DOI: http://dx.doi.org/10.26483/ijarcs.v8i7.4410

ISSN No. 0976-5697

Volume 8, No. 7, July – August 2017

International Journal of Advanced Research in Computer Science

RESEARCH PAPER

Available Online at www.ijarcs.info

# GREEDY DISCRETE ANT COLONY OPTIMIZATION FOR HIGH COVERAGE TEST SUITE GENERATION

T. Ramasundaram
Research Scholar,
Department of Computer Science
Periyar University, Salem
Tamil Nadu, India

Dr. V.Sangeetha
Assistant Professor and Head
Department of Computer Science
Periyar University Constituent College of Arts & Science
Pappireddipatti, Dharmapuri, Tamil Nadu, India

*Abstract:* Test suite optimization is significant problems in software engineering research to reduce testing cost of software program. Recently, few research works have been designed for test suite generation and reduction. However, there is a requirement for new technique to improve coverage rate of test suite generation and to remove redundant test cases. In order to overcome such limitations, a Greedy Discrete Ant Colony Optimization (GDACO) technique is proposed. The main objective of GDACO technique is to optimize the coverage capability of test suite generation. The GDACO technique initially used Ant Colony Optimization (ACO) algorithm for generating the test suites. The ACO algorithm selects test cases from test cases set based on trail's probability and subsequently update pheromone trails until the maximum iteration is reached. This process results in generation of test suites for testing software programs. After that, GDACO technique used Greedy Discretization algorithm to test suite optimization. The Greedy Discretization algorithm designed in GDACO technique chooses the test cases which cover most test requirements and removes redundant test cases in test suites. Therefore, GDACO technique finally obtains minimal cardinality subset of test suites with higher coverage rate of faults. The GDACO technique conducts the experimental works on parameters such as coverage rate, average rate of test suite reduction and execution time. The experimental result demonstrates that the GDACO technique is able to improve the coverage rate of software faults and also increases the average rate of test suite reduction when compared to state-of-the-art-works.

***Keywords:*** Test suite, Test case, software program, coverage, ACO algorithm, Greedy Discretization algorithm

## 1. INTRODUCTION

Software Testing is the process of testing the functionality of a software program through analytical methods. Test cases play a vital role in the process of software testing for determining the software quality. Therefore, test suites are generated with aid of test cases for testing software programs. Most of research works has been designed for test suite generation and optimization. But, optimizing the coverage capability of test suite generation was not sufficient.

A Memetic Algorithm was designed in [1] for whole test suite generation and optimization. However, coverage capability was not ensured. Test case minimization approach was developed in [2] to reduce the number of test cases in configuration-aware structural testing. However, this approach takes more computational time for achieving test suite minimization.

The efficiency of test suites generated was analyzed in [3] to fulfill four coverage criteria with aid of counter example-based test generation and a random generation approach. But, code coverage does not ensure test quality. Cuckoo Search Algorithm was designed in [4] to systematically reduce the number of test cases through considering the combinations of inputs.

A novel approach called whole test suite generation was developed in [5] for test data generation that covers all coverage goals and simultaneously reduces the total size of test case. Though, optimizing coverage capability of test suite was remained unsolved. An intelligent search-based method was intended in [6] to generate test cases automatically for coverage-oriented soft-ware testing. This method provides better performance in terms of test coverage and the number of test cases. But, test coverage was not at required level.

A Tabu Search hyper-heuristic strategy was presented in [7] for t-way test suite generation. However, test suite optimization was remained unaddressed. A test-suite-generation approach was developed in [8] for efficiently achieving complete multi-goal test-coverage of product-line implementations. However, optimization criteria for ordering of test goals were not considered.

A Parallel Genetic Algorithm Based on Spark was employed in [9] for Pairwise Test Suite Generation and to reduce test suite size. But, finding a smaller test suite size was remained unsolved. A novel regression test selection approach was designed in [10] based on analysis of state and dependence models of components to generate a regression test suite. However, execution time was more. A different techniques designed for test suite minimization was analyzed in [11] for enhancing the testing process of test suites and achieving all the testing requirements.

Based on the above mentioned techniques and methods presented, Greedy Discrete Ant Colony Optimization (GDACO) technique is developed. The research objective of GDACO technique is formulated as follows,

- To optimize the coverage capability of test suite generation, GDACO technique is introduced.
- To generate the test suites for testing software programs, Ant Colony Optimization (ACO) algorithm is used in GDACO technique.
- To perform test suite reduction, Greedy Discretization Algorithm is employed in GDACO technique.

The rest of this paper is organized as follows. Section II explains Greedy Discrete Ant Colony Optimization

(GDACO) technique with the assist of architecture diagram. Section III and Section IV explains the experimental settings and details performance analysis with the aid of parameters. Section V describes the related works. Finally, Section VI concludes this paper.

## 2. A GREEDY DISCRETE ANT COLONY OPTIMIZATION TECHNIQUE

In software engineering, a test suite contains set of test cases for testing a software program to prove that it has some specific set of characteristics. Test suite reduction technique is required to reduce the cost of software testing by removing the redundant test cases from the test suite. Therefore, a Greedy Discrete Ant Colony Optimization (GDACO) technique is designed to improve coverage capability of test suite and to reduce redundancy of test cases for test suite optimization. The GDACO technique generates the test suites for evaluating the efficiency of software programs with aid of Ant Colony Optimization (ACO) algorithm. The test suites consist of numerous test cases for testing the software programs. After generating the test suites, Greedy Discretization algorithm is used to select a minimal subset of a test suite that covers all test requirements. Thus, GDACO technique reduces the test cases in test suites through choosing a set of test cases that are fulfill the testing criteria. As a result, GDACO technique improves coverage rate of test cases for discovering the more faults in software programs and also reduces test cases redundancy of software testing. This resulting in reduced test cost for improving software quality. The overall architecture diagram of Greedy Discrete Ant Colony Optimization technique for test suite minimization is shown in below Figure 1.

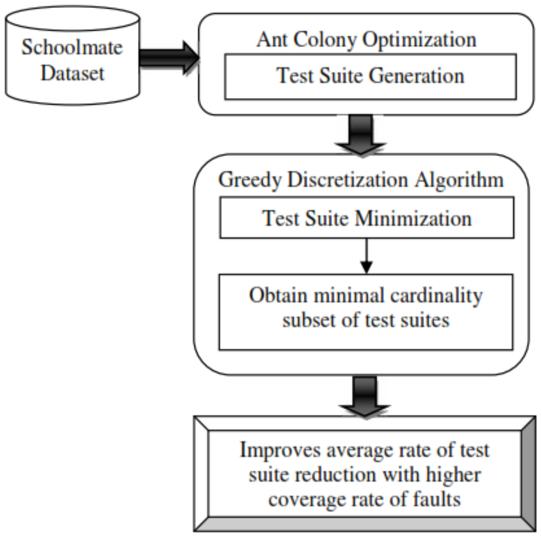
Schoolmate Dataset
Ant Colony Optimization
Test Suite Generation
Greedy Discretization Algorithm
Test Suite Minimization
Obtain minimal cardinality subset of test suites
Improves average rate of test suite reduction with higher coverage rate of faults

Figure 1. **Architecture of Greedy Discrete Ant Colony Optimization Technique**

As shown in Figure 1, GDACO technique takes schoolmate data set as input. Next, GDACO technique applies Ant Colony Optimization algorithm for generating test suites. Then, GDACO technique used Greedy Discretization Algorithm for test suite minimization which resulting in minimal cardinality subset of test suites for finding the more number of faults in schoolmate data set. Therefore, GDACO technique increases average rate of test suite reduction with higher coverage rate of faults. The detailed explanation about GDACO technique is described in forthcoming sections.

### *A. Test Suite Generation Using Ant Colony Optimization*

The GDACO technique used Ant Colony Optimization (ACO) algorithm to generate a set of test suites for achieving high fault coverage in a short time. Ant colony optimization (ACO) is a metaheuristic which discovers solutions for complex combinatorial optimization problems. The ACO algorithm is depends on the behavior of real ants for determining the shortest path from the nest to the food source and back to the nest through putting a chemical substance termed pheromone. Foragers can follow the trail to food determined by other ants trail through sensing pheromone. Therefore, a shortest route is identified for food source. By using this ACO algorithmic process, GDACO technique selects the test cases from the test case set in order to generate a test suites for testing the software programs based on user test requirements. The following diagram shows the Test Suite Generation process using Ant Colony Optimization.

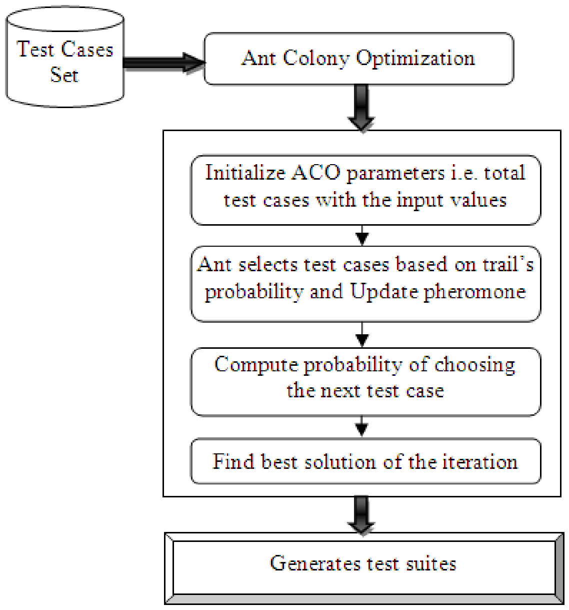
Test Cases Set
Ant Colony Optimization
Initialize ACO parameters i.e. total test cases with the input values
Ant selects test cases based on trail's probability and Update pheromone
Compute probability of choosing the next test case
Find best solution of the iteration
Generates test suites

Figure 2. **Test Suite Generation Process Using Ant Colony Optimization**

As shown in Figure 2, Ant Colony Optimization algorithm initially takes test cases set as input and then initializes the ACO parameters. In ACO algorithm, Ant selects the test cases for generating test suites based on trail's probability and subsequently update pheromone trails. This process is repeated until the maximum iteration is found. Finally, ACO algorithm finds best solutions of iterations in order to form the test suites.

For generating the test suites, ACO algorithm initially constructs a directed graph G= (V, E) in which V represents the vertices (i.e. test cases) and E denotes the edges between the two vertices. Each edge $e \in E$ is allocated with a weight which signifies the amount of pheromone that an ant may deposit on track with the primary weights assigned to 1 as shown in Figure 2. The graph is traversed through the ants based on a probabilistic approach where each crossing results in generation of test suites for evaluating the software quality. A graph structure of ACO algorithm for Test Suite Generation is shown in below Figure 3.

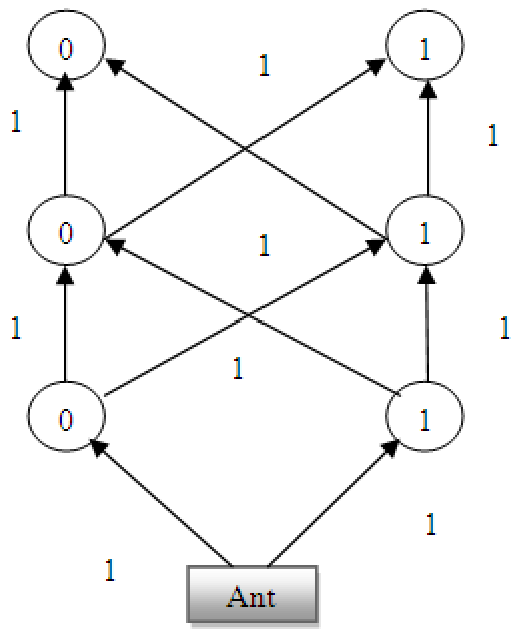

Figure 3. **Graph Structure of ACO Algorithm for Generating Test Suites**

As shown in Figure 3, during each level of graph traversal, an ant has to select between two vertices that correspond to the identical initial input. But, each vertex corresponds to a dissimilar input value possibility such as 0 or 1. An ant selects one of the two vertices based on the trail's probability by using following mathematical formula,

$$\rho = \frac{\omega_1}{\omega_2 + \omega_1} \qquad (1)$$

From the equation (1), $\omega_1$ represents the weight of the candidate trail and $\omega_2$ denotes the weight of the alternate trail. Let assume $T_{ij}(t)$ is the intensity of trail on edge $(i, j)$ at time $t$. An ant picks the next initial input depends on time $t + n$. The pheromone trails are updated by using following mathematical equation,

$$T_{ij}(t+n) = \rho . T_{ij}(t) + \Delta T_{ij} \qquad (2)$$

From the equation (2), $\rho$ is a coefficient that indicates the trail's probability between $t$ and $t + n$. An ant $k$ uses the pheromone trail to compute the probability of choosing $j$ as the next vertex when at a vertex $i$ by using following mathematical equation,

$$\Delta T_{ij} = \begin{cases} \dfrac{(T_{ij}^{\alpha} \mu_{ij}^{\beta})}{\sum_{j \in N_i} (T_{ij}^{\alpha} \mu_{ij}^{\beta})} \; for\ j \in N_i \\ 0, for\ j\ which\ not \in N_i \end{cases} \qquad (3)$$

From the equation (3), the probability of choosing the next vertex (i.e. test case) is determined. The above process is repeated until the maximum iteration is found. The algorithmic process of ACO Algorithm for generating test suite is shown in below,

**// ACO Based Test Suite Generation Algorithm**

```
Input: Test cases set: {t_1,t_2,t_3,…t_n}
Output: Test Suites : T={T_1,T_2,T_3,…T_n}
Step 1: Begin
Step 2: Initialize ACO parameters like total test cases with
        the input values
Step 3: While the termination condition is met, do
Step 4: Ant selects test cases based on trail's probability
        using (1)
Step 5: Update the pheromone trails using (2)
Step 6: Compute the probability of choosing the next test
        case using (3)
Step 8: End while
Step 9: Return the best solution found
Step 10: End
```

**Algorithm 1 ACO Based Test Suite Generation**

As shown in algorithm 1, Ant initially chooses the test cases for generating test suites based on trail's probability and subsequently update pheromone trails. During each level of the graph traversal, the ants find out the probability of choosing a vertex through creating a random number x. If x is less than ρ then the ant select the vertex to the current trail (i.e. choose test case for generating test suites) and the vertex value is retained. Otherwise, the adjacent vertex is select. This process is repeated until all the ant traverses all graph levels using the same procedure. The vertices selected for each level of the graph traversal is collected together in order to generate a test a suites.

### *B. Test Suite Reduction Using Greedy Discretization Algorithm*

The GDACO technique used Greedy Discretization algorithm for determining the optimal solution to the test suite reduction problem. The Greedy Discretization algorithm repeatedly removes the test case which unsatisfied user test requirements from the test suite set T until all the requirements are covered. The following diagram shows the process of Greedy Discretization algorithm for obtaining minimal cardinality subset test suites.

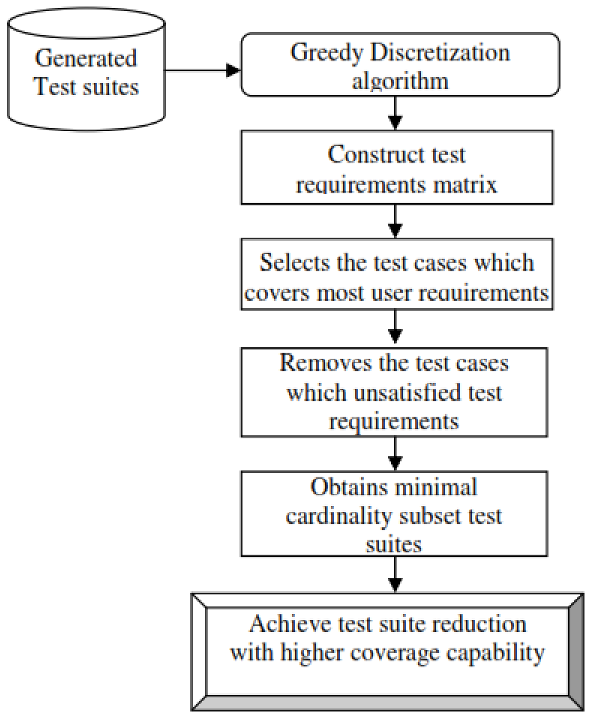
Generated Test suites
Greedy Discretization algorithm
Construct test requirements matrix
Selects the test cases which covers most user requirements
Removes the test cases which unsatisfied test requirements
Obtains minimal cardinality subset test suites
Achieve test suite reduction with higher coverage capability

Figure 4. **Process of Greedy Discretization Algorithm for Test Suites Reduction**

As shown in Figure 4, Greedy Discretization algorithm initially takes generated test suites (i.e. the output of ACO algorithm) as input. Then, Greedy Discretization algorithm creates the test requirement table. The test requirement table includes the information's about test cases and their test requirements to be satisfied. After that, Greedy Discretization algorithm selects the test cases which cover most user requirements and subsequently removes the test cases which unsatisfied test requirements and redundant test suites. Therefore, GDACO technique obtains minimal cardinality subset test suites with higher coverage capability for testing software. This in turn helps for reducing the testing cost of software program.

Greedy Discretization Algorithm initially constructs a test requirement matrix termed as TR table from the requirement and test case of software. TR is a two dimensional 0-1 value table with size of $m * n$. The test suite $T = \{t_1, t_2, t_3, \dots t_n\}$ is represented in row and the requirement $R = \{r_1, r_2, r_3, \dots r_n\}$ is represented in the column. That is each row of the table represent requirements fulfill by a particular test case. TR table value is value is determined by following mathematical formula,

$$\mathrm{TR}(i,j) = \begin{cases} 1 \text{ if } t_i \text{ statisfy } r_j \\ 0 \text{ if } t_i \text{ cannot statisfy } r_j \end{cases} \quad (4)$$

From the equation (4), TR table value is measured. The following table 1 shows the example of TR table value with a test suite of five test case and their test requirements to be satisfied.

Table I. TR Table

| **Test Case** | **Requirements to be Satisfied** | | | | |
|---|---|---|---|---|---|
| | $r_1$ | $r_2$ | $r_3$ | $r_4$ | $r_5$ |
| $t_1$ | 1 | 1 | 1 | 0 | 0 |
| $t_2$ | 0 | 1 | 1 | 1 | 0 |
| $t_3$ | 1 | 0 | 0 | 0 | 1 |
| $t_4$ | 0 | 0 | 1 | 0 | 1 |
| $t_5$ | 1 | 0 | 1 | 0 | 1 |

The TR table with $m$ rows and $n$ columns, it is essential for choosing the subset of rows to cover all of the columns in the matrix with minimal execution time. The GDACO technique used greedy discretization algorithm for removing the redundant test cases in different test suites. Let us consider 5 test suites with 9 test cases as shown in Figure 5.

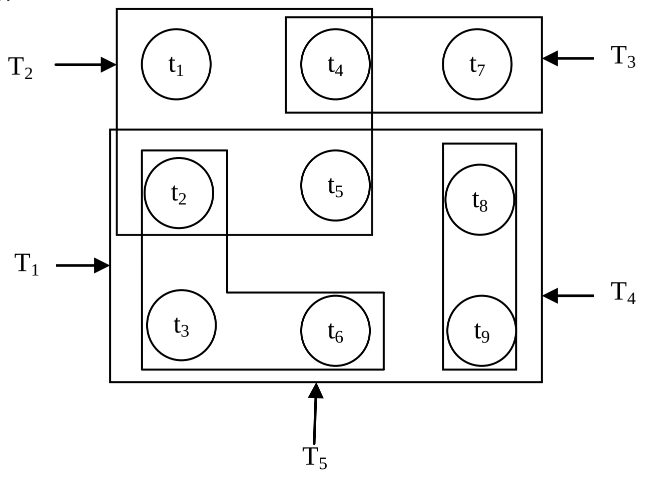

Figure 5. **Example of Greedy Discretization Algorithm Process for Test Suite Reduction**

As shown in Figure 5, the greedy discretization algorithm iteratively chooses test cases which covers maximum test requirements until all the requirements are fulfilled. Consequently, greedy discretization algorithm removes the test cases which are redundant and unsatisfied test requirements. From the Figure 5, greedy discretization algorithm picks the test suites T1, T2 and T3 as a minimal cardinality subset test suites which covers the all test requirements of software program. As a result, GDACO technique achieves higher coverage rate. The algorithmic process of greedy discretization algorithm for test suite minimization is shown in below,

```
// Greedy Discretization Based Test Suite Minimization Algorithm

Input: Test Suites : T={T_1,T_2,T_3,…T_n}
Output: Minimal Cardinality Subset of Test Suites
Step 1: Begin
Step 2: Constructs test requirement table using (4)
Step 3: For each Test Suite
Step 4: Choose the test cases which covers maximum test
        requirements until all the requirements are fulfilled
Step 5: Eliminates the test cases which is redundant and
        unsatisfied test requirements
Step 6: End for
Step 7: Return minimal cardinality subset of test suites
Step 6: End
```

**Algorithm 2 Greedy Discretization Based Test Suite Minimization**

By using the above algorithmic process, GDACO technique acquires minimal cardinality subset of test suites which covers maximum test requirements. This helps for achieving higher coverage rate of faults in software program.

## 3. EXPERIMENTAL SETTINGS

In order to evaluate the efficiency of proposed, Greedy Discrete Ant Colony Optimization (GDACO) technique is implemented in Java Language by using schoolmate data set. The GDACO technique employed schoolmate data set for discovering faults in software programs in order to increase software quality. This schoolmate data set consists of many PHP program. The performance of GDACO technique is measured in terms of coverage rate, average rate of test suite reduction and execution time.

## 4. RESULT AND DISCUSSIONS

In this section, the result analysis of GDACO technique is estimated. The effectiveness GDACO technique is compared against with two methods namely Memetic Algorithm [1] and Test case minimization approach [2] respectively. The efficiency of GDACO technique is evaluated along with the following metrics with the help of tables and graphs.

### A. *Measurement of Average Rate of Test Suite Reduction*

The average rate of test suite reduction measures the ratio of number of test suites reduced using GDACO technique to the total number of test suites taken as input. The average rate of test suite reduction is measured in terms of percentage (%) and mathematically formulated as,

$$average\ rate\ of\ test\ suite\ reduction\ = \frac{number\ of\ test\ suites\ is\ obtained\ after\ test\ suite\ reduction}{total\ number\ of\ test\ suites} * 100 \quad (5)$$

From the equation (5), average rate of test suite reduction is measured. While the average rate of test suite reduction is higher, the method is said to be more efficient.

Table II. Tabulation for Average Rate of Test Suite Reduction

| Number of Test Suites | Average Rate of Test Suite Reduction (%) | | |
|---|---|---|---|
| | Memetic Algorithm | Test Case Minimization Approach | GDACO Technique |
| 10 | 63.16 | 71.15 | 80.56 |
| 20 | 65.85 | 74.64 | 81.25 |
| 30 | 66.19 | 75.92 | 82.98 |
| 40 | 69.83 | 76.65 | 84.62 |
| 50 | 72.06 | 78.63 | 85.92 |
| 60 | 73.85 | 81.06 | 88.67 |
| 70 | 75.19 | 84.25 | 89.90 |
| 80 | 78.61 | 86.86 | 91.02 |
| 90 | 79.18 | 89.82 | 93.15 |
| 100 | 81.33 | 90.92 | 94.81 |

Table 2 presents the results obtained for average rate of test suite reduction based on diverse number of test suites in range of 10-100 using three methods. From the table value, it is clear that the average rate of test suite reduction using GDACO technique is higher when compared to existing Memetic Algorithm [1] and Test case minimization approach [2].

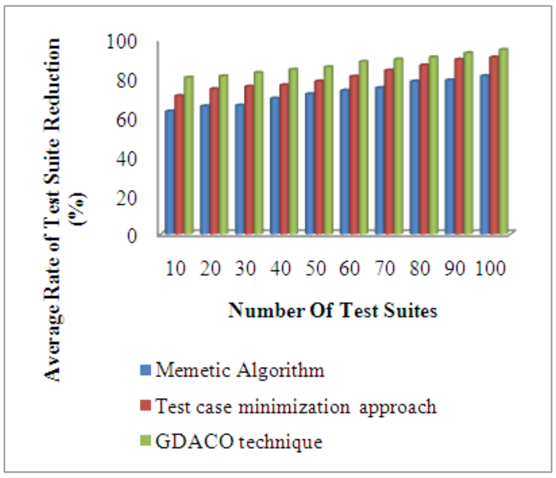

Figure 6. **Measure of Average Rate of Test Suite Reduction Vs Number of Test Suites**

Figure 6 describes the impact of average rate of test suite reduction with respect of different number of test suites. As illustrated in figure, the proposed GDACO technique is provides better average rate of test suite reduction when compared to existing Memetic Algorithm [1] and Test case minimization approach [2]. Besides, while increasing the number of test suite, the average rate of test suite reduction is also gets increased using all three methods. But comparatively, the average rate of test suite reduction using proposed GDACO technique is higher. This is because of application of Greedy Discretization Based Test Suite Minimization in GDACO technique. With aid of Greedy Discretization algorithm, proposed GDACO technique picks test cases which cover more test requirements until all the requirements are fulfilled and consequently eliminates the test cases which are redundant and unsatisfied test requirements. This in turn helps for improving the average rate of test suite reduction in an effective manner. Therefore, proposed GDACO technique increases the average rate of test suite reduction by 21% when compared to Memetic Algorithm [1] and 8% when compared to Test case minimization approach [2] respectively.

## *B. Measurement of Coverage Rate*

In GDACO technique, coverage measures the rate at which a maximum number of faults covered by a reduced test suites form the total number of test suites generated. The average coverage rate (CR) is measured in terms of percentages (%) and mathematically formulated as,

$$CR = \frac{total\ no.of\ test\ suites - reduced\ test\ suites\ which\ covers\ more\ faults}{total\ number\ of\ test\ suites} X100 \quad (6)$$

From the equation (6), coverage rate of test suites is measured. While the coverage rate is higher, the method is said to be more efficient.

Table III. Tabulation for Coverage Rate

| Number of Test Suites | Coverage Rate (%) | | |
|---|---|---|---|
| | Memetic Algorithm | Test Case Minimization Approach | GDACO Technique |
| 10 | 55.16 | 62.25 | 71.54 |
| 20 | 56.98 | 64.85 | 73.24 |
| 30 | 58.69 | 65.87 | 74.35 |
| 40 | 60.25 | 68.38 | 77.13 |
| 50 | 63.68 | 70.72 | 79.26 |
| 60 | 64.19 | 74.68 | 82.31 |
| 70 | 67.16 | 75.62 | 83.87 |
| 80 | 69.81 | 77.38 | 85.58 |
| 90 | 70.14 | 78.15 | 86.86 |
| 100 | 74.63 | 81.06 | 89.68 |

Table 3 shows the comparative result analysis of coverage rate is obtained based on different number of test cases using three methods. The GDACO considers the framework with diverse number of test suites in range of 10-100 for conducting experimental works using Java Language. Form the table value, it is illustrative that the coverage rate using GDACO technique is higher as compared to existing Memetic Algorithm [1] and Test case minimization approach [2].

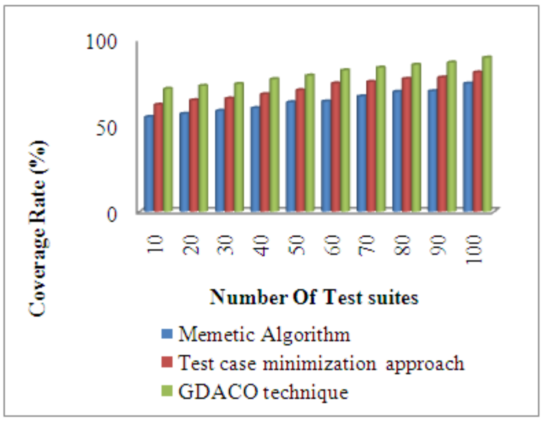

Figure 7. **Measure of Coverage Rate Vs Number of Test Suites**

Figure 7 depicts the impact of coverage rate with respect of diverse number of test suites. As exposed in figure, the proposed GDACO technique is provides better coverage rate for discovering the more faults in software program when compared to existing Memetic Algorithm [1] and Test case minimization approach [2]. In addition, while increasing the number of test suite, the coverage rate is also gets increased using all three methods. But comparatively, the coverage rate using proposed GDACO technique is higher. This is owing to application of Greedy Discretization Based Test Suite Minimization in GDACO technique. By using Greedy Discretization algorithm, proposed GDACO technique chooses test cases which cover maximum test requirements until all the requirements are satisfied. This in turn assists for improving the coverage rate of faults in an effectual manner. As a result, proposed GDACO technique increases the coverage rate by 26% when compared to Memetic Algorithm [1] and 12% when compared to Test case minimization approach [2] respectively.

### C. *Measurement of Execution Time*

In GDACO technique, the execution time measures the amount of time taken for generating the test suites. The execution time (ET) is measured in terms of milliseconds (ms) and mathematically formulated as,

$$ET = N * time(generating\ one\ test\ suites \quad (7)$$

From the equation (7), execution time of test suites generation is measured. While the execution time is lower, the method is said to be more efficient.

Table IV. Tabulation for Execution Time

| Number of Test Suites | Execution Time (ms) | | |
|---|---|---|---|
| | Memetic Algorithm | Test Case Minimization Approach | GDACO Technique |
| 10 | 24.6 | 19.2 | 11.5 |
| 20 | 30.2 | 25.8 | 18.8 |
| 30 | 35.9 | 32.4 | 23.4 |
| 40 | 41.7 | 39.5 | 28.2 |
| 50 | 48.6 | 46.8 | 32.7 |
| 60 | 55.5 | 53.1 | 38.9 |
| 70 | 63.1 | 59.4 | 42.1 |
| 80 | 72.6 | 65.2 | 46.8 |
| 90 | 80.7 | 72.9 | 50.5 |
| 100 | 86.5 | 80.5 | 56.7 |

Table 4 shows the result analysis of execution time with respect to different number of test suites in range of 10-100 using three methods. From the table value, it is expressive that the execution time of test suite generation using GDACO technique is lower when compared to existing Memetic Algorithm [1] and Test case minimization approach [2].

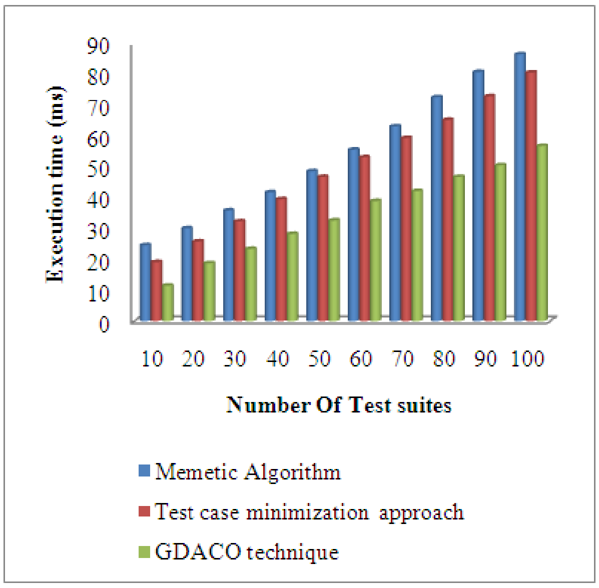

Figure 8. **Measure of Execution time Vs Number of Test Suites**

Figure 8 demonstrates the impact of execution time with respect of dissimilar number of test suites using three methods. As shown in figure, the proposed GDACO technique is provides better execution time for generating test suites when compared to existing execution time. As well, while increasing the number of test suite, the execution time is also gets increased using all three methods. But comparatively, the execution time using proposed GDACO technique is lower. This is due to application of ACO based Test Suite Generation in GDACO technique in which ant chooses the test cases for generating test suites based on trail's probability. The vertices selected during each level of the graph traversal are collected together order to generate a test a suites with lower time. This in turn supports for reducing the execution time in a significant manner. Thus, proposed GDACO technique reduce the execution time of test suite generation by 36% when compared to Memetic Algorithm [1] and 30% when compared to Test case minimization approach [2] respectively.

## 5. RELATED WORKS

Multiple coverage criteria was applied in [12] for efficient test Suite minimization and improving the capability of fault detection. A novel method was designed in [13] that remove the test case redundancy with aid of fuzzy clustering technique and provides good results for conditions/path coverage. But, time complexity taken for removing the redundancy was higher.

A Hierarchical Clustering Approach was presented in [14] for test suite minimization in which a branch coverage criterion is selected as the code coverage criteria in order to reduce the test suite. However, a reduced test suite does not cover more faults. A genetic algorithm was used in [15] to decrease the test case in regression testing. This genetic algorithm reduces test cost of regression testing and enhances the efficiency of the software with the optimized test suite.

A data mining-based algorithm was presented in [16] in which concept of maximal frequent item set mining is used for test suite reduction. A novel technique was designed in [17] to lessen the size of test suite by using improved precision slices. But, average rate of test suite reduction was lower. An effective strategy was intended in [18] to generate a balanced test suite for spectrum-based fault localization. But, coverage rate was lower.

A model-based approach was designed in [19] to lessen the amount of fault detection rate in the test suite generation. Test case classification was performed in [20] using tuned fuzzy logic for test suite reduction. However, it takes more execution time for reducing test suites.

## 6. CONCLUSION

An efficient Greedy Discrete Ant Colony Optimization (GDACO) technique is developed with the objective of improving the coverage capability of test suite generation. The GDACO technique initially generates the test suites by using ACO algorithm. The ACO algorithm chooses test cases from test cases set depends on trail's probability and consequently update pheromone trails in order to generate test suites. Afterwards, GDACO technique performs test suite optimization with assists of Greedy Discretization algorithm. The Greedy Discretization algorithm selects the test cases which cover test requirements and subsequently eliminates redundant test cases in test suites. As a result, GDACO technique finally gets minimal cardinality subset of test suites with higher coverage rate for identifying the faults in software programs. This in turn assists for reducing the testing cost of software program. The efficiency of GDACO technique is test with the metrics such as coverage rate, average rate of test suite reduction and execution time. With the experiments conducted for GDACO technique, it is observed that the coverage rate provided more accurate results for improving software quality when compared to state-of-the-art works. The experimental results show that GDACO technique is provides better performance with an improvement of average rate of test suite reduction with higher coverage rate when compared to the state-of-the-art works

## 7. REFERENCES

[1] Gordon Frasera, Andrea Arcurib, “A Memetic Algorithm for whole test suite generation”, The Journal of Systems and Software, Elsevier, Volume 103, Pages 311–327, May 2014

[2] Bestoun S. Ahmed, “Test case minimization approach using fault detection and combinatorial optimization techniques for configuration-aware structural testing”, Engineering Science and Technology, an International Journal, Elsevier, Volume 19, Pages 737–753, 2016

[3] Gregory Gay, Matt Staats, Michael Whalen, Mats P. E. Heimdahl, “The Risks of Coverage-Directed Test Case Generation”, IEEE Transactions on Software Engineering, Volume 41, Issue 8, Pages 803-819, 2015

[4] Bestoun S. Ahmed, Taib Sh. Abdulsamad, Moayad Y. Potrus, “Achievement of Minimized Combinatorial Test Suite for Configuration-Aware Software Functional Testing Using the Cuckoo Search Algorithm”, Information and Software Technology, Elsevier, Volume 66, Pages 13-29, October 2015

[5] Gordon Fraser, and Andrea Arcuri, “Whole Test Suite Generation”, IEEE Transactions on Software Engineering, Volume 39, Issue 2, Pages: 276-291, February 2013

[6] Shunkun Yang, Tianlong Man, Jiaqi Xu, Fuping Zeng, Ke Li, “RGA: A lightweight and effective regeneration genetic algorithm for coverage-oriented software test data generation”, Information and Software Technology, Elsevier, Volume 76, Pages 19–30 2016

[7] Kamal Z. Zamlia, Basem Y. Alkazemib, Graham Kendall, “A Tabu Search hyper-heuristic strategy for t-way test suite generation”, applied Soft Computing, Elsevier, Volume 44, Pages 57–74, 2016

[8] Johannes Bürdek, Malte Lochau, Stefan Bauregger, Andreas Holzer, Alexander von Rhein, Sven Apel, and Dirk Beyer,” Facilitating Reuse in Multi-goal Test-Suite Generation for Software Product Lines”, Springer, Pages 84–99, Apr 2015

[9] Rong-Zhi Qi, Zhi-Jian Wang and Shui-Yan Li, “A Parallel Genetic Algorithm Based on Spark for Pairwise Test Suite Generation”, Journal of Computer Science and Technology, Volume 31, Issue 2, Pages 417–427, 2015

[10] Tamal Sen and Rajib Mall, “State-Model-Based Regression Test Reduction for Component-Based Software”, International Scholarly Research Network, ISRN Software Engineering, Volume 2012, Article ID 561502, Pages 1-9, 2012

[11] Shilpi Singh, Raj Shree, “An Analysis of Test Suite Minimization Techniques”, international Journal of Engineering Sciences and Research Technology, Volume 5, Pages 252-260, 2016

[12] P. Velmurugan, Rajendra Prasad Mahapatra, “Effective Test Case Minimization and Fault Detection Capability Using Multiple Coverage Technique”, International Journal of Applied Engineering Research, Volume 11, Issue 8, Pages 5389-5394, 2016

[13] Gaurav Kumar, Pradeep Kumar Bhatia, “Software testing optimization through test suite reduction using fuzzy clustering”, CSI Transactions on ICT, Springer, Volume 1, Issue 3, Pages 253–260, September 2013

[14] Fayaz Ahmad Khan, Anil Kumar Gupta, Dibya Jyoti Bora, “An Efficient Technique to Test Suite Minimization using Hierarchical Clustering Approach”, International Journal of Emerging Science and Engineering, Volume 3 Issue 11, Pages 1-10, September 2015

[15] Sudhir Kumar Mohapatra, Srinivas Prasad, “Finding Representative Test Case for Test Case Reduction in Regression Testing”, International journal of Intelligent Systems and Applications, Volume 11, Pages 60-65, 2015

[16] Preethi Harris, Nedunchezhian Raju, “Towards test suite reduction using maximal frequent data mining concept”, International Journal of Computer Applications in Technology, Volume 52, Issue 1, Pages 48-58, 2015

[17] Chhabi Rani Panigrahi, Rajib Mall, “Regression test size reduction using improved precision slices”, Innovations in Systems and Software Engineering, Springer, Volume 12, Issue 2, Pages 153–159, June 2016

[18] Ning Li, RuiWang, Yu-li Tian, and Wei Zheng, “An Effective Strategy to Build Up a Balanced Test Suite for Spectrum-Based Fault Localization”, Hindawi Publishing Corporation, Mathematical Problems in Engineering, Volume 2016, Article ID 5813490, Pages 1-13, 2016

[19] Prabhu Jayagopal and Malmurugan Nagarajan, “A Novel Prioritization Algorithm Model Based Test-Suite Generation Using Regression Testing”, Journal of Computer Science, Volume 10, Issue 2, Pages 190-197, 2014

[20] R.Kamalraj and R. Rajivkannan, “Asian Journal of Information Technology”, Volume 15, Issue 9, Pages 1437-1442, 2016