Automated Test Case Prioritization using RGrasp

M.Nalini Sri
Assistant Professor, Department of Electronics and Computer Engineering
K. L. University, Vijayawada, India

Lakshmi.A
Student, Department of Electronics and Computer Engineering
K. L. University, Vijayawada, India

ABSTRACT
Several alterations in the software design would sometimes result in the failure of the system which has been operating effectively, meeting all the specifications at that point in time. In order to recognize the unpredictability in the performance of the system, testing is carried out. Regression testing involves validating the modified software and detects whether new faults have been introduced into the test code which has been previously tested. It is very inefficient to perform the re-execution of each test case as it is very time consuming. So, test case prioritization has been introduced. Test case prioritization involves systematizing of test cases in an order, based on some objective such as block coverage, fault detection rate, thus enhancing the performance of the regression testing. In this paper, we have proposed a new test case prioritization metaheuristic termed RGRASP, for performing automatic test case prioritization, along with various search based algorithms for regression test case prioritization .The aim of this paper is to provide an insight in performing prioritization using numerous techniques.

Keywords
Regression testing, Test case prioritization, Greedy algorithm, Additional Greedy algorithm, GRASP, RGRASP

1. INTRODUCTION
Whenever a new model has been added as a part of integration or the system is modified, the software changes. Due to this modification, new dataflow paths are established, new input/output may occur and new control logic is invoked. These changes may cause problems with the functions that have previously worked flawlessly. Moreover, the effect that these changes would bring is even unpredictable. For this reason, regression test is performed in order to uncover the errors or the regressions that have arisen due to the adjustment of the system [1].

There are two ways of conducting regression testing. Firstly, executing each and every test case, so that the entire system is tested to observe or measure its performance. But, there may not be adequate resources, required for the whole system testing and additionally, it results in the consumption of time. So, the second method has been implemented. It includes scheduling the test cases in an execution order according to some criterion, so that the most favorable tests are executed first which would result in escalating the performance of regression testing. Such a practice is known as regression test case prioritization. The main purpose of this prioritization is to increase the likelihood that if the test cases are used for regression testing in the order, they will more closely meet some objective than they would if they were executed in some other order.

In this paper, four search techniques are presented, which includes Greedy algorithm, Additional Greedy algorithm, Genetic algorithm, Simulated Annealing together with a metaheuristic technique called GRASP (Greedy Randomized Adaptive Search Procedure). In addition to these, we have projected an innovative method called RGRASP (Reactive Greedy Randomized Adaptive Search Procedure) for carrying out automatic test case prioritization.

The rest of the paper is organized as follows. Section 2 describes about the test case prioritization. Various search based algorithms are depicted in Section 3. In Section 4, the GRASP technique and RGRASP method along with its associated algorithms illustrating its working are demonstrated. Section 5 gives conclusion to the paper.

2. TEST CASE PRIORITIZATION
Test case prioritization is a technique in which each test case is assigned a priority. Priority is allocated according to some basic criterion and test cases with highest priority are scheduled first. There may be many criterions or the objectives, based on which the test cases are scheduled. Some of the measures on which test case prioritization technique focuses are the coverage measures or the so called 'coverage objectives'. As pointed out by C. L. B. Maria et al.[2], Firstly, APBC (Average Percentage of Block Coverage), which measures the rate at which a particular test suite covers the blocks of test code. Secondly, APSC (Average Percentage of Statement Coverage), which assess the rate at which a prioritized test suite would cover the statements in the code to be tested. Thirdly, APDC (Average Percentage of Decision Coverage), which evaluates the rate at which a prioritized test suite covers the decision statements in the code for which the testing is to be performed.

Besides, these measures the other objectives that the test case prioritization can address includes: the rate at which risk high faults can be detected, the rate at which the reliability of the system, under test, can be detected and improved, rate of cost per coverage of code components, rate of cost per coverage of features listed in a required specification and many more such effects.

Test case prioritization technique can be implemented both manually as well as automatically. Previously, Greedy algorithm has been used in scheduling the test cases so as to obtain an optimal ordering. But this algorithm resulted in providing the only the local optimal solution but may not provide the optimal test case ordering as stated by Rothermal [3] and Li et al.[4]. As such various other metaheuristic search techniques are involved in finding the optimal or near optimal solutions to the optimization problems.

Metaheuristic search techniques [5], are high-level frameworks that utilize the automated discovery of heuristics in order to find solutions to combinatorial problems at a reasonable computational cost. In the context of software engineering, a new research field called SBSE(Search -based Software Engineering) [2], has been emerged by the application of the metaheuristics, to well known complex software engineering problems. In this field, the software engineering problems are modeled as optimization problems, by defining an objective function or set of constraints and the solutions to such problems are found by the application of the search based techniques.

3. SEARCH – BASED PRIORITIZATION ALGORITHMS

This section signifies some of the search based test case prioritization techniques which are recurrently used in dealing with the test case prioritization problems. Let us have a brief description of working of each algorithm.

3.1 Greedy Algorithm

Greedy Algorithm is an accomplishment of the "next-best" search philosophy. It is based on the principle that the element that is, the test case with the maximum weight or the highest percentage of coverage is considered first and is added to the initially empty solution. Then, it is followed by the next test case with highest weight, and the process goes on till a complete but a suboptimal solution has been obtained.

Consider the example of statement coverage for a program containing m statements and a test suite containing n test cases. For the Greedy Algorithm, the statements covered by each test case should be counted first, which can be accomplished in O(m n) time; then, the test cases should be sorted according to the coverage. In the second step, quicksort can be used, thereby increasing the time complexity by O(n log n). Typically, m is greater than n, in which case, the cost of this prioritization is O(m n) as stated in [4].

For example, consider a problem, with four test cases. Test case A covers ten statements, the maximum that can be covered by a single test case among the four. Test case B, covers five statements. Test cases C and D cover the same number of statements but less than B, and so the Greedy Algorithm could return either A B C D or A B D C depending upon the order in which test cases are considered.

Let APBC be the coverage criterion, and let the partial solution contains three test cases that covers 1000 blocks of code. Suppose consider there are two other test cases that could be to a solution. The first solution covers 750 blocks of code, but out of these 400 have been already covered by the current solution. Then, this solution covers 75% of the blocks, but the actual added coverage of this test case is 35% of the coverage. The second test case covers 500 blocks of code, but none of these blocks were covered by the current solution .This means that this solution covers 50% of the blocks. The Greedy algorithm would select the first test case, because it has the greater percentage of block coverage overall.

3.2 Additional Greedy Algorithm

The Additional Greedy Algorithm always adds a locally optimal test case to a partial test suite. During each iteration, the algorithm adds the test case which gives the maximum coverage gain to the partial solution.

Again, consider statement coverage: The Additional Greedy Algorithm requires coverage information to be updated for each unselected test case following the choice of a test case. Given a program containing m statements and a test suite containing n test cases, selecting a test case and readjusting coverage information has cost O(m ,n) and this selection and readjustment must be performed O(n) times. Therefore, the cost of the Additional Greedy Algorithm is $O(m, n^2)$ as highlighted in [4].

Let us consider the same example from Section 3.1. Let a partial solution contain three test cases that covers 1000 blocks of code. There are remaining two test cases: the first covers 750 blocks of code, out of these 400 have been already covered by the current solution, the second covers 500 blocks of code, but none of these blocks were covered by the current solution. Then, the first solution covers 35% of the block coverage while the second test case covers 50% blocks of code. The Additional Greedy Algorithm would select the second test case, because that solution has greater percentage of weight related to the current partial solution.

3.3 Genetic Algorithm

Genetic Algorithm represents a class of adaptive search techniques which are mainly employed to solve optimization problems. It includes an initial population which is a set of randomly generated individuals. Each individual is represented by a sequence of variables/parameters (called genes), known as the chromosome. The procedure works, until a stopping criterion is met, as the new populations are generated based on the previous population. The generation of the new population is done through "genetic operators" and the choice of selecting individuals of the current solution that will generate the new population individuals. This algorithm prioritizes the test cases based on the fitness value.

In the genetic algorithm proposed by Li et al. [4], the initial population is fashioned by randomly choosing from the test case pool. The fitness function was calculated as follows:

$$\text{fitness (pos)} = 2.\frac{(pos-1)}{(n-1)} \qquad (1)$$

where *pos* is position of the test case in the current test suite and *n* is the population size.

The Crossover Algorithm (Recombination) is used to produce two offspring o_1 and o_2 from two parents p_1 and p_2, following the ordering chromosome crossover style adopted by Antoniol et al. [3]:

- A random position *k* is selected in the chromosome.
- The first k elements of p_1 become the first *k* elements of o_1.
- The last *n-k* elements of o_1 are the sequence of the *n-k* elements which remain when the *k* elements selected from p_1 are removed from p_2.
- o_2 is obtained similarly, composed of the first *n-k* elements of p_2 and the remaining elements of p_1 (when the first *n-k* elements of p_2 are removed).

The mutation is generally performed by selecting the two test cases and interchanging their positions in the test case sequence.

3.4 Simulated Annealing

Simulated annealing is a generalization of a Monte Carlo method. Its name comes from annealing in metallurgy, where a melt, initially disordered at high temperature, is slowly cooled, with the purpose of obtaining a more organized system (a local optimum solution). The system approaches a frozen ground state with $T = 0$. Each step of simulated annealing algorithm replaces the current solution by a random solution in its neighborhood, based on a probability that depends on the energies of the two solutions.

These are some of the search based algorithms which resulted in solving the problems that are associated with the test case prioritization.

4. TEST CASE PRIORITIZATION USING RGRASP

This section suggests a innovative methodology for test case prioritization using RGrasp metaheuristic. Prior to this approach, Grasp algorithm has been explained, which would be employed in the novel approach proposed later.

4.1 Grasp Algorithm

GRASP is an acronym for Greedy Randomized Adaptive Search Procedure. This procedure is also termed as "multistart algorithm", as it is frequently, carried out in order to obtain the most favorable solution [6]. GRASP is a top level general strategy, which various other heuristics in search of the feasible solutions to the test case prioritization problems in their domain. GRASP has two phases namely, construction phase, and local search phase [7].

4.1.1 Construction Phase

In this phase, the feasible solution is iteratively constructed one element at a time by application of a randomized greedy strategy.

This strategy holds a Restrict Candidate List (RCL) which maintains all the conceivable test cases or the elements that are about to be added in each iteration. These conceivable elements signify the test cases that contribute with the best coverage value. This RCL is regulated in length by the parameter α. If α = 0, then there is only one optimal solution with highest coverage in the list and it follows the perfect greedy algorithm. If α = 1, then there are possible number of test cases, which are randomly picked up in each iteration with respect to some greedy function. This function measures the benefit of selecting each element.

This heuristic is adaptive, because the benefits associated with every element are updated in each iteration of the construction phase to reflect the changes brought on by the selection of the previous element. The solution thus obtained at each iteration is then stepped into the next phase termed local search phase.

4.1.2 Local Search Phase

In case of many deterministic methods, the solutions that are generated by the construction phase are not guaranteed to be locally optimal with respect to the simple neighborhood definitions. So, it is always beneficial to apply the local search to attempt to improve each constructed solution. Using the local search procedure, the current solution is superseded with the local optimum in the neighborhood solution set. After this course of action, this local optimal solution is compared with that of the optimal solution found in the previous iterations. Based on their percentage of coverage values, these two solutions are exchanged accordingly.

Evidently, the response of this algorithm intensively, depends on the value of the parameter α. So, to facilitate the diminution of this impact, the GRASP algorithm has been modified to RGRASP metaheuristic, as emphasized in [8].

4.2 RGrasp Metaheuristic

RGrasp approach follows the Grasp algorithm, besides, updating the values of the parameter α based on the former performance. For this course of action, this approach, has initially, determined a array of values for α. Moreover, each value of α is assigned a probability, of being selected, $1/n$ where 'n' is the measure of the set of α values. For each one of the i value of α, the probabilities p_i are evaluated recursively, based on the below equation as mentioned in [6]

$$pi = \frac{qi}{\sum_{j=1}^{n} qj} \qquad (2)$$

with $q_i = S^*/A_i$, *where* S^* is the incumbent solution and A_i is the average value of all the solutions found with $\alpha=\alpha_i$. Now, when a particular value of α stimulates a better solution, then its respective probability of being selected is improved in future. Alternatively, if the solution proves to be bad, then the probability associated with that specific α value is reduced in the further iterations.

4.3 RGrasp Algorithm

The RGrasp algorithm is shown in the pseudo-code illustrated in the Figure 1 below, as pointed in [7], [8]. The first step involves assigning the probabilities of selecting each value of α. Initially, all the probabilities are assigned to 1/n where n is the quantity of set of values of α represented by αSet. Then the construction and the local search phases of the Grasp are executed iteratively, until the stopping criterion is attained.

```
(1) Initialize probabilities associated with α as 1/n
(2) For k=1 to max_iterations do
(3)   α← select_α(αSet);
(4)   solution←run_constructionphase(α);
(5)   solution←run_localsearchphase(solution);
(6)   update_solution(solution, best_solution);
(7) end;
(8) return opimum_solution;
```

Figure 1: RGrasp for Test Case Prioritization

After performing the constructive and local phases, the best solution is updated when a new solution is identified as the better solution in each iteration.

Now, let's consider how the probabilities of selecting are shifted for each value of α. It is illustrated in Figure 2 as shown below:

```
procedure select_α(αSet)

(1) α ← α with probability
      pi = qi / Σ_{j=1}^{n} qj ;
(2) return α;
```

Figure 2: Selection of α

As, described in the local search phase of GRASP Algorithm, probabilities of p_i are reevaluated at each iteration by using (1). After selecting the value for α, we need to perform the execution of the construction phase of Grasp.

It is explained in the following Figure 3, [8]. Initially an empty solution is considered. Now, the candidate set C is initialized with the test cases from various test suites with respect to the greedy function.

As, such the coverage of all the test cases in the candidate set are evaluated. Now, for each iteration, one test case which increases the coverage of the current solution is selected by the greedy evaluation function. This element is randomly selected from the Restricted Candidate List), which has the elements with best values. After the element is incorporated to the partial solution, the RCL is updated. The increment of coverage is then reevaluated.

```
(1)solution←{};
(2) initialize the candidate set C with random test cases
from the pool of test cases;
(3) calculate the coverage c´(e) for all e ε C;
(4) while C≠{} do
(5)    c^min = min{c´(e)| e ε C};
(6)    c^max = max{c´(e)| e ε C};
(7)    RCL={e ε C | c´(e) ≤ c^min + α (c^max – c^min)};
(8)    s← test case from RCL at random;
(9)    solution← solution U {s};
(10)   update C;
(11)   recalculate c´(e) for all e ε C;
(12) end;
(13) update_αSet(solution);
(14) return solution;
```

Figure 3: Algorithm for Constructive Phase of RGrasp

The *α*Set is updated after the solution is found, in order to change the selection probabilities of the *α*Set elements. This update is exemplified in Figure 4. Besides this, the pseudo-code that is associated with the local search phase of the RGRASP metaheuristic is demonstrated in Figure 5.

procedure update_ αSet (solution)

update probabilities of all the values of α in αSet using the equation

$$pi = \frac{qi}{\sum_{j=1}^{m} qj}$$

Figure 4: Procedure for Updating α

while *s* is not locally optimum do

(1) find *s´* ε neighborhood (*s*), with f(*s´*) < f(*s*);
(2) *s*← *s´;*
(3) end*;*
(4) return*(**s** as the optimal solution the test case prioritization problem);*

Figure 5: Algorithm for Local Search Phase of RGrasp

Let *s* be the test suite generated by the construction phase. Now, the first test case on the test suite is exchanged with the other test cases, one at a time, such that $n - 1$ new test suites are generated, exchanging the first test case with the *i* th one, where *i* varies from 2 to *n*, and *n* is the length of the original test suite. The original test suite is then compared with all generated test suites. If one of those test suites is having better coverage than the original one, it replaces the original solution. Thus, solving the test case prioritization problems automatically, by generating the best optimal solution on the basis of some criterion is made possible through RGRASP.

5. CONCLUSION

As the system's parameters are changed, in order to identify its performance, we generally perform regression testing. In regression testing, we need to carry out the re-execution of every test case. But re-execution of each and every test case is a time consuming process and also requires certain resources which may not be available at that particular instant of time. So, in order to increase the effectiveness of regression testing, test case prioritization is made. This test case prioritization mechanism involves prioritizing the test cases, so that the most efficient or the test cases with the highest priority are executed first in determining the performance of the system due to the new environment. This prioritization can be performed both manually and automatically. In this paper we proposed a unique approach called RGrasp Algorithm for automated test case prioritization.

6. REFERENCES

[1] Roger S. Pressman Software Engineering a practitioner's approach 6/e, 2005.

[2] C.L.B.Maia et al, "Automated test case prioritization with reactive grasp", Advances in Software Engineering, volume 2010, Hindawi Publishing Corporation, article id 428521, dol:10.1155/2010/428521.

[3] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE Transactions on Software Engineering*, vol. 27, no. 10, pp. 929–948, 2001.

[4] 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.

[5] Modern Heuristic Techniques for Combinatorial Problems, C.R. Reeves, ed. John Wiley & Sons, 1993.

[6] M. Resende and C. Ribeiro, "Greedy randomized adaptative search procedures," in *Handbook of Metaheuristics,* F. Glover

and G. Kochenberger, Eds., pp. 219–249, Kluwer Academic Publishers, Dordrecht, The Netherlands, 2001.

[7] T.A.Feo, M.G.C. Resende, "*Greedy randomized adaptive search procedures*" in Journal of Global Optimization, 6, 109-134 (1995).

[8] P.R.Srivastav, "Test case prioritization" in *Journal of Theoretical and Applied Information technology.*

[9] G. Antoniol, M. D. Penta, and M. Harman, "Search-based techniques applied to optimization of project planning for a massive maintenance project," in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM '05)*, pp. 240–252, Budapest, Hungary, September 2005.

[10] G. Rothermel, R. Untch, C. Chu, and M.J. Harrold, "*Test Case Prioritization: An Empirical Study,"* Proc. Int'l Conf. Software Maintenance, pp. 179-188, Sept. 1999.

[11] SEBASE, Software Engineering By Automated Search, September 2009, http://www.sebase.org/applications.