A REACTIVE GRASP APPROACH FOR REGRESSION TEST CASE PRIORITIZATION

Camila Loiola Brito Maia  
Laboratório de Computação Natural e Inteligente (LACONI)  
Universidade Estadual do Ceará (UECE)  
Av. Paranjana, 1700 - 60.740-903 – Fortaleza - CE  
camila.maia@gmail.com

Rafael Augusto Ferreira do Carmo  
Laboratório de Computação Natural e Inteligente (LACONI)  
Universidade Estadual do Ceará (UECE)  
Av. Paranjana, 1700 - 60.740-903 – Fortaleza - CE  
carmorafael@gmail.com

Gustavo Augusto Lima de Campos  
Laboratório de Computação Natural e Inteligente (LACONI)  
Universidade Estadual do Ceará (UECE)  
Av. Paranjana, 1700 - 60.740-903 – Fortaleza - CE  
gustavo@larces.uece.br

Jerffeson Teixeira de Souza  
Laboratório de Computação Natural e Inteligente (LACONI)  
Universidade Estadual do Ceará (UECE)  
Av. Paranjana, 1700 - 60.740-903 – Fortaleza - CE  
jeff@larces.uece.br

ABSTRACT

Some modifications in software can affect some functionality that had been working until that point. In order to detect such a problem, the ideal solution would be testing the whole system once again, but there may be insufficient time or resources for this. An alternative solution would be to order the test cases so that the most beneficial tests are executed first, in such a way only a subset of the test cases can be executed with little lost of effectiveness. Such a technique is known as Regression Test Case Prioritization. In this paper, we propose the use of the Reactive GRASP metaheuristic to prioritize test cases, and compare it with others search-based algorithms previously described in the literature. Four programs were used in the experiments. The results demonstrated good coverage performances with some time overhead.

KEYWORDS. Reactive GRASP. Regression Test Case Prioritization. Metaheuristic.
1. Introduction

More than often, when a system is modified, the modifications can affect some functionality that had been working until that point in time. Due to the unpredictability of the effects such a modification may cause to the system’s functionalities, it is recommended to test the system, as a whole or partially, once again every time there is modification. This is commonly known as Regression Testing. Its purpose is to guarantee that the software changes don’t affect the functions that were working previously.

There are basically two ways to perform Regression Tests. The first one is by simply executing all test cases in order to test the entire system once again. Unfortunately, and usually, there may be insufficient resources to allow for the re-execution of all test cases. The second would be to order the test cases so that the most beneficial tests are executed first, in such a way only a subset of the test cases can be executed with little lost of effectiveness. Such a technique is known as Regression Test Case Prioritization.

The Regression Test Case Prioritization problem is highly related to the Regression Test Case Selection problem. Algorithms that solve the first problem can be easily modified so as to solve the second one also. The Regression Test Case Selection problem can be directly modeled as a set covering problem, which is a well-known NP-Hard problem (Cormen et al., 2001).

To order the test cases, it is necessary to consider a base comparison measure. A straightforward measure to evaluate a test case would be based on upon APFD (Average of the Percentage of Faults Detected). Higher APFD numbers mean faster fault detection rates (Rothermel et al. 1999). However, it is not possible to know the faults exposed by a test case in advance, so this value cannot be estimated before testing has taken place. Therefore, the research on test case prioritization concentrates on coverage measures. The following coverage criteria have been commonly used: APBC (Average Percentage Block Coverage), which measures the rate at which a prioritized test suite covers the blocks of the code; APDC (Average Percentage Decision Coverage), which measures the rate at which a prioritized test suite covers the decision statements in the code; and APSC (Average Percentage Statement Coverage), which measures the rate at which a prioritized test suite covers the statements. In this work, these three coverage measures will be considered.

Greedy Algorithms have been employed in many researches regarding test case prioritization, in order to find an optimal ordering (Rothermel et al., 2001). Such Greedy algorithms perform by iteratively adding a single test case to a partially constructed test suite, if this test case covers, as much as possible, some piece of code not covered yet. Despite the vast use, as pointed out by Rothermel et al. (2001) and Li et al. (2007), Greedy Algorithms may not choose the optimal test case ordering. This fact justifies the application of global approaches, i.e., approaches which consider the evaluation of the ordering as a whole, not individually to each test case. In that context, metaheuristics have become the focus in this field.

Metaheuristic search techniques are algorithms that find optimal or near optimal solutions to optimization problems (Glover et al., 2003). The application of Genetic Algorithms, an evolutionary metaheuristic, has been shown to be effective for regression test case prioritization (Walcott et al., 2006. Yoo et al., 2007). We examine in this paper the application of another well-known metaheuristic, GRASP, to this problem.

The remaining of this paper is organized as follows: Section 2 describes other works related to the regression test case prioritization problem and some algorithms which have been previously applied to this problem. These algorithms will be employed in the evaluation of our approach later on the paper; Section 3 describes the GRASP metaheuristic and the proposed algorithm using Reactive GRASP; Section 4 presents the details of the experiments, and Section 5 reports the conclusions of this research.

2. Related Work

This section will report the use of search-based prioritization approaches and
metaheuristics and will describe some algorithms implemented in Li et al. (2007) which will have their performance compared to the approach proposed later on this paper.

2.1. Search-based Prioritization Approaches

The works below employed search-based prioritization approaches, such as greedy and metaheuristic-based solutions.

Elbaum et al. (2004) analyze several prioritization techniques and provide responses to what technique is more suitable for specific test scenarios and their conditions. The metric APFD is calculated through a greedy heuristic. Rothermel et al. (2001) describe a technique that incorporates a Greedy Algorithm, called Optimal Prioritization, which considers the known faults of the program, and the test cases are ordered using the fault detection rates.

Walcott et al. (2006) proposes a test case prioritization technique with a genetic algorithm which reorder test suites based on testing time constraints and code coverage. This technique significantly outperformed other prioritization techniques described in the paper, improving in, on average, 120% the APFD over the others.

Yoo et al. (2007) describe a Pareto approach to prioritize test case suites, based on multiple objectives, such as code coverage, execution cost and fault-detection history. The objective is finding an array of decision variables (test case ordering) that maximize an array of objective functions, as described previously. Three algorithms were compared: a re-formulation of a greedy algorithm (additional greedy algorithm), Non Dominating Sorting Genetic Algorithm (NSGA-II) (Deb et al., 2000), and a variant of NSGA-II, called vNSGA-II. For two objective functions, a genetic algorithm outperformed the additional greedy algorithm, but for some programs the additional greedy algorithm produced the best results. For three objective functions, additional greedy algorithm had reasonable performance.

Li et al. (2007) compare five algorithms: Greedy Algorithm, which adds test cases that achieve the maximum value for the coverage criteria, Additional Greedy Algorithm, which adds test cases that achieve the maximum coverage not already consumed by a partial solution, 2-Optimal Algorithm, which selects two test cases that consume the maximum coverage together, Hill Climbing, which performs local search in a defined neighborhood, and Genetic Algorithm, that generates new test cases based on previous ones. The authors separate test suits in 1,000 small suites of size 8-155, and 1,000 large suites of size 228-4,350. Six C programs were used in the experience, ranging from 374 to 11,148 lines of code. The coverage metrics studied in that work were APBC, APDC and APSC, as described earlier. For each program, the block, decision, and statement coverage data were found by tailor-made version of a commercial tool, Cantata++. The coverage data were obtained over 500 executions for each search algorithm, using a different suite for each execution. For small programs, the performance was almost identical for all algorithms, considering both small and large test suites. The Greedy Algorithm performed the worst and the Genetic Algorithm and Additional Greedy produce the best results. For all coverage criteria, the results were similar.

2.2. Algorithms

This section describes some algorithms which have been used frequently in the literature to deal with the test case prioritization problem, and will have their performance compared to the approach proposed later on this paper.

2.2.1. Ordering Algorithm

The Ordering Algorithm described in this paper is the Greedy Algorithm described in Li et al. (2007). We have decided to change its name since the original algorithm described in Li et al. (2007) is not in fact a greedy algorithm, since it does not perform a locally optimum decision at each interaction.

The Ordering Algorithm performs in the following way: all candidate test cases are
ordered by their coverage. Then, the test case with highest percentage of coverage is added to an initially empty solution. Next, the test case with the second highest percentage is added, and so on, until all test cases have been added.

For example, let APBC be the coverage criterion, and let a partial solution contain two test cases that cover 10 blocks of code. Suppose there are two other test cases that can be added to the solution. The first one covers 8 blocks, but 5 of these were already covered by the current solution. Then, the added coverage is of 3 blocks. The second test case covers only 4 blocks of code, but none of these blocks were covered by the current solution. The Ordering Algorithm selects the first test case, because it covers more blocks overall.

2.2.2. Greedy Algorithm

The Greedy Algorithm described in this paper is the Additional Greedy Algorithm described in Li et al. (2007).

A Greedy Algorithm adds a locally optimal test case to a partial test suite. Starting from an empty solution, the algorithm follows these steps: for each iteration the algorithm adds the test case which gives the major coverage gain to the partial solution.

For example, let a partial solution contain two test cases with 40% coverage. Suppose there are two other test cases that can be added to the solution. The first one covers 30%, but these 30% of code were already covered by the test cases present in the current solution. The second test case covers only 5%, but these pieces of code were not covered by the current solution. Therefore, the second test case provides a higher coverage gain over the first one, and is consequently added to the solution.

2.2.3. Genetic Algorithm

Genetic Algorithm is a type of Evolutionary Algorithm which has been employed extensively to solve optimization problems (Holland, 1975). In this approach, an initial population of solutions - in our case a set of test suites - is randomly generated. Next, new populations are generated iteratively based on the previous population until a stopping criterion is reached (Harman, 2007). The evolution from one to the next population is performed via several genetic operators, including: operations of selection, i.e., the biased choice of which individuals of the current population will reproduce to generate individuals for the new population. This selection prioritizes individuals with high fitness value, which represents how good this solution is. The other two evolutionary operators are: crossover, i.e., the combination of individuals to produce the offspring, and mutation, which randomly changes a particular individual.

In the genetic algorithm proposed by Li et al. (2007), the initial population is produced by selecting test cases randomly from the test case pool. The fitness function is based on the test case position in the current test suite. The fitness value was calculated as follows:

\[ \text{fitness}(pos) = 2 \frac{(pos - 1)}{(n - 1)} \]

where \( pos \) is the test case’s position in actual test suite, and \( n \) is the population size.

The crossover algorithm is the same as the one proposed by Antoniol et al. (2005), and works as follows. Let \( p_1 \) and \( p_2 \) be the parents, and \( o_1 \) and \( o_2 \) be the offspring. A random position \( k \) is selected, and the first \( k \) elements of \( p_1 \) become the first \( k \) elements of \( o_1 \), and the last \( n-k \) elements of \( o_1 \) are the \( n-k \) elements of \( p_2 \) which remain when the \( k \) elements selected from \( p_1 \) are removed from \( p_2 \). In the same way, the first \( k \) elements of \( p_2 \) become the first \( k \) elements of \( o_2 \), and the last \( n-k \) elements of \( o_2 \) are the \( n-k \) elements of \( p_1 \) which remain when the \( k \) elements selected from \( p_2 \) are removed from \( p_1 \). The mutation is performed by randomly exchanging the position of two test cases.

3. Reactive GRASP for Test Case Prioritization

This section is intended to present a novel approach for test case prioritization based on the Reactive GRASP metaheuristic.
3.1. The Reactive GRASP Metaheuristic

Metaheuristics are general search algorithms that find a good solution, sometimes optimal, to optimization problems. In this section we present, in a general fashion, the metaheuristic which will be employed to prioritize test cases by the approach proposed later on this paper.

GRASP (Greedy Randomized Adaptive Search Procedures) is a metaheuristic with two phases: construction and local search (Resende et al., 2001), see Figure 1. In the construction phase, a feasible solution is built, applying some greedy algorithm. The greedy strategy adds to an initially empty solution one element at a time. This algorithm tends to find a local optimum. Therefore, in order to avoid this local best, GRASP uses a randomization strategy as follows. The Restrict Candidate List (RCL) stores the possible elements which can be added at each step in this construction phase. The element to be added is picked randomly from this list. RCL is associated with a parameter named α, which limits the length of the RCL. If α = 0, only the best element - with highest coverage - will be present in the RCL, making the construction process a pure greedy algorithm. Otherwise, if α = 1, the construction phase will be completely random, because all possible elements will be in RCL. The parameter α should be set to calibrate the random and greedy the construction process will be.

![Figure 1. GRASP’s phases](image_url)

In the local search phase, the current solution is replaced by a best solution in its neighborhood. After this process, this local best is compares best solution already found. If it is better, the best solution is replaced by this one.

As can be easily seem, the performance of the GRASP algorithm will strongly depend of the choice of the parameter α. In order to decrease this influence, a GRASP variation, named Reactive GRASP (Resende et al., 2001. Prais et al., 2000), has been proposed. This approach performs GRASP while varying α according to its previous performance. In practice, Reactive GRASP will initially determine a set of possible values for α. Each value will have a probability of being selected in each iteration. Initially, all α probabilities are assigned to 1/n. The probabilities are reevaluated in each iteration, according to the following equation:

\[ p_i = \frac{q_i}{\sum_{j=1}^{m} q_j} \]

, with \( q_i = S^*/A_i \), where \( S^* \) is the incumbent solution and \( A_i \) is the average value of all solutions found with \( \alpha = \alpha_i \). This way, when a particular \( \alpha \) generate a good solution, its probability of being selected in the future will increase. On the other hand, if a bad solution is created, the \( \alpha \) value used in the process will have its selection probability decreased.
3.2. The Reactive GRASP Algorithm

The pseudocode below, in Figure 2, describes the Reactive GRASP algorithm.

```
1 initialize probabilities associated with α (all equal to \frac{1}{n})
2 for k = 1 to max_iterations do
3    α ← select_α(αSet);
4    solution ← run_construction_phase(α);
5    solution ← run_local_search_phase(solution);
6    update_solution(solution, best_solution);
7 end;
8 return best_solution;
```

Figure 2. Reactive GRASP for Test Case Prioritization Pseudocode

The first step initializes the probabilities associated with the choice of each α (line 1). Initially, all probabilities are assigned to 1/n, where n is the length of αSet, the set of α values. Next, the GRASP algorithm runs the construction and local search phases, as described next, until the stopping criterion is reached. For each iteration, the best solution is updated, if the new solution is better.

For each iteration, α is selected as follows, see Figure 3. Let S* be the incumbent solution, and let \( \bar{A}_i \) be the coverage average value of all solutions found with \( \alpha = \alpha_i \), where \( i = 1, \ldots, m \), and m is the number of test cases. The probabilities are reevaluated for each iteration by taking

\[
p_i = \frac{q_i}{\sum_{j=1}^{m} q_j}
\]

, as described in section 3.1.

```
procedure select_α(αSet)
1    α ← α with probability \( p_i = \frac{q_i}{\sum_{j=1}^{m} q_j} \)
2    return α
```

Figure 3. Selection of α

The pseudocode in Figure 4 details the construction phase. For each iteration, one test case which increases the coverage of the existing solution (set of test cases) is selected by a greedy evaluation function. This element is randomly selected from the RCL (Restricted Candidate List), which has the best elements (best coverage values). After the element is incorporated to the partial solution, the RCL is updated. The increment of coverage is then reevaluated.
A Pesquisa Operacional e o uso racional de recursos hídricos

02 a 05/09/08   João Pessoa, PB

Figure 4. Reactive GRASP for Test Case Prioritization - Construction Phase

The αSet is updated after the solution is found, in order to change the selection probabilities of the αSet elements. This update is detailed as follows, in figure 5:

```
procedure update_αSet(solution)
  1. update probabilities of all α in αSet, using
  2. \[ p_i = \frac{q_i}{\sum_{j=1}^{m} q_j} \]

Figure 5. Update of α
```

After the construction phase, a local search phase is executed, in order to improve the generated solution. This phase is important to avoid the problems mentioned by Rothermel et al. (2001) and Li et al. (2007), where greedy algorithms may fail to choose the optimal test case ordering.

The pseudocode for the local search is described in Figure 6:

```
1. while s not locally optimal do
  2. Find \( s' \in \text{Neighbour}(s) \) with \( f(s') < f(s) \);
  3. \( s \leftarrow s' \);
  4. end;
  5. return s;
```

Figure 6. Reactive GRASP for Test Case Prioritization – Local Search Phase

Let \( s \) be the test suite generated by the construction phase. The local search is performed as follows: the first test case on the test suite is exchanged with the other test cases, one at a time, i.e., \( n-1 \) 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 better - in terms of coverage - than the original one, it replaces the original solution. This strategy was chosen because, even with very little computational effort, any exchange with the first test case can generate a very significant difference in coverage. In addition, it would be prohibitive to test all possible exchanges, since it would generate \( n^2 \) new test suits, instead of \( n-1 \), in which most of them would exchange the last elements, with no significant difference in coverage.

4. Empirical Evaluation

In order to evaluate the performance of the proposed approach, a series of empirical testes were executed. More specifically, the experiments were designed to answer the following question:
1) How does the Reactive GRASP approach compare, in terms of coverage and time performance, to other search-based algorithms, including the Genetic, Ordering and Greedy Algorithms?

In addition to this result, the experiments can confirm results previously described in the literature, including the performance of the Greedy algorithm.

4.1. Experimental Design

Four programs were used in the tests: print_tokens, print_tokens2, schedule, and schedule2. These programs were assembled by researchers at Siemens Corporate Research (Hutchins et al., 1999). The Table 1 below presents the programs’ characteristics:

Table 1. Programs used in the Evaluation

<table>
<thead>
<tr>
<th>Program</th>
<th>LoC</th>
<th>Blocks</th>
<th>Decisions</th>
<th>Test Pool Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Print_tokens</td>
<td>726</td>
<td>126</td>
<td>123</td>
<td>4,130</td>
</tr>
<tr>
<td>Print_tokens2</td>
<td>570</td>
<td>103</td>
<td>154</td>
<td>4,115</td>
</tr>
<tr>
<td>Schedule</td>
<td>412</td>
<td>46</td>
<td>56</td>
<td>2,650</td>
</tr>
<tr>
<td>Schedule2</td>
<td>374</td>
<td>53</td>
<td>74</td>
<td>2,710</td>
</tr>
</tbody>
</table>

Besides Reactive GRASP, other search algorithms have also been implemented, in order to compare their effectiveness. They are: Ordering Algorithm, Greedy Algorithm and Genetic Algorithm. These algorithms were implemented exactly as described in the section 3 of this paper. For the Genetic Algorithm, as presented by Li et al. (2007), the population size was set at 50 individuals and the algorithm was terminated after 100 generations. Stochastic universal sampling was used in selection and mutation, the crossover probability (per individual) was set to 0.8 and the mutation probability was set to 0.1. For the Reactive GRASP approach, the maximum number of iterations was set, by preliminary experimentation, to 300.

In the experiments, we considered the three coverage criteria described earlier (APBC, APDC and APSBC). In addition, we considered different percentages of the pool of test cases. For example, if the percentage is 5%, we chose 5% of test cases, randomly, from the pool, to compare the performance of the algorithms over the four programs. We tested with 1%, 2%, 3%, 5%, 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90%, and 100%. Each algorithm was executed 10 times for each coverage criterion and each percentage.

The pools of test cases used in the experiments were collected from SEBASE (2008). The test cases used are composed of ‘0’ s and ‘1’ s, where ‘0’ represents ‘code not covered’, and ‘1’ represents ‘code covered’. The length of a test case is the quantity of portions of code of the program. For example, when we are analyzing the decision coverage, the length of the test cases is the quantity of decisions on the program, and ‘0’ for the first decision means that the first decision is not covered by the test suite, a ‘1’ for the second decision means that the second decision is covered by the test suite, and so on.

All experiments were performed on Ubuntu Linux workstations with kernel 2.6.22-14, a Core Duo processor, and 1 GB of main memory. The programs used in the experiment were implemented in the Java programming language.

4.2. Results

The results are presented in Tables 2 and 3.

Table 3 details the average of the coverage percentage achieved for each coverage criterion, each program and each algorithm. The TSSp column is the percentage of test cases selected from the test case pool.

In order to facilitate the comparison of the algorithms, we have calculated the difference of the mean coverage percentage for each pair of algorithms and the significance of this difference using the t-test (see Table 2). The level of significance was defined at 0.05, i.e., if the significance level in difference between the coverage percentage of two algorithms is equal or more than 5% (or 0.05), we will consider the difference in performance was not significant.
The mean differences on time execution are also presented in Table 2, in seconds.

<table>
<thead>
<tr>
<th>Algorithm (x)</th>
<th>Algorithm (y)</th>
<th>Mean Coverage Difference (%) (x - y)</th>
<th>Coverage Difference Significance (t-test)</th>
<th>Time Mean Difference (sec) (x - y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ordering Algorithm</td>
<td>Greedy Algorithm</td>
<td>-0.093</td>
<td>0.000</td>
<td>-1,491</td>
</tr>
<tr>
<td></td>
<td>Genetic Algorithm</td>
<td>0.351</td>
<td>0.000</td>
<td>-8,436</td>
</tr>
<tr>
<td></td>
<td>Reactive GRASP</td>
<td>-0.092</td>
<td>0.025</td>
<td>-49,947</td>
</tr>
<tr>
<td>Greedy Algorithm</td>
<td>Ordering Algorithm</td>
<td>0.093</td>
<td>0.000</td>
<td>1,491</td>
</tr>
<tr>
<td></td>
<td>Genetic Algorithm</td>
<td>0.445</td>
<td>0.000</td>
<td>-6,945</td>
</tr>
<tr>
<td></td>
<td>Reactive GRASP</td>
<td>0.001</td>
<td>0.490</td>
<td>-48,456</td>
</tr>
<tr>
<td>Genetic Algorithm</td>
<td>Ordering Algorithm</td>
<td>-0.351</td>
<td>0.000</td>
<td>8,436</td>
</tr>
<tr>
<td></td>
<td>Greedy Algorithm</td>
<td>-0.445</td>
<td>0.000</td>
<td>6,945</td>
</tr>
<tr>
<td></td>
<td>Reactive GRASP</td>
<td>-0.443</td>
<td>0.000</td>
<td>-41,511</td>
</tr>
<tr>
<td>Reactive GRASP</td>
<td>Ordering Algorithm</td>
<td>0.092</td>
<td>0.025</td>
<td>49,947</td>
</tr>
<tr>
<td></td>
<td>Greedy Algorithm</td>
<td>-0.001</td>
<td>0.490</td>
<td>48,456</td>
</tr>
<tr>
<td></td>
<td>Genetic Algorithm</td>
<td>0.443</td>
<td>0.000</td>
<td>41,511</td>
</tr>
</tbody>
</table>

4.3. Analysis

Analyzing the results obtained from the experiments, which were detailed in Table 3 and summarized in Table 2, several relevant results can be pointed out. First, the Greedy Algorithm had the best performance of all. It performed significantly better than both the Ordering Algorithm (with significance at 0.000) and the Genetic Algorithm (at 0.000). The good performance of the Greedy Algorithm had already been demonstrated in several works, including (Li et al, 2007. Yoo et al., 2007).

Surprisingly, the Genetic Algorithm approach performed the worst in our evaluation. Indeed, even replicating the parameterization presented in Li et al. (2007), we could not replicate the results presented in that article. This fact may be explained, at least partially, by the highly stochastic nature of the Genetic Algorithm.

The Reactive GRASP algorithm had the second best performance. This strategy also significantly outperformed both the Ordering Algorithm (with significance at 0.025) and the Genetic Algorithm (at 0.000). When compared to the Greedy Algorithm, there were no significant differences in terms of coverage. In fact, the results showed that the difference between the mean coverage obtained by these two algorithms was only 0.001 %. Comparing both metaheuristic-based approaches, the better performance obtained by the Reactive GRASP algorithm over the Genetic Algorithm is clear, not only by the general significance level of the mean coverage, at the 0.000 level, but also for the number of times the GRASP approach performed better. For all 168 experiments, the Genetic Algorithm generated a better coverage only once - for the block criterion, the schedule program and 100% of tests begin considered. The two algorithms tied also once. For all other tests, the Reactive GRASP performed better. These results qualify the Reactive GRASP algorithm as a good global coverage solution for the prioritization test case problem.

It is also important to mention that the results were consistently similar across coverage criteria. This fact had already been reported by Li et al. (2007). It suggests that there is no need to consider more than one criterion in order to generate good prioritizations of test cases. In addition, we could not find any significant difference in the coverage performance of all algorithms when varying the percentage of test cases being considered. Note that we’ve tried from 1 to 100% of test cases for each program and criterion, and the performances of all algorithms remained unaltered. This demonstrated that the ability of the four algorithms discussed here is not related to the number of test cases required to order.
<table>
<thead>
<tr>
<th>Block Coverage %</th>
<th>Decision Coverage %</th>
<th>Statement Coverage %</th>
</tr>
</thead>
</table>

In terms of efficiency, as expected, the use of global approaches, such as both metaheuristic-based algorithms evaluated here, adds an overhead to the process. Considering time efficiency,
one can see from Table 2 that the Ordering Algorithm performed more efficiently than all other solutions. This algorithm was, on average, 1.491 seconds faster than Greedy Algorithm, 8.436 faster than the Genetic Algorithm and almost 50 seconds faster than the Reactive GRASP approach. This result demonstrates, once again, the great performance obtained by the Greedy Algorithm compared to its counterpart, since it was significantly better, performance-wise, and achieved these results with a very similar execution time. On the other spectrum, we had the Reactive GRASP algorithm, which performed on average 48,456 seconds slower than the Greedy Algorithm and 41,511 seconds slower than the Genetic Algorithm. Since the programs considered in this evaluation were not large in any practical sense, these results may suggest the Reactive GRASP can be restrictive in practical situations. In fact, it suggests that new evaluations should be considered in order to verify whether the good performance obtained by this algorithm was, at least partially, obtained through high computational effort. In favor of both metaheuristic-based approaches is the fact that one may calibrate the time required for prioritization depending on time constraints and characteristics of programs and test cases. This flexibility is not present in the Ordering Algorithms.

Table 4 summarizes the results described above.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Coverage performance</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ordering</td>
<td>● It was better only than Genetic Algorithm.</td>
<td>Fast</td>
</tr>
<tr>
<td>Greedy</td>
<td>● It had the best performance of all.</td>
<td>Fast</td>
</tr>
<tr>
<td>Genetic</td>
<td>● It had the worst performance of all;</td>
<td>Medium</td>
</tr>
<tr>
<td></td>
<td>● It generated a better coverage only once.</td>
<td></td>
</tr>
<tr>
<td>Reactive GRASP</td>
<td>● It had the second best performance of all;</td>
<td>Slow</td>
</tr>
<tr>
<td></td>
<td>● No significant difference between Reactive GRASP and Greedy Algorithm;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>● It outperformed the Genetic Algorithm (it’s a good global solution).</td>
<td></td>
</tr>
</tbody>
</table>

5. Conclusions and Future Works

Regression testing is an important component of any software development process. Test Case Prioritization is intended to avoid the execution of all test cases every time a change is made to the system. Modeled as an optimization problem, this prioritization problem can be solved with well-known search-based approaches, including metaheuristics.

This paper proposed the use of the Reactive GRASP metaheuristic for the regression test case prioritization problem and compared its performance with other solutions previously reported in the literature. Since the Reactive GRASP algorithm performed significantly better - in terms of coverage performance - than the Genetic Algorithm and similarly to the Greedy Algorithm and it avoids the problems mentioned by Rothermel et al. (2001) and Li et al. (2007), where greedy algorithms may fail to choose the optimal test case ordering, the use of the Reactive GRASP algorithm is indicated to the problem of test case prioritization, especially when time constraints are not too critical, since the Reactive GRASP added a considerable overhead.

As future work, larger programs, which represent more realistic situations, will be considered in the evaluation. This will indicate whether the low time performance obtained by the Reactive GRASP will pose a problem in practical environments. In addition, we will evaluate the Reactive GRASP with different number of iterations. This will elucidate whether its good performance was due to its intelligent search heuristics or its computational effort. Finally, other metaheuristics will be considered, including Tabu Search, Simulated Annealing and VNS, among others.
References


Holland, J.H. (1975), Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence, Univ. of Michigan


