniques described below - testing the white box and testing the black box - assume that the code is being executed, and the difference is only in the information that the tester owns. In both cases, this is dynamic testing. (Srividya, 2010):

In static testing, the program code is not executed - the program analysis is based on the source code, which is manually calculated or analyzed by special tools. In some cases, not the source, but an intermediate code (such as a bytecode or a code on MSIL) is analyzed. Also, static testing includes the testing of requirements, specifications, documentation.

Regression testing

After making changes to the next version of the program, regression tests confirm that the changes made did not affect the performance of the rest of the application's functionality. Regression testing can be performed both manually and by means of test automation.

Test scenarios

Testers use test scenarios at different levels: both in component, and in integration and system testing. Test scenarios are usually written to check the components in which the probability of failures is greatest or the error found on time can be costly.

Testing the "white box" and "black box"

Depending on the access of the test developer to the source code of the tested program, they distinguish "white box testing" and "black box testing". When testing a white box (also known as a transparent box), the test developer has access to the source code of the programs and can write code that is associated with the libraries of the software being tested. This is typical for component testing, in which only certain parts of the system are tested. It ensures

that the components of the structure are workable and stable, to a certain extent. When testing a white box, code coverage metrics or mutational testing are used. When testing a black box, the tester has access to the program only through the same interfaces as the customer or user, or through external interfaces that allow another computer or another process to connect to the system for testing. For example, a testing component can virtually automatically press keys or mouse buttons in a program under test using a process interaction mechanism, with certainty whether everything is right, that these events cause the same response as real keystrokes and mouse buttons. Typically, the testing of the black box is conducted using specifications or other documents that describe the requirements for the system. Typically, in this type of testing, the coverage criterion consists of covering the input data structure, covering the requirements and covering the model (in model-based testing). When testing a gray box, the test developer has access to the source code, but when performing tests directly, access to the code is usually not required.

If "alpha" and "beta testing" refer to stages before the release of the product (and also implicitly to the volume of the testing community and limitations on the testing methods), testing the "white box" and "black box" relates to the ways in which the tester achieves the goal.

Beta testing is generally limited to the black box technique (although a constant part of the testers usually continues testing the white box parallel to the beta test). Thus, the term "beta testing" may indicate the status of the program (closer to release than "alpha"), or it may indicate a certain group of testers and the process performed by this group. That is, the tester can continue to work on testing the white box, although the program is already a "beta stage," but in this case it is not part of the "beta" test (Khan, 2011).

Figure 1 depicts working software testing workflow. There are the long run and cosset imperatives to product testing, exhaustive testing might not be performed. Hence, those mechanization for testing may be required as of latest a lot of tools and methods are utilized to mechanize the steps. For a successful testing, several steps have been followed; detect code paths for testing, produce path test data suit; test procedure on the software under test (SUT) for test data; test assessment and manufacture quality model.

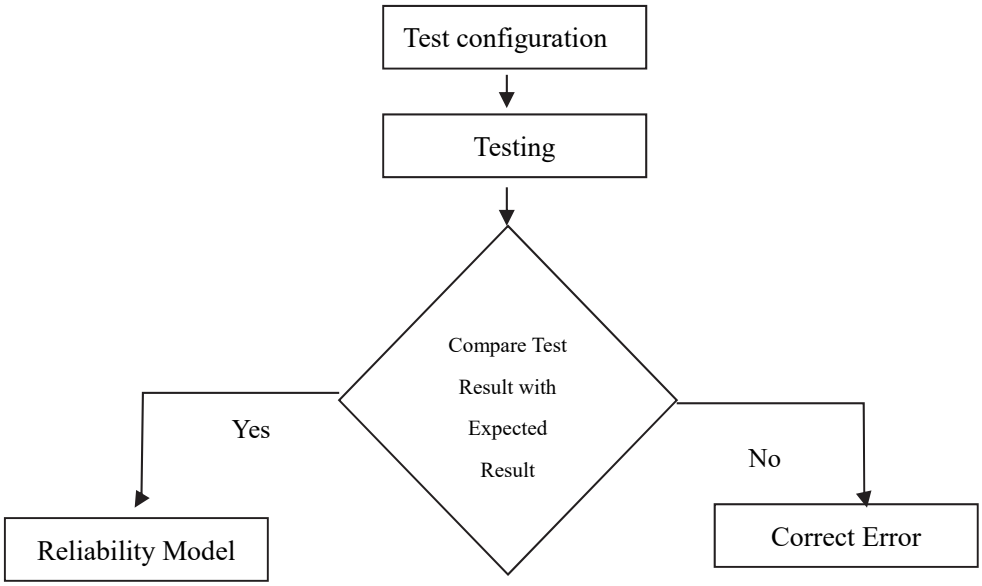

Figure 1. Software testing workflow

Efficient testing cover as much as testing similarly as could be allowed over a period of time. for least cost, optimal paths and test data have to be detected from many paths and the prioritization of paths has to be applied with the goal that the majority of the defects need aid will be found in sooner steps. path testing is the extreme helpful methods in testing to detect faults in program modules (Kaiser, 2015).

3.2 Software Path Coverage Test

Coverage basic path testing techniques is the procedure where the group of program for basic path are being

considered as the test target. It takes test information for program input space, then considers those test data as input, to run the program and performs that basic path. Basic paths have three characteristics: every path is independent; every edge in the program is available; every paths in the program do not have a place with the path set. Might be acquired by paths linear operation in the basic path set. Essential path testing techniques and fault propagation path testing technique are two methods that can be used in sequential programs. Program path in reference test is a sequence of statements to be executed, depends on a set of control flow graph nodes. In the real testing, significantly over easy procedure, the paths number may be large. So it is required to limit the number of paths to a certain extent so that the program loops is executed just once. Fault propagation path is an approach that will depict the advancement for defects where errors happen in software nodes; they might bit by bit spread on different nodes. In the transform for errors spread, errors will decide the path in which error propagation possibility is bigger, where spread possibility might be taken from defects data historic or alternatively be assessed as stated by those parameters of the system (Yichen et.al, 2016).

4. Artificial Bee Colony

Artificial bee colony (ABC) algorithm is an optimization technique that simulates the foraging behavior of honey bees, and has been successfully applied to various practical problems. ABC belongs to the group of swarm intelligence algorithms and was proposed by Karaboga in 2005.

A set of honey bees, called swarm, can successfully accomplish tasks through social cooperation. In the ABC algorithm, there are three types of bees: employed bees, onlooker bees, and scout bees. The employed bees search for food around the food source in their memory; meanwhile they share the information of these food sources to the onlooker bees. The onlooker bees tend to select good food sources from those found by the employed bees. The food source that has higher quality (fitness) will have a large chance to be selected by the onlooker bees than the one of lower quality. The scout bees are few employed bees, which abandon their food sources and search new ones.

The ABC algorithm is a swarm based meta-heuristic algorithm that consists of three different types of bees:

1) Employed Bees

Each employed bee is assigned to a food source. It is responsible for collecting nectar from that food source and fly back to its hive to share the information of the food source, including location, profitability of the nectar in that food source, etc., with other honey bees who are unemployed.

2) Onlooker Bees

All onlooker bees are unemployed and waiting at the hive. The employed bees will carry out a process called “waggle dance” to share the information of its assigned food source with the onlooker bees. After that, each onlooker bee will choose a food source by probability. The more profitable the food source is, the higher chance the onlooker bees will choose that food source.

3) Scout Bees

If a food source does not have profitable nectar any more, the employed bees will abandon that food source and become scout bees. All scout bees are unemployed and will choose a new source near their hive randomly.

The ABC algorithm makes use of two characteristics of the foraging behavior: recruitment of foragers to rich food sources giving positive feedback and abandonment of poor sources by foragers leading to negative feedback (Karaboga, 2009). Its critical part is to repeat this foraging process in order to keep searching better food sources so the ABC algorithm is regarded as an iterative algorithm, and therefore a stopping criterion is applied to terminate the foraging process. The detailed ABC algorithm is as the following:

First, a certain number of food sources θ_i are randomly generated. Each employed bee is assigned to a food source and the fitness of each food source is evaluated. After that, each employed bee searches for a new food source f_i around a food source θ_i using a neighborhood operator and the fitness of f_i is evaluated. If the new source is fitter than that of the old one, it replaces the old source and the employed bee changes to exploit the new food source.

When the employed bees go back to their hive, it shares the information with the onlooker bees. Each onlooker bee chooses a food source θ_i by a roulette wheel selection method. The higher the fitness value of the food source, the larger the chance the food source is chosen. Then, each onlooker bee searches a new food source f_i near to its selected food source θ_i by a neighborhood operator and the fitness of the new food source f_i is evaluated. If the fitness value of the new one is better than that of the old one, it replaces the old food source. For a food source

that has more than one onlooker bee, the best new food source replaces the old one. If a food source has neighborhood operator applied for a certain number of times, called "Limit", it is expected that the quality of the food source cannot be improved. The food source is abandoned and the employed bee assigned to that food source becomes a scout bee and searches for a new food source randomly. Again, each employed bee is assigned to a food source. The whole foraging process is repeated. The foraging process terminates when the number of predefined "Maximum Cycle" is reached. See the flow chart in Figure 1.

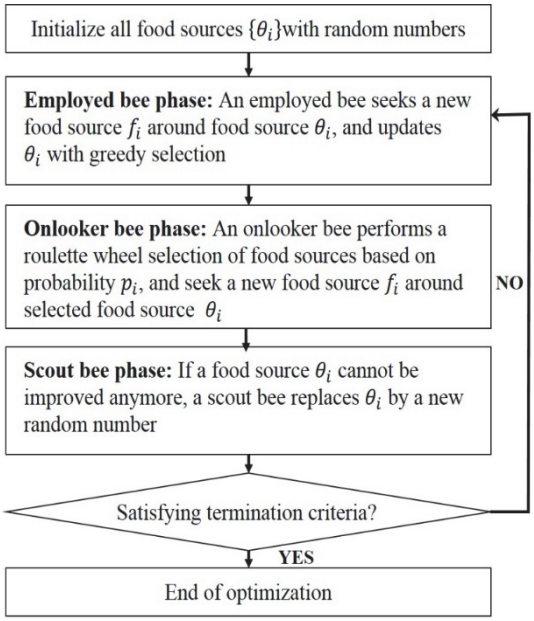

Figure 1. Flow chart for ABC algorithm

5. Testing ABC on the CEC2005 Benchmarks Functions

The CEC2005 benchmark set is classified into 2 main categories either Unimodal Functions (F1-F5) as shown in table 1 or Multimodal Functions (F6-F23) as shown in table 2. Multimodal Functions consists of Basic Functions (F6-F12), Expanded Functions (F13-F14) and Hybrid Composition Functions F15 to F23 combining multiple test problems into a complex landscape.

The mathematical formulations for Unimodal Functions are shown in table 3. Multimodal Functions mathematical formulations are shown in table 4.

Table 1. Classification of CEC2005 Benchmark Functions (Unimodal Functions)

Function	Unimodal Functions (1- 5)
1	F1: Shifted Sphere Function
2	F2: Shifted Schwefel's Problem 1.2
3	F3: Shifted Rotated High Conditioned Elliptic Function
4	F4: Shifted Schwefel's Problem 1.2 with Noise in Fitness
5	F5: Schwefel's Problem 2.6 with Global Optimum on Bounds

Table 2. Classification of CEC2005 Benchmark Functions (Multimodal Functions)

	Function	Multimodal Functions (6- 23)
Basic Functions	6	F6: Shifted Rosenbrock's Function
	7	F7: Shifted Rotated Griewank's Function without Bounds
	8	F8: Shifted Rotated Ackley's Function with Global Optimum on Bounds
	9	F9: Shifted Rastrigin's Function

Expanded Functions	10	F10: Shifted Rotated Rastrigin's Function
	11	F11: Shifted Rotated Weierstrass Function
	12	F12: Schwefel's Problem 2.13
	13	F13: Expanded Extended Griewank's plus Rosenbrock's Function (F8F2)
	14	F14: Shifted Rotated Expanded Scaffer's F6
Hybrid Composition Functions	15	F15: Hybrid Composition Function
	16	F16: Rotated Hybrid Composition space Function
	17	F17: Rotated Hybrid Composition Function with Noise in Fitness
	18	F18: Rotated Hybrid Composition Function
	19	F19: Rotated Hybrid Composition Function with a Narrow Basin for the Global Optimum
	20	F20: Rotated Hybrid Composition Function with the Global Optimum on the Bounds
	21	F21: Rotated Hybrid Composition Function
	22	F22: Rotated Hybrid Composition Function with High Condition Number Matrix
	23	F23: Non-Continuous Rotated Hybrid Composition Function

Table 3. Unimodal Functions mathematical formulation

Function	Formula
1.	$f1(x) = \sum^{n} x_i^2$
2.	$f2(x) = \sum_{i=1}^{n} \lvert x_i \rvert + \prod_{i=1}^{n} \lvert x_i \rvert$
3.	$f3(x) = \sum_{i=1}^{n} \left(\sum_{i=1}^{n} x_j \right)$
4.	$f4(x) = \max i\{\lvert x_i \rvert, 1 \le i \le n\}$
5.	$f5(x) = \sum_{i=1}^{n=1} [100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2]$

Table 4. Multimodal Basic Functions

6.	$f6(x) = \sum_{i=1}^{n} (x_i + 0.5)^2$
7.	$f7(x) = \sum_{i=1}^{n} ix_i^4 + random(0,1)$
8.	$f8(x) = \sum_{i=1}^{n} -x_i \sin\sqrt{\lvert x_i \rvert} * \sum_{i=1} ix_i^4 * random(0,1) *$
9.	$f9(x) = \sum_{i=1}^{n} [x_i^2 - 10\cos(2\pi x_i) + 10]$

10. $$f10(x) = -20\exp\left(-0.2\sqrt{\frac{1}{n}\sum_{x=i}^{n} x_i^2}\right) - \exp\left(\frac{1}{n}\sum_{i=1}^{n}\cos(2\pi x_i)\right) + 20 + e$$

11. $$f11(x) = \frac{1}{4000}\sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n}\cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$$

$$f12(x) = \frac{\pi}{n}\left\{10\sin(\pi y_1) + \sum_{i=1}^{n=1}(y_i - 1)^2[1 + 10\sin^2(\pi y_{i+1})] + (y_n - 1)^2\right\} + \sum_{i=1}^{n} u(x_i, 10,100,4)$$

12. $$y_i = 1 + \frac{x_i + 1}{4}$$

$$u(x_i, a, k, m) = \begin{Bmatrix} k(x_i - a)^m & x_i > a \\ 0 & -a < x_i < a \\ k(-x_i - a)^m & x_i < a \end{Bmatrix}$$

Table 5. Multimodal expanded Functions

13. $$f13(x) = 0.1\left\{\sin^2(3\pi x_1) + \sum_{i=1}^{n}(x_i - 1)^2[1 + \sin^2(3\pi x_i + 1)] + (x_n - 1)^2[1 + \sin^2(2\pi x_n)]\right\} + \sum_{i=1}^{n} u(x_i, 5,100,4)$$

14. $f14(x) = -\sum_{i=1}^{n}\sin(x_i).\left(\sin\left(\frac{ix_i^2}{\pi}\right)\right)^{2m}$, m=10

Table 6. Hybrid Composition Functions

15. $$f15(x) = \sum_{i=1}^{11}\left[a_i - \frac{x_1(b_i^2 + b_i x_2)}{b_i^2 + b_i x_3 x_4}\right]$$

16. $$f16(x) = 4x_1^2 - 2.1x_1^4 + \frac{1}{3}x_1^6 + x_1x_2 - 4x_2^2 + 4x_2^4$$

17. $$F17(X) = \left(x_2 - \frac{5.1}{4\pi^2}x_1^2 + \frac{5}{\pi}x_1 - 6\right)^2 + 10\left(1 - \frac{1}{8\pi}\right)\cos X_1 + 10$$

18. $$f18(x) = [1 + (x_1 + x_2 + 1)^2(19 - 14x_1 + 3x_1^2 - 14x_2 + 6x_1x_2 + 3x_2^2)]\, x\, [30 + (2x_1 - 3x_2)^2 x\, (18 - 32x_1 + 12x_1^2 + 48x_2 - 36x_1x_2 + 27x_2^2]$$

19. $$f19(x) = -\sum_{i=1}^{4} C_i \exp\left(-\sum_{j=1}^{3} a_{ij}(x_j - p_{ij})^2\right)$$

20. $$f20(x) = -\sum_{i=1}^{4} C_i \exp\left(-\sum_{j=1}^{6} a_{ij}(x_j - p_{ij})^2\right)$$

21. $f21(x) = -\sum_{i=1}^{5}[(X - a_i)(X - a_i)^T + C_i]^{-1}$

22. $f22(x) = -\sum_{i=1}^{7}[(X - a_i)(X - a_i)^T + C_i]^{-1}$

23. $f23(x) = -\sum_{i=1}^{10}[(X - a_i)(X - a_i)^T + C_i]^{-1}$

6. Experimental Results and Analysis

The experiments were performed using 5 different programs. Some of them are taken from a benchmark functions shown in table7. The fitness value is a numerical value that expresses individual quality compared with current local solution in order to search for the optimal least fitness value. Result with lowest fitness value will be the optimal solution from the programs.

Korel's path distance relation for every variable was used to compute fitness value. The fitness path distance is the sum of variables fitness value along the path. The process to compute the fitness value is illustrated as the following:

- Random set of test cases are being generated. Enhancement of the current solution is being applied by utilizing the randomly selected cases.
- Calculate the fitness value for every candidate solution.
- For every particular swarm a fitness value is allocated, each swarm search for the local minimum value within the search area in order to find / get better value, the new value is kept and replaces the old value.
- allocate candidate solution from better fitness to worst fitness
- Onlooker phase starts from the best fitness solution.
- If termination conditions are found then the search finish, else onlooker local search issue to enhance candidate solution fitness.
- If the phase ends without reaching the ending conditions, the phase is initiated again to substitute sources that reach the maximum number of tries.

7. Case Studies

Table 7. Programs used as case studies

Program1	Program2
If (j >=80)	while (j >=75)
{….. }	{….. }
Else if (k >= 70)	while (k >= 65)
{….. }	{….. }
Else If (x >=60)	while (x >=55)
{….. }	{….. }
Else If (y>=50)	while (y>=45)
{….. }	{….. }
Else If (z>=25)	while (z>=35)
{….. }	{….. }

Experiments and results for program 1

In this test case a program with 5 variables (x,y,z,j,k) was used. According to Korel branch distance relation, the distance at first variable J would be zero if j - 80 >= 0, the distance at second variable K would be zero if K - 70 >=

0, at the third variable X the distance would be 0 if x - 60 >= 0, at the next variable Y the distance would be 0 if y - 50 <= 0, and finally, the last variable will be 0 if z – 25.

Table 8. Program1 case study

	J	K	x	y	z	j korel branch distance	k korel branch distance	X korel branch distance	Y korel branch distance	z korel branch distance	Fitness
1	91	50	75	100	54	11	0	15	50	29	105
2	89	64	84	68	66	9	0	24	18	41	92
3	81	87	71	82	87	1	17	11	32	62	123
4	62	72	89	52	99	0	2	29	2	74	107
5	84	56	72	86	79	4	0	12	36	54	106
6	70	91	84	92	59	0	21	24	42	34	121
7	77	67	71	72	88	0	0	11	22	63	96
8	76	97	61	84	65	0	27	1	34	40	102
9	53	65	72	79	85	0	0	12	29	60	101
10	90	56	80	80	70	10	0	20	30	45	105
11	96	66	62	53	75	16	0	2	3	50	71
12	99	63	61	88	85	19	0	1	38	60	118
13	55	87	90	55	85	0	17	30	5	60	112
14	78	95	60	72	93	0	25	0	22	68	115
15	68	98	63	93	93	0	28	3	43	68	142
16	76	97	85	69	51	0	27	25	19	26	97
17	99	57	79	84	68	19	0	19	34	43	115
18	59	92	85	75	84	0	22	25	25	59	131
19	100	93	100	59	82	20	23	40	9	57	149
20	59	67	72	94	76	0	0	12	44	51	107
21	67	87	62	58	59	0	17	2	8	34	61
22	100	80	76	90	69	20	10	16	40	44	130
23	66	78	95	58	82	0	8	35	8	57	108
24	50	54	54	66	86	0	0	0	16	61	77
25	63	78	89	98	51	0	8	29	48	26	111

The fitness value used for path of the program1 is 61, which is the sum of the previously mentioned distances calculated using the following equation (1):

$$F= (J-80) + (K-70) +(X-60) +(Y-50) + (Z-25) \quad (1)$$

Experiments and results for program 2

In the second test case, program 2 with 5 variables (x,y,z,j,k) was used. Again, Korel branch distance relation was used. The distance at first variable J would be zero if j - 75 >= 0, the distance at second variable K would be zero if K - 65>= 0, at the third variable the distance would be 0 if x - 55 >= 0, at the next variable the distance would be 0 if y - 45 <= 0, the last variable 0 if z – 35.

Table 9. Program2 case study

J	K	x	y	z	j korel branch	k korel branch	X korel branch	Y korel branch	z korel branch	Fitness

						distance	distance	distance	distance	distance	
1	86	84	52	94	78	11	19	0	49	43	122
2	92	57	59	86	100	17	0	4	41	65	127
3	74	54	62	63	83	0	0	7	18	48	73
4	86	57	94	75	71	11	0	39	30	36	116
5	66	96	83	52	81	0	31	28	7	46	112
6	70	98	69	58	95	0	33	14	13	60	120
7	81	88	92	92	80	6	23	37	47	45	158
8	53	69	86	91	100	0	4	31	46	65	146
9	53	81	64	74	68	0	16	9	29	33	87
10	89	79	53	82	94	14	14	0	37	59	124
11	51	58	88	97	84	0	0	33	52	49	134
12	78	66	99	54	84	3	1	44	9	49	106
13	86	61	66	73	89	11	0	11	28	54	104
14	100	57	98	51	57	25	0	43	6	22	96
15	77	68	78	63	100	2	3	23	18	65	111
16	91	80	92	68	57	16	15	37	23	22	113
17	78	76	82	62	68	3	11	27	17	33	91
18	86	65	60	67	56	11	0	5	22	21	59
19	72	73	72	67	72	0	8	17	22	37	84
20	87	73	100	84	67	12	8	45	39	32	136
21	61	64	92	61	50	0	0	37	16	15	68
22	95	50	69	78	68	20	0	14	33	33	100
23	62	53	52	51	83	0	0	0	6	48	54
24	52	91	54	60	73	0	26	0	15	38	79
25	94	96	83	80	100	19	31	28	35	65	178

The fitness value used for path of program 2 is 54, which is the sum of the previously mentioned distances calculated according to the following equation:

$$F = (J-75) + (K-65) + (X-55) + (Y-45) + (Z-35) \qquad (2)$$

8. Conclusions

This paper presented Artificial Bee Colony Algorithm based (ABC) to test data generation for software structural testing particularly, path testing. It applies Meta heuristic measure which detect optimal result to find fitness stop if result might be found. Efficiently optimal fitness value among a set of values was found, from the local minimal fitness values among search. The presented approach have been tested to by execute ABC algorithm over creating testing data for the criteria of path coverage testing. The results show the success and the ability of the ABC algorithm in software path testing by finding the optimal fitness values.

References

Al Khattab, A., Al-Shalabi, H., Al-Rawad, M., Al-Khattab, K., & Hamad, F. (2015). The Effect of Trust and Risk Perception on Citizen's Intention to Adopt and Use E-Government Services in Jordan. *Journal of service science and management, 8*(03), 279.

Alhadidi, B., & Fakhouri, H. N. (2008, August). Automation of iron difficiency anemia blue and red cell number calculating by intictinal villi tissue slide images enhancing and processing. In Computer Science and Information Technology, 2008. ICCSIT'08. International Conference on (pp. 407-410). IEEE.

Al-Sayyed, R. M., Fakhouri, H. N., Rodan, A., & Pattinson, C. (2017). Polar Particle Swarm Algorithm for Solving

Cloud Data Migration Optimization Problem. *Modern Applied Science, 11*(8), 98.

Al-Shwabkah, Y., Hamad, F., Taha, N., & Al-Fadel, M. (2016). The integration of ICT in library and information science curriculum analytical study of students' perception in Jordanian Universities. *Library Review, 65*(6/7), 461-478.

Andrea, A. (2012). A Theoretical and Empirical Analysis of the Role of Test Sequence Length in Software Testing for Structural Coverage. *IEEE Transactions on software engineering, 38*(3), 2012

Andreas, H., Jochen, Q., & Matthias, W. (2016). Experience Report: White Box Test Case Generation for Automotive Embedded Software. IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops.

Danilov, A. I., Khomonenko, A. D., & Danilov, A. (2015). Dynamic Software Testing Models, 978-1-4673-6961-9/72 15©2015 IEEE.

Duan, H., & Qiao, P. (2014). Pigeon-inspired optimization: A New swarm intelligence optimizer for air robot path planning. *International Journal ofIntelligent Computing and Cybernetics, 7,* 24-37.

Hamad, F., & Adwan, O. (2018). Policy Based Approach for Information Transfer over Mobile ad hoc Network using Messages Privacy Control. *Modern Applied Science, 12*(5), 22.

Hamad, F., & Alawamrah, A. (2018). Measuring the Performance of Parallel Information Processing in Solving Linear Equation Using Multiprocessor Supercomputer. *Modern Applied Science, 12*(3), 74.

Hamad, F., Tbaishat, D., & Al-Fadel, M. (2017). The role of social networks in enhancing the library profession and promoting academic library services: A comparative study of the University of Jordan and Al-Balqaa'Applied University. *Journal of Librarianship and Information Science, 49*(4), 397-408.

Hudaib, A. A., Fakhouri, H. N., Al Adwan, F. E., & Fakhouri, S. N. (2016). A Survey about Self-Healing Systems (Desktop and Web Application). *Communications and Network, 9*(01), 71.

Indrajit, N. T., & Siddharth, A. P. (2016). Voltage Stability Enhancement and Voltage Deviation Minimization Using Multi-Verse Optimizer Algorithm. International Conference on Circuit, Power and Computing Technologies [ICCPCT].

Kaabneh, K., Abu-Hammad, E., & Hamd, F. (2007, November). Enhanced Skin Detection Technique Using Block Matching. In Signal Processing and Communications, 2007. ICSPC 2007. IEEE International Conference on (pp. 21-24). IEEE.

Krishna, K., Mohan, A. K. V., & Srividya, A. (2010). Software Reliability Estimation Through Black Box and White Box Testing at Prototype Level, 978-1-4244-8343-3/10©2010 IEEE.

Lei, Zh., Bei-Bei, Y., Junpeng, L., Kai-Yuan, C., Stephen, S. Y., & Jia, Yu. (2014). A History-Based Dynamic Random Software Testing, IEEE 38th Annual International Computers, Software and Applications Conference Workshops.

Lu´ıs Pina, & Michael, H. (2016). Tedsuto: A General Framework for Testing Dynamic Software Updates. IEEE International Conference on Software Testing, Verification and Validation.

Manish, J., & Dinesh, G. (2016). Aspect oriented programming and types of software testing, Second International Conference on Computational Intelligence & Communication Technology.

Md. Alauddin. (2016). Mosquito Flying Optimization (MFO), International Conference on Electrical, Electronics, and Optimization Techniques (ICEEOT) – 2016, 978-1-4673-9939-5/16 ©2016 IEEE.

Mirjalili, S., & Mirjalili, A. S. M. (2014). Lewis, Grey Wolf Optimizer. *Advances in Engineering Software, 69*, 46-61.

Mirjalili, S., Mirjalili, S. M., & Hatamlou, A. (2016). Multi-verse optimizer: a nature-inspired algorithm for global optimization. *Neural Computing and Applications, 27,* 495-513.

Mumtaz, A. K., & Mohd, S. (2011). Analysis of Black Box Software Testing Techniques: A Case Study, 978-1-4673-0098-8/11©2011 IEEE.

Nicha, K., & Jirapun, D. (2010). A Test Case Generation Technique and Process CEUR Workshops proccedings, 646.

Parampreet, K., & Ashish, Kr. L. (2016). An Approach to Improve Test Path Generation: Inclination towards Automated Model-based Software Design and Testing, 978-1-5090-1489-7/16©2016 IEEE.

Seyedali, S. M. M., & Abdolreza, H. (2015). A Multi-Verse Optimizer: a nature-inspired algorithm for global optimization, Springer, Neural Comput & Applic. https://doi.org/10.1007/s00521-015-1870-7

Shivkumar, H. T. (2012). Software Testing Techniques. *Int. journal of Advanced Research in Computer Science and Software Engineering, 2*(12).

Sumon, B., M. S. K., & Mamun, S. A. (2015). Applying Ant Colony Optimization in Software Testing to Generate Prioritized Optimal Path and Test Data. Int'l Conf. on Electrical Engineering and Information & Communication Technology (ICEEICT).

Tanachapong, W., Sirapat, C., & Khamron, S. (2016). Multilevel Thresholding Selection Based on Chaotic Multi-Verse Optimization for Image Segmentation. 13th International Joint Conference on Computer Science and Software Engineering (JCSSE).

Wang, K., & Wang, Y. (2016). Software Test Case Generation Based on the Fault Propagation Path Coverage, IEEE 2016, 978-1-5090-029-8.

Xiao, Z., & Jianghua, Zh. (2014). IEEE Workshop on Advanced Research and Technology in Industry Applications (WARTIA).

Zaghoul, F. A., Rababah, O., & Fakhouri, H. (2014, March). Website search engine optimization: Geographical and cultural point of view. In Computer Modelling and Simulation (UKSim), 2014 UKSim-AMSS 16th International Conference on (pp. 452-455). IEEE.