Intelligent Automation & Soft Computing
DOI:10.32604/iasc.2021.017239
Article

Tech Science Press

Optimizing the Software Testing Problem Using Search-Based Software Engineering Techniques

Hissah A. Ben Zayed and Mashael S. Maashi*

Department of Software Engineering, College of Computer and Information Sciences, King Saud University, Riyadh, 11451, Kingdom of Saudi Arabia
*Corresponding Author: Mashael S. Maashi. Email: mmaashi@ksu.edu.sa
Received: 25 January 2021; Accepted: 03 April 2021

Abstract: Software testing is a fundamental step in the software development lifecycle. Its purpose is to evaluate the quality of software applications. Regression testing is an important testing methodology in software testing. The purpose of regression testing is to validate the software after each change of its code. This involves adding new test cases to the test suite and running the test suite as the software changes, making the test suite larger. The cost and time of the project are affected by the test suite size. The challenge is to run regression testing with a smaller number of test cases and larger amount of software coverage. Minimization of the test suite with maximization of the software coverage is an NP-complete problem. Search-based software engineering is an important topic in software engineering, which addresses software engineering optimization problems to find the optimal/approximate solution of the given problem. This study investigated an approach to reducing the regression testing effort and saving time. It also solved the regression testing optimization problem by achieving the maximum test suite coverage with the minimum test suite size. Several experiments were conducted to obtain the optimal solutions for the regression testing problem. We propose an optimization methodology that combines a genetic algorithm and a greedy algorithm to optimize regression testing by respectively maximizing the software test coverage and minimizing the test suite size. The proposed methodology can conveniently deliver fault-free, fully covered, and powerful programs for mission-critical functions. It can be applied to test a real-time system that has high requirements for reliability, security, and safety.

Keywords: Software testing; test suite coverage; regression testing; search-based software engineering optimization; genetic algorithm; greedy algorithm; optimization

1 Introduction

Software testing is an important aspect of the software development lifecycle. Software testing refers to identifying the test cases that detect software errors (SUT) [1]. The set of test cases for SUT is called a *test suite*. Agile methodology [2] imposes constraints and limitations on the software development lifecycle, such as a low execution cost, low budget, and rapid deployment, which increase the test case numbers run in the maintenance and development stages, and thus increase the test suite size. Software maintenance [3] is an essential element in the development lifecycle. The changes made by software engineers in response to a new feature requested or to fix a bug may affect the whole system. Regression testing [4] plays an important role in software testing; it suppresses changes throughout the program. Regression testing is a critical task in software testing, and it requires significant effort. The most important element in regression testing is the test suite [5,6]. Issues and bugs discovered by regression testing can reduce the software budget. The reduction of the test suite is one of the most crucial techniques used to perform regression testing. It is an NP-complete problem [7]. Large test cases cannot be completed in a reasonable amount of time. The effort spent on testing can be optimized by selecting fewer test cases that are expected to find more defects. Software engineering problems can be solved by search-based software engineering (SBSE) techniques [8]. These methods are based on meta-heuristics, and they address software engineering problems. In 1976, the earliest study was proposed to optimize a software engineering problem reported in the field of software testing [9]. In the context of artificial intelligence, search-based methodologies are mainly identified based on their ability to find the optimal solution with an intensification of domain information as well as diversification on the solution space. The more complex the SE optimization problem, the more advanced the optimization method is required [10]. Therefore, SBSE approaches must be able to manage multiple objective/criteria constraints, uncertainty, and dynamic optimization problems. Genetic algorithms (GAs) and greedy algorithms have been successfully applied to different search and optimization problems. The success of those algorithms stems from their flexibility, simplicity, accuracy, and broad applicability [9]. The ease of identifying the limited number of genetic algorithm parameters is a reason for using GA. Therefore, we employed a combination of the GA and a greedy algorithm to optimize the regression testing problem.

We addressed two main issues: (i) How can we get the maximum coverage with the minimum number of test cases in a test suite for regression testing? and (ii) How can we adopt optimization algorithms to solve this problem? To answer these questions, we propose a framework that combines a genetic algorithm and a greedy algorithm to optimize the test suite by maximizing the test case coverage and minimizing the number of test cases in the regression testing. The proposed methodology aims to help the software tester reduce the regression testing effort and optimize the regression test suite with the objective of maximizing the test suite coverage and minimizing the test suite size. The major contributions of this study are summarized as follows.

- We designed a framework for optimizing regression testing to help the software tester minimize the effort and time needed for regression testing.
- We formulated a regression testing optimization problem.
- We applied a genetic algorithm to maximize the text case coverage of regression testing.
- We proposed a greedy-based method to minimize the size of the test suite for regression testing.

The rest of this article is structured as follows. The background and related work are provided in Section 2. Section 3 presents the proposed optimization methodology that addresses the regression testing problem. Section 4 describes the experimental settings. The results and discussion are presented in Section 5. Section 6 provides the conclusion and suggestions for further research.

2 Background and Related Works

2.1 Meta-Heuristic Techniques

In general, optimization methods are classified into two types: classical methods and meta-heuristics [11]. Meta-heuristic techniques are designed to obtain the optimal/near-optimal solutions of the optimization problems using heuristic information regarding the search area [12,13]. This study focuses on two meta-heuristics: GAs [14] and greedy algorithms [15]. GAs are global search heuristics that are widely applied to discrete optimization. The behavior of a GA is difficult to understand but easy to implement [14]. However, an appropriate chromosome representation is required for the solution domain. A fitness function for the given problem is needed for evaluating the domain of the solution. In a GA, solutions can be represented as real values, binary strings of 0 s and 1 s, and other encodings types. Applications of GAs are found in manufacturing, bioinformatics, engineering, mathematics, and other fields. A greedy algorithm [16] is a simple intuitive heuristic that is applied to solve optimization problems. It is based on the "next best" search philosophy. A greedy algorithm solves the problem iteratively by making the local/sub-optimal choice at each step until it obtains the overall optimal solution for the whole problem [15].

2.2 Regression Testing

Regression testing is a component of software testing. It is necessary to ensure that the change in the software has been validated. Discovering bugs and failures in the early stages of development is a mandatory, ongoing activity. IEEE [17] defined regression testing as, "*selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements*." The necessity for efficient methods for regression testing has increased with the growing application of iterative development processes and systematic usage in the software industry [18]. Regression testing is the most significant and time-consuming stage of software testing. Studies [19] show that more than 70% of the testing cost is regression testing, and about 55% of maintenance cost is spent for testing only. Other studies [20,21] have addressed different issues of regression testing for object-oriented software and conventional programs, including impact analysis and software change. The size of test suites grows whenever software is changed, and test cases are run to verify and validate old and new functionalities. The software code keeps changing based on the rectification of bugs/failures during the whole testing life cycle. To perform regression testing, we must select test cases for the SUT to perform regression testing. Then, either manual or automation tools are chosen to perform regression testing. After that, the regression tests are updated as required, and the reports of results are produced. One of the challenges in regression testing is the time consumed. A test suite needed for regression testing takes a long time to be completely executed and may involve running redundant test cases. The complexity of regression testing grows whenever the test suite updates according to changes on the tested software. Another challenge faced in regression testing is to validate and verify the SUT. It is necessary to decrease the size of the original test suite in regression testing. Therefore, a subset of test cases from the original test suite that cover the modifications are selected. This subset of test cases should have the capability to cover all the defects with fewer test cases. Regression test optimization can help software testers select an appropriate subset of test cases from the prime test suite.

2.3 Overview of Testing Optimization Problems

In the literature, several SBSE optimization techniques are employed to solve software problems that occur in different stages of software testing [22]. Search-based software testing (SBST) can be traced back to the 1970s. However, it has recently gained a renewed interest from researchers [23]. A broad range of results pertaining to SBST has been published. The most common SBST problems are test case selection, test data generation, non-functional testing problems, and test case prioritization. Various testing problems have been solved by applying SBST techniques [22]. In this section, we discuss only the test case selection problem, as the study's objective is to optimize the test suite for regression testing by selecting the minimum number of test cases with the maximum amount of software coverage.

One of the most substantial software selection testing problems is software testing optimization [21]. Prioritization and selection are the two main solutions for solving the testing problem. A prioritization algorithm for the test cases can be employed as a test case selection method. The minimization of the size of a test suite involves the selection of a minimum number of test cases. A study [24] presented exploratory findings from a genetic algorithm applied to breeding software testing. The guiding of fitness function can provide a focused search that produces a large number of localized test cases. It can be extended to produce random actions based on the test scenario. A large proportion or long series testing involves the reduplicated execution of many test cases. In [25], five algorithms for selecting test cases were compared, including a firewall [25], simulated annealing (SA) [26], reduction [27], slicing [28], and dataflow [29]. Test cases were used as a satisfaction function for the code. A study [25] compared qualitative and quantitative analysis, including test case number, maintenance type, requirements process, timely execution, completeness, accuracy, and testing level. The results showed that SA could obtain better solutions with respect to the metrics of accuracy and the number of test cases. In [30], NSGA-II algorithms were used to solve the multi-objective version of the selection test case problem. In experiments, four smaller programs were used in the space program (9,564 lines of code) and the Siemens suite (374, 412, 570, and 726 lines of code). NSGAII obtains the best performance and found the complete Pareto optimal front.

3 The Proposed Regression Testing Optimization Methodology

In this study, we propose a new methodology that optimizes the regression testing problem. The primary goal is to solve the regression testing problem with two objectives. The first objective is the maximization of the coverage by the test suite. The second objective is the minimization of the test suite. As the objectives of our regression testing problem are independent objectives, they are not in conflict with one another. We decided to solve them separately from each other [31]. The methodology framework consists of two stages, as shown in Fig. 1. In the first stage, the maximization coverage process deals with the first objective. The second stage is the minimization of the test suite process that deals with the second objective. We applied a GA in the first stage and a reduction greedy algorithm-based technique in the second stage.

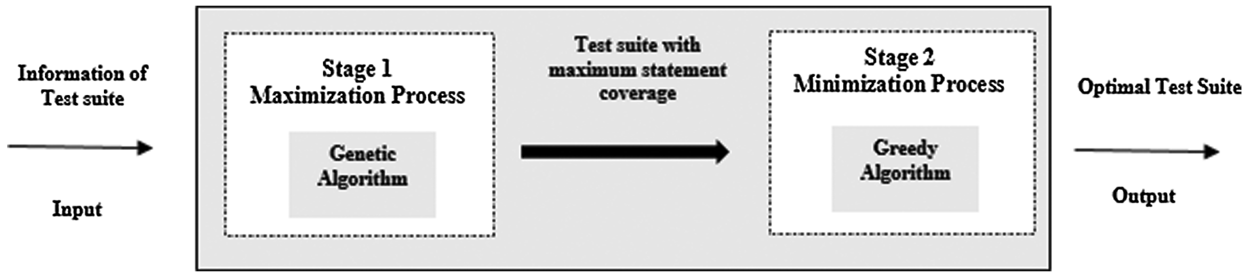
Information of Test suite
Input
Stage 1 Maximization Process
Genetic Algorithm
Test suite with maximum statement coverage
Stage 2 Minimization Process
Greedy Algorithm
Optimal Test Suite
Output

Figure 1: The proposed framework for the regression testing optimization methodology

The pseudocode of the regression testing optimization methodology is given in Fig. 2. The proposed framework comprises two stages: the first stage addresses the first objective, which is maximizing the coverage of the test suite by applying the genetic algorithm to provide a test suite with the maximum statement coverage (step 1). The input of the first stage is the information of the given test suite, including the total number of test cases, total number of statements, and total number of statements covered by each test case. The output of the first stage is the optimal test cases with maximum statement coverage, which serves as the input of the second stage. The second stage addresses the second objective, which is minimizing the size of the test suite by applying a reductive greedy-based technique to obtain the minimum number of test cases (step 2). Eventually, the output from the second stage is the final optimal test suite with the minimum size of the test suite that has the maximum statement coverage. Further details of the maximization and minimization processes are provided in Subsections 3.2 and 3.3, respectively.

```
Input: Test suite Information (TS, #TC test cases, #S statements, #NSC)
▷ TS is the test suite
▷#TC is number of test cases.
▷ #NSC is number of statements.
▷ #NSC is number of statements covered by each test case.
Begin
Stage 1: Maximization statement coverage using Genetic Algorithm
1.1:  while (! #TC)
1.2:     for each test case (chromosome)
1.4:           Evaluate test case
1.5:           repeat
1.6:             Select test cases  for recombination
1.7:             Apply genetics operators
1.8:             Evaluate test cases
1.9:       until (meet the termination condition)
1.10:    end for
1.11     Output TS`
         ▷ TS` is the test suite with the maximum covered by each test case.
Stage 2: Minimization the size of test suite TS` using Greedy Algorithm
2.1 while (TS` ≠ ∅)
2.2:        Select test cases from TS` that have the maximum statement coverage.
2.3:        Add test case to TS``
2.4:          Delete all test cases from TS` that are compatible with statement
coverage
2.7:     end while
Output: TS` the optimal test suite with the minimum #TC
End
```

Figure 2: Pseudocode of the regression testing optimization methodology

3.1 Problem Description and Formulation

To perform the regression testing, we ran the original test suite *TS*, which contains all the test cases *TC* that have several statement coverage *SC* in the SUT. In regression testing, running the original test suite is time-consuming and costly. To avoid this, we needed to select an optimal subset test suite *TS′* from the original test suite TS. The process is based on optimizing the original test suite *TS* with the objectives of maximizing the statement coverage SC of the software function and then minimizing the size of the test suite (TS).

A test suite *TS* contains the number of test cases $TS = \{tc_1, tc_2, ..., tc_n\}$, $TC_i \in TS$, where *n* is the size of *TS*; *SC* represents the total statement coverage (requirement) of the SUT; and *m* is the total number of the statement coverage by the test suite. Each TC_j has SC_j. *TS'* is a subset of *TS* that covers *SC′* of *SC* (see Tab. 1 for the regression testing notations). The regression optimization problem is formulated as follows:

$$Max\ F_1(x) = \sum\nolimits_{i=1}^{n} X_{tci} \tag{1}$$

$$X_{tci} = \frac{\sum\nolimits_{j=1}^{m} SC_j}{m} \tag{2}$$

$$Min\ F_2(x) = |TS'| \tag{3}$$

3.2 Maximizing the Software Coverage Based on GA

We employ a genetic algorithm to achieve the maximization of the software coverage in the regression testing. The genetic algorithm consists of five main phases, starting from the initial population followed by evaluating the fitness function and applying the GA operators of crossover and mutation. In the beginning, the initial population *TSs* is built up randomly with specific population size. Every chromosome (solution) represents a test suite *TS* that contains *n* number of genes that represent a test case *TC*. Each test case *TC* covers *m* number of statements SC. All solutions are encoded in binary coding in a way that represents the characteristics of our solution. Every chromosome represents a test. The binary code for each test case (gene) is represented by the status of the statement coverage, which is 1 if the test case is covered and 0 otherwise. Each candidate solution in the population is evaluated using Eq. (2). On the basis of the fitness function, the fitter chromosome has a bigger probability of being selected. Thus, the test cases with maximum coverage paths are selected. Then, a new population is generated by applying the GA operators (mutation and crossover). A single point of crossover is applied to generate a new child (new solutions). Steps 1.1–1.9 in Fig. 2 are repeated till a termination condition is met. Eventually, the optimal test suite TS′ is obtained.

Table 1: Notations for problem formulation

Notations	Descriptions
$\boldsymbol{TS}$	The original test suite
$\boldsymbol{TS'}$	The optimal test suite with the maximum statement coverage
$\boldsymbol{TS''}$	The final optimal test suite with the minimum test cases that have the maximum statement coverage
$\boldsymbol{N}$	The total number of test cases (size of the test suite)
$\boldsymbol{M}$	The total number of the statements covered by the test suite
$\boldsymbol{sc_i}$	Statement coverage by test case i
$\boldsymbol{SC}$	The total statement coverage (requirement)
$\boldsymbol{TC}$	Test cases

3.3 Minimizing the Size of the Test Suite Based on a Greedy Algorithm

In this section, we deal with the objective of minimizing the number of test cases in the regression testing. To reduce the size of the test suite, we propose a reduction technique based on the greedy algorithm. The pseudocode of the reduction greedy-based technique is shown in Fig. 3.

```
Let $TS`$ be the set of all test cases $TC_n$
Let $TS``$ be the minimized test suite that covers the same statements as $TS`$
$TS``$ = empty
    While ($TS` \neq \varnothing$)
        DO
        Select $TC_i$ in TC that have the maximum $SC_j$
        Add $TC_i$ to $TS``$
        Delete all $TC$ from $TS`$ that are compatible with $TC_i$
    End While
Return $TS``$ as an optimized solution
```

Figure 3: Pseudocode of the proposed reduction greedy-based technique

The optimal test suite TS' derived from the genetic algorithm (stage 1) is used as an input for the reduction process. The test cases TC will be minimized by removing and filtering the test suite TS based on statements covered SC by the test cases. The proposed reduction technique chooses a subset of the original test suite that has the maximum coverage TS′. Initially, the test suite consists of TS′. The optimal solution TS' is mapped with test cases and the covered statements by each test case and inserted into the covered matrix, as shown in Tab. 2. The coverage matrix shows the relationship between the statements covered SC_j by the test cases Tc_i.

Table 2: A sample of test case coverage matrix

Test case number TC	1	2	3	4	5	6	…
Optimal solution TS'	0	1	1	1	1	1	…
Total statement SC	0	4	7	3	4	8	…

In Fig. 3, the proposed reduction technique starts with an empty test suite TS'', which has the minimum test cases covering the same statements as TS'. Then, the values for Tc_i are selected from TS' that have the maximum covered statements and are added to TS''. The selected values for Tc_i that are all compatible test cases with Tc_i from TS′ are deleted. This process is repeated until TS' is empty. TS'' is the optimal test suite solution.

4 Experimental Settings

Experiments using two datasets (discussed later in subsection 4.1) were conducted to measure the evaluation performance of our proposed regression testing framework that combines a GA with a greedy algorithm. We implemented our framework in Java 8 using IntelliJ IDEA 2019.1.3 (Community Edition) platform. We performed 20 independent runs on a Dual-Core Intel Core i5, 2.9 GHz, 8 GB of RAM running MacOS 10.15.3. GA was running for 25,000 evaluation functions. The population size of the GA was set to 100, and the number of generations was set to 250 in each run, with a random crossover point and mutation. The mutation probability rate was set to 0.07; the crossover probability rate was set to 1.0. All experimental parameters were set to values commonly used in the literature for similar optimization problems [21,32–39]. Tab. 3 lists the parameter settings.

Table 3: Parameter settings

Parameters	Setting
Population size	100
Stopping criteria	Maximum generation
Maximum generation	250
Selection type	Roulette wheel selection
Crossover rate	1.0
Crossover type	Random crossover
Mutation rate	0.07

In our experiment, we used two standalone applications from samate.nist. NIST provides a software assurance reference dataset that is useful for researchers and testing tool developers. The programs were originally developed by the Siemens Corporate Research Center, and are available on the Aristotle analysis system's webpage [40].

The first application (Wireshark) contains a test suite with 127 test cases and a coverage of 96 statements. The second application (Coffee MUD) contains a test suite with 188 test cases and a coverage of 368 statements. In our experiments, we ran the datasets 20 times for each application. Tab. 4 shows the summary of the regression testing datasets used in this study.

Table 4: Regression testing datasets

Dataset	Application	Total number of test cases	Total number of statements covered
1	Wireshark	127	96
2	Coffee MUD	188	368

5 Results and Discussion

Multiple experiments were conducted to obtain the optimal test suite for the regression testing. Our primary objectives were to maximize the coverage of the test suite and minimize the test case number in the test suite at hand. The minimum value, maximum value, average value, and standard deviation value of the obtained results for experiments for 20 runs on each application dataset are shown in Tabs. 5 and 6. The first column indicates the value of the first objective, namely maximizing the statement coverage using the GA. The second column indicates the value of the second objective, namely minimizing the number of test cases by applying the reduction of the greedy-based technique. The third column indicates the percentage of the statement coverage before applying the reduction on the test suite. The last column indicates the percentage of the statement coverage after applying the reduction on the test suite.

Table 5: Results on dataset 1

	Objective 1 Max SC	Objective 2 Min TC	Percentage of coverage before TC reduction %	Percentage of coverage after TC reduction %
Max	5.4300	0.7965	542.71%	110.42%
Min	4.6312	0.7661	462.50%	104.17%
Avg	4.8831	0.7791	488.28%	107.71%
Sdv	0.2078	0.0086	0.2077	0.0193

Table 6: Results on dataset 2

	Objective 1 Max SC	Objective 2 Min TC	Percentage of coverage before TC reduction %	Percentage of coverage after TC reduction%
Max	2.3500	0.5716	234.51%	102.45 %
Min	1.9500	0.4798	194.84%	100%
Avg	2.1260	0.5230	213.09%	101.11 %
Sdv	0.1137	0.0250	0.1134	0.0083

It is obvious from the results that the GA could effectively find the maximum statement coverage, as the percentage of the test suite coverage was more than 100%. This means that the test suite covered some statements in more than one test case. The redundancy of covering the statements is normal in the design of test cases. The test suite might include some test cases that cover the same statement more than one time. This duplication of the statement coverage needs to be prevented by reducing the test suite. Therefore, we can demonstrate that the reduction greedy-based technique has effectively reduced the size of the test suite by obtaining the minimum number of test cases. The percentage of test suite coverage after the reduction is almost equal to 100%, which means that the reduction technique is able to minimize the test size without affecting the test suite coverage obtained by the GA. Fig. 4 shows a boxplot analysis of the test suite on application dataset 1 and application dataset 2. Fig. 4(a) shows the analysis of the test suite on application dataset 2. From the right are the original test suite, the test suite after maximizing the coverage, and the test suite after minimizing the size (the optimal test suite). The boxplot analysis of the test suite on application dataset 2 is shown in Fig. 4(b). From the right are the original test suite, the test suite after maximizing the coverage, and the test suite after minimizing the size (the optimal test suite). The redundant coverage of the statements more than once is solved by our proposed reduction greedy-based algorithm.

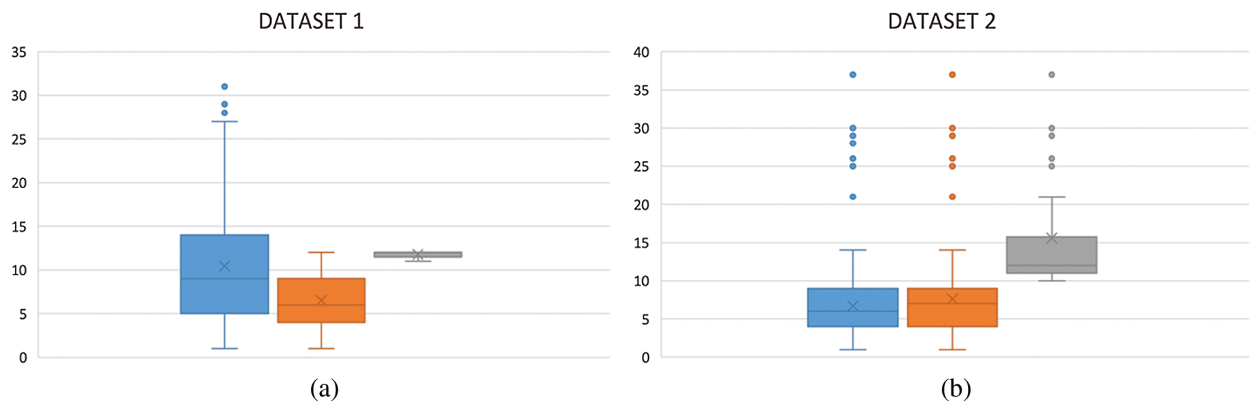

Figure 4: Boxplot analysis of the obtained optimal test suite on (a) dataset 1 and (b) dataset 2

The limitation of this study is the size of the test suite. This will affect the population size of the genetic algorithm, and a small test suite will not give the GA enough solution space to produce accurate results. However, we proposed and applied the solution of regression testing, which always has a large test suite size. This involves another limitation of the proposed solution that can be applied only for regression testing, regardless of the test level. The proposed solution requires some specific data from the test suites as inputs, like the number of test cases, number of statements covered, and number of statements covered by each test case, which we can take as complexity.

6 Conclusions and Future Work

The optimization of regression testing is crucial in the software development life cycle. It is essential to deliver fault-free, complete, and powerful software. In this study, a regression testing optimization methodology was proposed based on SBSE to find the optimal/approximate solution for the regression testing optimization problem. The proposed methodology combines a genetic algorithm and a greedy algorithm to optimize the regression testing by maximizing the software test coverage and minimizing the test suite size. The proposed framework is comprised of two stages. The first stage addresses the first objective, which is maximizing the coverage of the test suite by applying the genetic algorithm to provide a test suite with maximum statement coverage. The second stage addresses the second objective, which is minimizing the size of the test suite by applying a reduction greedy-based technique to obtain the minimum number of test cases. The regression testing optimization methodology can help testers and quality engineers in regression testing of software under testing to economize their efforts and improve software quality. The experimental results show that our proposed methodology can effectively produce fault-free, completely functional requirement coverage and efficient software in a short time and with little effort. It can be applied to test a real-time system that has high safety, security, and reliability requirements. The performance of the proposed methodology can be improved by hybridization of GAs and other algorithms such as local search algorithms. The application of other SBSE approaches to the regression testing optimization problem warrants further investigation.

Acknowledgement: The authors thank the Deanship of Scientific Research at King Saud University for funding and supporting this research through the initiative of the DSR Graduate Students' Research Support.

Funding Statement: The authors received funding from the Deanship of Scientific Research at King Saud University for this study (H. Ben Zayed). https://dsrs.ksu.edu.sa/ar/node/2228.

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

[1] G. Rothermel, M. J. Harrold, J. von Ronne and C. Hong, "Empirical studies of test-suite reduction," *Journal of Software Testing, Verification, and Reliability*, vol. 12, no. 4, pp. 219–249, 2002.

[2] G. Kumar and P. K. Bhatia, "Impact of agile methodology on software development process," *International Journal of Computer Technology and Electronics Engineering (IJCTEE)*, vol. 2, no. 4, pp. 6, 2012.

[3] M. Gopaul, R. Paavan and R. Vimla, "Review of software maintenance problems and proposed solutions in it consulting firms in Mauritius," *International Journal of Computer Applications*, vol. 156, no. 4, pp. 12–20, 2016.

[4] J. Gao, D. Gopinathan, Q. Mai and J. He, "A systematic regression testing method and tool for software components," in *30th Annual Int. Computer Software and Applications Conf. (COMPSAC'06)*, Chicago, IL, pp. 455–466, 2006.

[5] T. Koju, S. Takada and N. Doi, "Regression test selection based on intermediate code for virtual machines," in *Int. Conf. on Software Maintenance*, Amsterdam, Netherlands, pp. 420–429, 2003.

[6] Y. Chen, D. S. Rosenblum and K. Vo, "TESTTUBE: A system for selective regression testing," in *16th Int. Conf. on Software Engineering*, Sorrento, Italy, pp. 211–220, 1994.

[7] J. Bible, G. Rothermel and D. S. Rosenblum, "A comparative study of coarse- and fine-grained safe regression test-selection techniques," *ACM Transactions on Software Engineering and Methodology*, vol. 10, no. 2, pp. 149–183, 2001.

[8] M. Harman, S. A. Mansouri and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Computing Surveys*, vol. 45, no. 1, pp. 1–61, 2012.

[9] B. Korel, "Automated software test data generation," *IEEE Transactions on Software Engineering*, vol. 16, no. 8, pp. 870–879, 1990.

[10] A. S. Sayyad and H. Ammar, "Pareto-optimal search-based software engineering (POSBSE): A literature survey," in *2nd Int. Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE)*, San Francisco, CA, USA, pp. 21–27, 2013.

[11] M. Harman, "The current state and future of search-based software engineering," in *Future of Software Engineering (FOSE '07)*, Minneapolis, MN, USA, pp. 342–357, 2007.

[12] S. Russell and P. Norvig, "Problem solving," in *Artificial intelligence a modern approach*, 4th. edition, Upper Saddle River, NJ: Prentice Hall, pp. 63–110, 2010.

[13] C. Blum, "Metaheuristics in combinatorial optimization: Overview and conceptual comparison," *ACM Computing Surveys*, vol. 35, no. 3, pp. 268–308, 2003.

[14] D. Beasley, D. Bull and R. R. Martin, "An overview of genetic algorithms: Part 1, fundamentals," *University Computing*, vol. 15, no. 2, pp. 65–69, 1993.

[15] A. Malik, A. Sharma and M. V. Saroha, "Greedy algorithm," *International Journal of Scientific and Research Publications*, vol. 3, no. 8, pp. 1–5, 2013.

[16] Z. Li, M. Harman and R. M. Hierons, "Search algorithms for regression test case prioritization," *IEEE Transactions on Software Engineering*, vol. 33, no. 4, pp. 225–237, 2007.

[17] IEEE, "IEEE standard for software and system test documentation," in *IEEE Std 829-2008*, New York, United State, IEEE, pp. 1–150, 2008.

[18] P. K. Chittimalli and M. J. Harrold, "Re-computing coverage information to assist regression testing," in *IEEE Int. Conf. on Software Maintenance*, Paris, France, 164–173, 2007.

[19] B. Ba-Quttayyan, H. Mohd and F. Baharom, "Regression testing systematic literature review—A preliminary analysis," *International Journal of Engineering and Technology*, vol. 7, no. 19, pp. 418–424, 2018.

[20] D. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima *et al.,* "Change impact identification in object oriented software maintenance," in *Int. Conf. on Software Maintenance Software Maintenance*, Victoria, BC, Canada, Canada, pp. 202–211, 2007.

[21] P. B. Thakur and T. Verma, "A survey on test case selection using optimization techniques in software testing," *International Journal of Innovative Science, Engineering & Technology*, vol. 2, no. 4, pp. 593–596, 2015.

[22] J. Gao, D. Gopinathan, Q. Mai and J. He, "A systematic regression testing method and tool for software components," in *30th Annual Int. Computer Software and Applications Conf. (COMPSAC'06)*, Chicago, IL, pp. 455–466, 2006.

[23] M. Harman, "The current state and future of search-based software engineering," in *Future of Software Engineering (FOSE '07)*, Minneapolis, MN, USA, pp. 342–357, 2007.

[24] R. Kumar and S. Singh, "Breeding software test cases for pairwise testing using GA," *Global Journal of Computer Science and Technology*, vol. 10, no. 4, pp. 97–102, 2010.

[25] A. J. Nebro, J. J. Durillo, F. Luna, B. Dorronsoro and E. Alba, "A cellular genetic Algorithm for multiobjective optimization," *International Journal of Intelligent Systems*, vol. 24, no. 7, pp. 726–746, 2009.

[26] N. Pang, Y. You and Y. Shi, "Application research on the simulated annealing in balance optimization of multi-resource network planning," in *2008 Second Int. Sym. on Intelligent Information Technology Application*, Shanghai, China, pp. 113–117, 2008.

[27] K. Deb, A. Pratap, S. Agarwal and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, 2002.

[28] K. Deb, "Multi-objective optimisation using evolutionary algorithms: An introduction," in *Multi-objective Evolutionary Optimisation for Product Design and Manufacturing*, London: Springer, pp. 3–34, 2011.

[29] C. M. Fonseca and P. J. Fleming, "An overview of evolutionary algorithms in multiobjective optimization," *Evolutionary Computation*, vol. 3, no. 1, pp. 1–16, 1995.

[30] S. Yoo and M. Harman, "Pareto efficient multi-objective test case selection," in *Proc. of the ACM/SIGSOFT Int. Sym. on Software Testing and Analysis (ISSTA '07)*, London, United Kingdom, pp. 140, 2007.

[31] R. C. Purshouse and P. J. Fleming, "Conflict, harmony, and independence: Relationships in evolutionary multi-criterion optimisation," in *Evolutionary Multi- Criterion Optimization*, Berlin, Heidelberg: Springer, pp. 16–30, 2003.

[32] M. Mohammed, M. Abd Ghani, R. Hamed, S. Mostafa, M. Ahmad *et al.,* "Solving vehicle routing problem by using improved genetic algorithm for optimal solution," *Journal of Computational Science*, vol. 21, no. 10, pp. 255–262, 2017.

[33] M. A. Mohammed, M. K. Abd Ghani, N. Arunkumar, O. I. Obaid, S. A. Mostafa *et al.,* "Genetic case-based reasoning for improved mobile phone faults diagnosis," *Computers & Electrical Engineering*, vol. 71, no. 42, pp. 212–222, 2017.

[34] M. Mohammed, M. Ahmad and S. Mostafa, “Using genetic algorithm in implementing capacitated vehicle routing problem,” in *Proc. of the Int. Conf. on Computer & Information Science (ICCIS)*, pp. 257–262, 2012.

[35] Z. A. A. Alyasseri, A. T. Khader, M. A. Al-Betar, A. K. Abasi and S. N. Makhadmeh, “EEG signal denoising using hybridizing method between wavelet transform with genetic algorithm,” in *Proc. of the 11th National Technical Seminar on Unmanned System Technology 2019*, Singapore: Springer, pp. 449–469, 2020.

[36] K. Abdulkareem, M. Mohammed, S. Gunasekaran, M. Al-Mhiqani, A. Mutlag *et al.,* “A review of fog computing and machine learning: Concepts, applications, challenges, and open issues,” *IEEE Access*, vol. 7, pp. 153123–153140, 2019.

[37] S. Mostafa, M. Ahmad, A. Mustapha and M. Mohammed, “Formulating layered adjustable autonomy for unmanned aerial vehicles,” *International Journal of Intelligent Computing and Cybernetics*, vol. 10, no. 4, pp. 430–450, 2017.

[38] M. A. Mohammed, S. S. Gunasekaran, S. A. Mostafa, A. Mustafa and M. K. A. Ghani, “Implementing an agent-based multi-natural language anti-spam model,” in *2018 Sym. on Agent, Multi-Agent Systems and Robotics (ISAMSR)*, Putrajaya, Malaysia, pp. 1–5, 2018.

[39] S. Mostafa, S. Gunasekaran, A. Mustapha, M. Mohammed and W. Abduallah, “Modelling an adjustable autonomous multi-agent internet of things system for elderly smart home,” in *Int. Conf. on Applied Human Factors and Ergonomics*, Washington, USA: Springer, pp. 301–311, 2018.

[40] C. Oliveira, “Test Suites, Standalone apps,” (2015, October 28). [Online]. Available: https://samate.nist.gov/SARD/testsuite.php#applications.