RegressAid – A CASE Tool for Minimization of Test Suite for Regression Testing

A.Charan Kumari
Department of Physics & Computer Science
Dayalbagh Educational Institute
Dayalbagh, Agra, India

K.Srinivas
Department of Electrical Engineering
Dayalbagh Educational Institute
Dayalbagh, Agra, India

M.P.Gupta
Department of Management Studies
Indian Institute of Technology
Delhi, India

ABSTRACT
Software evolution is a natural phenomenon. As the software undergoes changes, it needs to be tested for the changes made along with the unchanged parts for consistency. This activity gradually increases the size of the test suite and becomes a challenging task for a software engineer to perform regression testing in a constrained environment of limited time. The activities of test case selection, test case prioritization or test suite minimization assists software engineers in regression testing by reducing the number of test cases. This paper presents a regression testing tool called 'RegressAid' to support software engineers in regression testing by minimizing the test suite while ensuring maximum code coverage and minimum execution time. This paper describes the tool along with its features. The efficacy of the tool is also demonstrated on two real world applications.

General Terms
Software Engineering, Testing.

Keywords
Multi-objective optimization, Regression testing, Test suite minimization

1. INTRODUCTION
Enhancements to the software are common during evolution. Every enhancement adds new test cases to the test suite. Regression testing ensures that the software is thoroughly tested with the complete test suite in order to ensure that the changes made to the software are consistent with the objectives of the software. Thus regression testing becomes a difficult task for the software engineer when size of the test suite increases, as it involves more time and effort. Methodologies like test case selection, test case prioritization and test suite minimization [1] assists software engineers in regression testing by reducing the number of test cases. Test case selection approaches select a subset of test cases for testing the changed portions of the software [2], while test case prioritization [3] orders the test cases as per some predefined performance goals. Test suite minimization reduces the size of the test suite by identifying the redundant test cases.

The NP-hard nature of the test suite minimization approach attracted many researchers to experiment with different metaheuristic search techniques. Tallam and Gupta [4] proposed a new greedy heuristic approach for selecting the minimal subset of test suite that covers all the requirements covered by the original test suite. In their experiments they found that their approach produced same size or smaller size test suites than prior heuristic approaches and had comparable time performance. In 2007, Zhong *et al.* [5] conducted an experimental study on four test suite reduction techniques – Harrold *et al*. heuristic, Chen and Lau's GRE heuristic, Mansour and EI-Fakin's genetic algorithm based approach and Balck *et al.* ILP-based approach . They also provided an insight for choosing an appropriate test suite reduction technique.

Smith and Kapfhammer [6] try to reduce and prioritize the test cases based on cost and ratio of code coverage to cost using greedy approaches. They experimented on eight real world applications and found that greedy approaches aid in identifying smaller and faster test suites. Recently, Shin Yoo and Mark Harman [7] introduced a multi-objective test suite minimization problem and instantiated this with two versions - two-objective formulation that caters for code coverage and execution cost and a three-objective formulation that caters for code coverage, execution cost and fault-history. They experimented on five non-trivial real-world programs using two algorithms: a re-formulation of the single-objective greedy algorithm and a hybrid variant of NSGA-II. Their empirical study investigated the relative effectiveness of two algorithms for Pareto efficient multi-objective test suite minimization and found that the multi-objective approach can lead to more efficient testing decisions.

By considering the size and complexities involved in the software and the radical increase in the size of the test suites, this paper presents a tool to assist software engineers in regression testing by minimizing the size of the test suite, while ensuring maximum code coverage and minimum execution time. The tool uses a Multi-objective Quantum-inspired Hybrid Differential Evolution (MQHDE) [8, 9] for optimizing the test suite. The tool also provides an option to minimize the test suite using the state-of-the-art multi-objective evolutionary optimization algorithm NSGA-II [10].

The rest of the paper is organized as follows. Section 2 describes the multi-objective test suite minimization problem and brief descriptions of the algorithms are provided in section 3. Section 4 explains the tool along with the results obtained by the tool on a real-world application data. Concluding remarks are given in Section 5.

2. MULTI-OBJECTIVE TEST SUITE MINIMIZATION PROBLEM
This section describes the multi-objective test suite minimization problem as formulated by Shin Yoo and Mark Harman [7]. The multi-objective test suite minimization

problem is to select a subset of test suite, based on multiple test criteria. That is, given a test suite S, a vector of M objective functions, the problem is to find a subset S′ of S such that S′ is a Pareto optimal set with respect to M. The objective functions are the mathematical elucidations of the test criteria.

The developed tool is based on the two-objective formulation of test suite minimization problem with statement code coverage as a measure of test adequacy and execution time as a measure of cost. Thus, code coverage is taken as one of the objective functions which is to be maximized for a given cost and time is the second objective that is to be minimized for a given code coverage. Therefore, the problem can be stated as to find a subset of the test suite S with code coverage C and execution time T such that the following two conditions are satisfied simultaneously [7].

T1: No other subset of S can achieve more coverage than C without spending more time than T.

T2: No other subset of S can finish in less time than T while achieving a coverage that is equal to or greater than C.

3. THE ALGORITHMS

This section briefly describes the two algorithms used in the tool for optimization of test suite.

3.1 Multi-objective Quantum-inspired Hybrid Differential Evolution (MQHDE)

The Multi-objective Quantum-inspired Hybrid Differential Evolution was designed by Charan Kumari *et al.* A detailed description of the algorithm can be found in [8, 9]. The algorithm integrates the features of differential evolution and genetic algorithm into the quantum paradigm for a fast and effective search. In the beginning, the quantum population is initialized in the range [-1, 1]. After observing and evaluating the quantum population, half of the quantum population is updated using the mutation operator of differential evolution and the remaining half of the quantum population is updated using the uniform crossover operator of genetic algorithm. The updated population is observed, its fitness is evaluated and both the populations are combined and sorted based on fast non dominated sorting [10]. The quantum population for the next iteration is obtained by picking up the quantum individuals from good fronts, while giving importance to the crowding distance measure to ensure good diversity in the quantum population. The pseudo code of MQHDE is given in Algorithm 1.

Algorithm 1: Multi-objective Quantum-inspired Hybrid Differential Evolution (MQHDE)

1: t = 0
2: Initialize Q(t) a population of 'N' qubit individuals with 'm' qubits in each.
3: Obtain P(t) by observing the states of Q(t).
4: Evaluate fitness of P(t).
5: Perform fast non-dominated sort on P(t)
6: **while** not termination condition **do**
7: t = t +1
8: Obtain half of the offspring population Q(t) using the quantum mutation operator applied on parent population Q(t-1) and elites of Q(t-1) as shown below:
$q_i(t) = q_{elite}(t-1) + F * (q_{r1}(t-1) – q_{r2}(t-1))$,
where $r_1 \neq r_2 \neq i$ and $F \in [0,2]$.
9: Obtain the remaining offspring population Q(t) using Quantum uniform crossover.
10: Obtain P(t) by observing the states of Q(t).
11: Evaluate the fitness of P(t).
12: Perform fast non-dominated sort on P(t-1) U P(t)
13: Form Q(t) by accommodating distinct Quantum individuals pertaining to the different Pareto-fronts starting from the best front by taking crowding distance into consideration.
14: **end while**

3.2 Non-Dominated Sorting Genetic Algorithm-II (NSGA-II)

The Non-dominated Sorting Genetic Algorithm was proposed by Deb *et al.* [10]. NSGA-II incorporates an explicit diversity and elite preserving mechanism to retain the best solutions found in all the iterations. After the population is initialized, it is sorted based on non-domination into different fronts based on the goodness of the solutions. In addition to the fronts found, a parameter called crowding distance (a measure of how close an individual is to its neighbors) is calculated for each individual. Large average crowding distance will result in better diversity in the population. Parents are selected from the population by using binary tournament selection based on the front and crowding distance. The selected population generates offspring using crossover and mutation operators. The parent and offspring populations are combined and sorted again based on non-domination. The individuals equal to the size of the population is selected based on the front and crowding distance. The main procedure [11] is outlined in Algorithm 2.

Algorithm 2: Non-dominated Sorting Genetic Algorithm-II (NSGA-II) [11]

1: Combine parent and offspring populations and create $R_t = P_t \cup Q_t$. Perform a non-dominated sorting on R_t and identify different fronts F_i , i = 1, 2, ……. , etc.
2: Set new population $p_{t+1} = \phi$. Set a counter i = 1.
Until $|P_{t+1}| + |F_i| < N$, perform $P_{t+1} = P_{t+1} \cup F_i$ and i = i + 1
3: Perform the crowding sort procedure and include the most widely spread $(N-|P_{t+1}|)$ solutions by using the crowding distance values in the sorted F_i to P_{t+1}.
4: Create offspring population Q_{t+1} from P_{t+1} by using the crowded tournament selection, crossover and mutation operators.

4. THE TOOL – REGRESSAID

This section gives a detailed description of the tool along with its features.

4.1 The user interface

Figure 1 depicts a snapshot of the user interface of RegressAid. This window collects information necessary to perform test suite minimization. It is divided into three sections. The first section accepts number of test cases in the test suite along with the lines of code of the software. The second section collects the text file names of execution cost file, containing the data regarding the cost of execution of

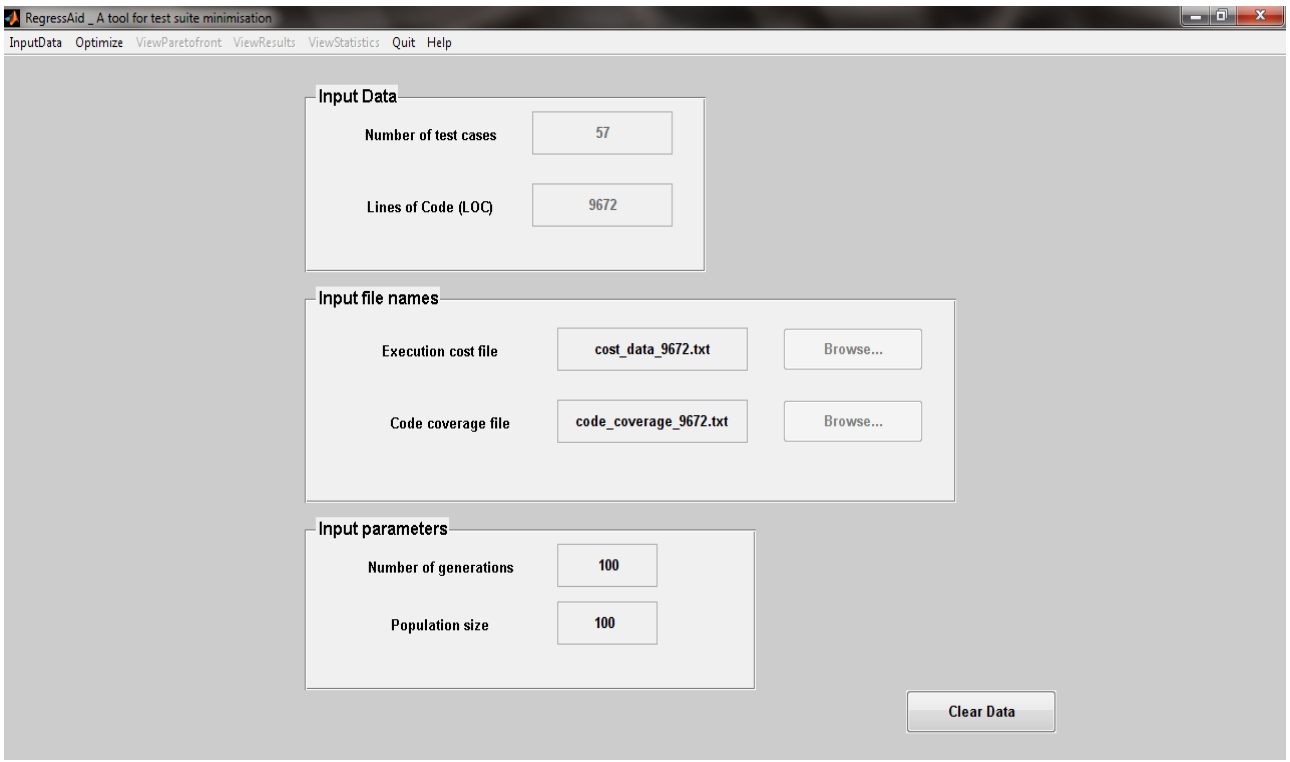

Figure 1 The user Interface of RegressAid tool

each test case and code coverage file, containing the data pertaining to statement coverage by each test case. The general parameters related to the optimization algorithm of population size and number of generations is gathered in the third section. The tab at the bottom of the window is used to clear the data entered.

4.2 Optimization

The optimize option opens a window with an option to select an algorithm for optimization between MQHDE and NSGA-II. Figure 2 depicts the features of the tool during the optimization process. The tool also provides a what-if analysis of the two objectives of code coverage and execution cost using a slider. The position of the pointer on the slider is described by the execution cost, code coverage and the selected test cases. An instance of such an analysis is shown in Figure 3.

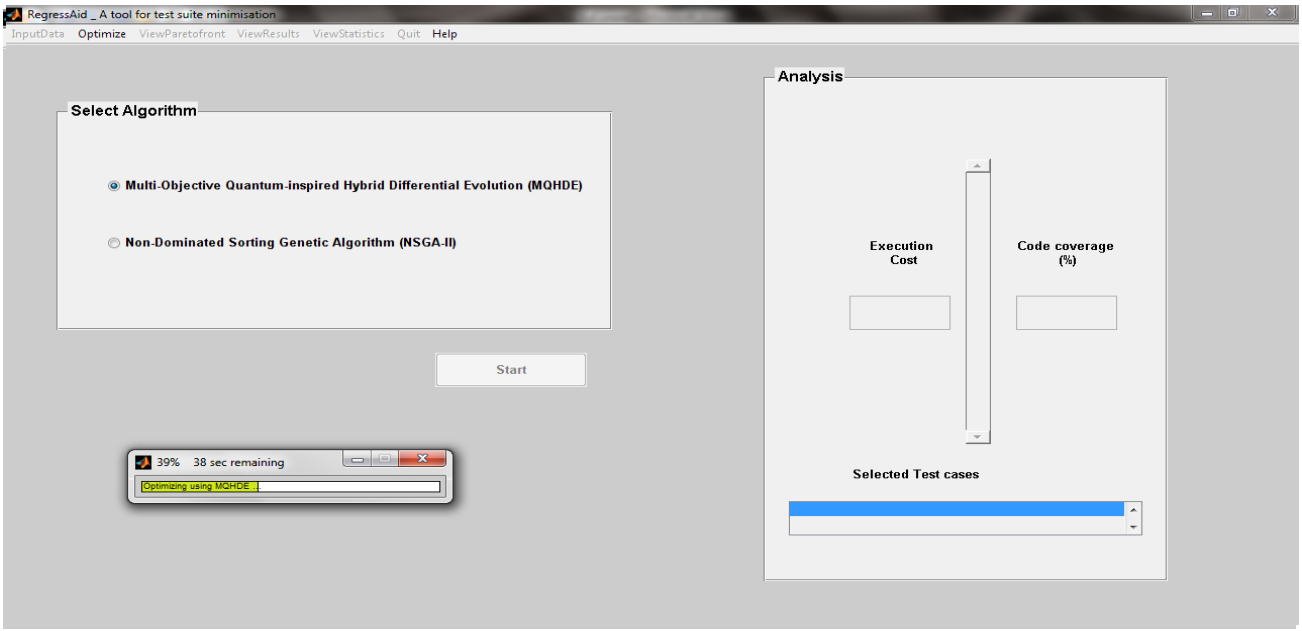

Figure 2 Optimization window (during optimization)

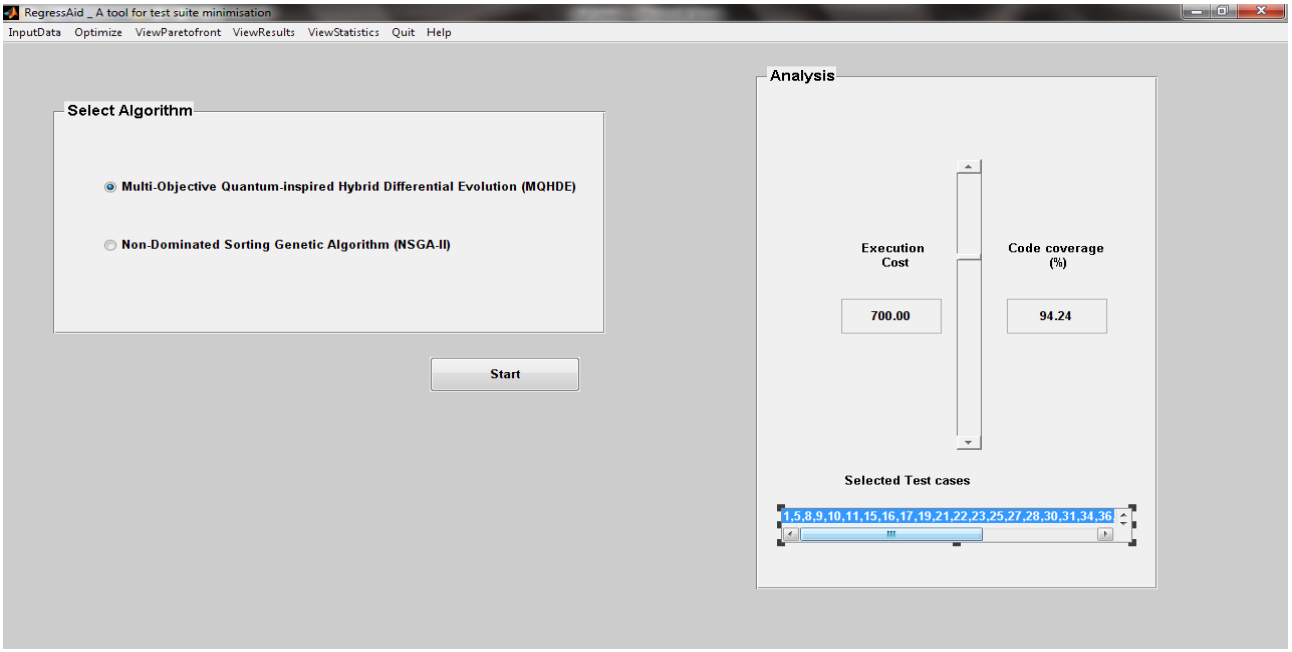

Figure 3 Optimization window (after optimization)

4.3 Results

The results after optimization process can be viewed in three different ways.

4.3.1 Pareto fronts

Pareto front is obtained by plotting all the Pareto efficient solutions in the objective space. As the test suite minimization is a multi-objective optimization problem, the Pareto front contains a number of non-dominated solutions rather than a single solution. The efficiency of the algorithm is determined by the distribution of these solutions on the Pareto front along with their closeness to the actual or true solutions. The solutions obtained are plotted against code coverage and execution cost.

4.3.2 Detailed view

Each solution obtained by the selected algorithm is presented in a detailed fashion. For each solution, the execution cost, amount of code coverage (in percentage) along with the selected test cases for that particular solution are listed.

4.3.3 Statistics

The overall statistics of the result obtained by the selected optimization algorithm are provided. These statistics mainly include the boundaries of the solutions achieved by the algorithm (minimum and maximum values obtained for code coverage and execution cost), size of the obtained Pareto front, execution time of the algorithm, along with the value of hypervolume metric, which measures convergence and diversity of the obtained solutions.

All these three forms of results are provided with save and print options.

4.3.4 Performance of RegressAid

The efficacy of the tool is tested using two real-world application data sets from the software industry. The first application consists of 28 test cases with 2487 lines of code and the second application is having 57 test cases with 9672 lines of code. The Pareto fronts obtained by MQHDE and NSGA-II for the first application are presented in Figure 4 and Figure 5 and for the second application in Figure 6 and Figure 7 respectively.

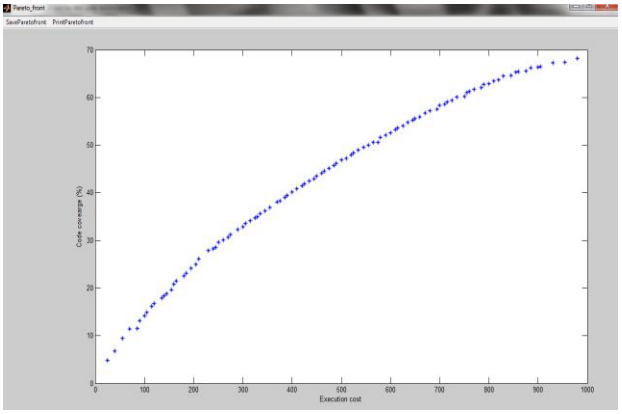

Figure 4 Pareto front obtained by MQHDE for the first application

The visual analysis of the Pareto fronts obtained in the two applications reveals that the size of the Pareto frontier obtained by MQHDE is greater than that of NSGA-II. The boundaries of the solutions and also the spread of the solutions obtained by MQHDE reveal the exploration and exploitation capabilities of MQHDE. The quality of the solutions obtained by MQHDE is also found to be better.

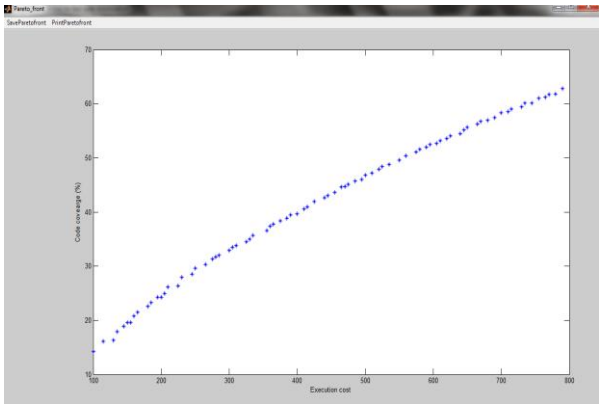

Figure 5 Pareto front obtained by NSGA-II for the first application

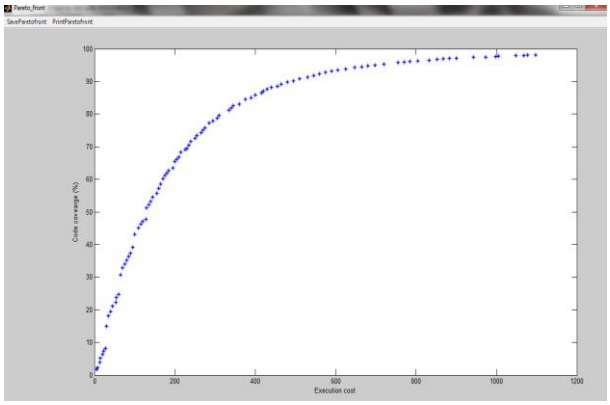

Figure 6 Pareto front obtained by MQHDE for the second application

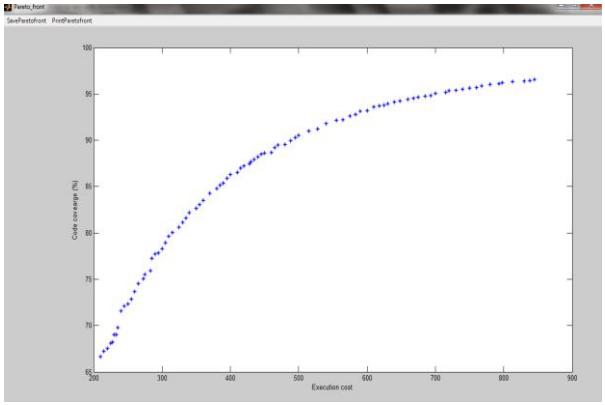

Figure 7 Pareto front obtained by NSGA-II for the second application

5. CONCLUSION

This paper presented the frame work of RegressAid, a tool for test suite minimization to assist the software engineers in regression testing. The tool has a user friendly interface. It provides an option between two algorithms (MQHDE and NSGA-II) for optimization. The tool also provides a *what-if* analysis on the objective function values of code coverage and execution cost. The RegressAid provides the results in the form of Pareto fronts, a detailed report of each obtained solution and also brief statistics indicating the quality of the solutions obtained. A help facility is also included to assist the user in the usage of the tool. The results obtained by the tool clearly indicate the efficacy of MQHDE over NSGA-II. Rich features, intuitive user interface and efficient algorithms makes RegressAid a useful tool for minimizing test suite for regression testing.

6. ACKNOWLEDGMENTS

The authors are extremely grateful to Revered Prof. P. S. Satsangi, Chairman, Advisory Committee on Education, Dayalbagh, for continued guidance and support.

7. REFERENCES

[1] Shin Yoo and Mark Harman. 2012. Regression testing minimization, selection and prioritization: a survey. Software: Testing, Verification and Reliability, vol. 22, no. 2, pp. 67–120.

[2] G. Rothermel and M. J. Harrold. 1997. A safe, efficient regression test selection technique. ACM Transactions on Software Engineering and Methodology, vol. 6, no. 2, pp. 173-210.

[3] Zheng Li, Mark Harman, and Robert M. Hierons. 2007. Search Algorithms for Regression Test Case Prioritization. IEEE Transactions On Software Engineering, vol. 33, no. 4, pp. 225-237.

[4] S. Tallam and N. Gupta. 2006. A Concept analysis inspired greedy algorithm for test suite minimization. SIGSOFT Software Engineering Notes, vol. 31, no. 1, pp. 35-42.

[5] Hao Zhong, Lu Zhang , Hong Mei. 2008. An experimental study of four typical test suite reduction techniques. Information and Software Technology, vol. 50, pp. 534-546.

[6] Adam M. Smith and Gregory M. Kapfhammer. 2009. An Empirical Study of Incorporating Cost into Test Suite Reduction and Prioritization. Symposium on Applied Computing. pp. 461-467.

[7] Shin Yoo and Mark Harman. 2010. Using hybrid algorithm for Pareto efficient multi-objective test suite minimisation. The Journal of Systems and Software, vol. 83, pp. 689–701.

[8] A. Charan Kumari, K. Srinivas and M. P. Gupta. 2013. Software Requirements Optimization Using Multi-Objective Quantum-Inspired Hybrid Differential Evolution. EVOLVE – A Bridge between Probability, Set Oriented Numerics, and Evolutionary Computation II Advances in Intelligent Syatems and Computing, vol. 175, pp. 107-120.

[9] A. Charan Kumari and K.Srinivas. 2013. Search-based Software Requirements Selection: A Case Study. International Journal of Computer Applications, Volume 64– No.21, pp. 28-34.

[10] Deb, K., 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.

[11] Deb, K., 2001, Multi-Objective Optimization using Evolutionary Algorithms. Wiley Chichester, UK.