Hindawi Publishing Corporation
Advances in Software Engineering
Volume 2010, Article ID 428521, 18 pages
doi:10.1155/2010/428521

Research Article

Automated Test Case Prioritization with Reactive GRASP

Camila Loiola Brito Maia, Rafael Augusto Ferreira do Carmo, Fabrício Gomes de Freitas, Gustavo Augusto Lima de Campos, and Jerffeson Teixeira de Souza

Optimization in Software Engineering Group (GOES.UECE), Natural and Intelligent Computing Lab (LACONI), State University of Ceará (UECE), Avenue Paranjana 1700, Fortaleza, 60740-903 Ceará, Brazil

Correspondence should be addressed to Camila Loiola Brito Maia, camila.maia@gmail.com

Received 15 June 2009; Revised 17 September 2009; Accepted 14 October 2009

Academic Editor: Phillip Laplante

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 approach. An alternative solution is 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. We also compare this metaheuristic with other search-based algorithms previously described in literature. Five programs were used in the experiments. The experimental results demonstrated good coverage performance with some time overhead for the proposed technique. It also demonstrated a high stability of the results generated by the proposed approach.

1. Introduction

More than often, when a system is modified, the modifications may affect some functionality that had been working until that point in time. Due to the unpredictability of the effects that such modifications may cause to the system's functionalities, it is recommended to test the system, as a whole or partially, once again every time a modification takes place. This is commonly known as regression testing. Its purpose is to guarantee that the software modifications have not affected the functions that were working previously.

A test case is a set of tests performed in a sequence and related to a test objective [1], and a test suite is a set of test cases that will execute sequentially. There are basically two ways to perform regression tests. The first one is by reexecuting all test cases in order to test the entire system once again. Unfortunately, and usually, there may not be sufficient resources to allow the reexecution of all test cases every time a modification is introduced. Another way to perform regression test is to order the test cases in respect to their beneficial factor to some attribute, such as coverage, and reexecute the test cases according to that ordering. In doing this, the most beneficial test cases would be 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. When the time required to reexecute an entire test suite is sufficiently long, test case prioritization may be beneficial because meeting testing goals earlier can yield meaningful benefits [2].

According to Myers [3], since exhaustive testing is out of question, the objective should be to maximize the yield on the testing investment by maximizing the number of errors found by a finite number of test cases. As Fewster stated in [1], software testing needs to be effective at finding any defects which are there, but it should also be efficient by performing the tests as quickly and cheaply as possible.

The regression test case prioritization problem is closely related to the regression test case selection problem. The Regression Test Case Selection problem can be directly modeled as a set covering problem, which is a well-known NP-Hard problem [4]. This fact points to the complexity of the Test Case Prioritization problem.

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 APFD (Average of the Percentage of Faults Detected). Higher APFD numbers

mean faster fault detection rates [5]. 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.

As an example, consider a test suite T containing n test cases that covers a set B of m blocks. Let TB_i be the first test case in the order T' of T that covers block i. The APBC for ordering T' is given by the following equation (equivalent for the APDC and APSC metrics) [6]:

$$\text{APBC} = 1 - \frac{TB_1 + TB_2 + \cdots + TB_m}{nm} + \frac{1}{2n}. \quad (1)$$

Greedy algorithms have been employed in many researches regarding test case prioritization, in order to find an optimal ordering [2]. 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 wide use, as pointed out by Rothermel [2] and Li et al. [6], Greedy algorithms may not choose the optimal test case ordering. This fact justifies the application of global approaches, that is, 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. In this work, we have tested Reactive GRASP, not yet used for test case prioritization.

Metaheuristic search techniques are algorithms that may find optimal or near optimal solutions to optimization problems [7]. In the context of software engineering, a new research field has emerged by the application of search techniques, especially metaheuristics, to well-known complex software engineering problems. This new field has been named SBSE (Search-Based Software Engineering). In this field, the software engineering problems are modeled as optimization problems, with the definitions of an objective function—or a set of functions—and a set of constraints. The solutions to the problems are found by the application of search techniques.

The application of genetic algorithms, an evolutionary metaheuristic, has been shown to be effective for regression test case prioritization [8, 9]. We examine in this paper the application of another well-known metaheuristic, GRASP, not applied yet neither to the regression test case selection problem nor to any other search-based software engineering problem. The GRASP metaheuristic was considered due to its good performance reported by several studies in solving complex optimization problems.

The remaining of this paper is organized as follows: Section 2 describes works related to the regression test case prioritization problem and introduces some algorithms which have been 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 and states future works.

2. Related Work

This section reports the use of search-based prioritization approaches and metaheuristics. Some algorithms implemented in [6] by Li et al. which will have their performance compared to that of the approach proposed later on this paper will also be described.

2.1. Search-Based Prioritization Approaches. The works below employed search-based prioritization approaches, such as greedy- and metaheuristic-based solutions.

Elbaum et al. [10] analyze several prioritization techniques and provide responses to which technique is more suitable for specific test scenarios and their conditions. The metric APFD is calculated through a greedy heuristic. Rothermel et al. [2] 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. [8] propose a test case prioritization technique with a genetic algorithm which reorders 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 and Harman [9] 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 to find an array of decision variables (test case ordering) that maximize an array of objective functions. Three algorithms were compared: a reformulation of a Greedy algorithm (Additional Greedy algorithm), Non-Dominating Sorting Genetic Algorithm (NSGA-II) [11], and a variant of NSGA-II, 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. [6] 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, which generates new test cases based on previous ones. The authors separated test suites 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 LoC (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 and coverage criteria, considering both small and large test suites. The Greedy algorithm performed the worst and the genetic algorithm and Additional Greedy algorithm produced the best results.

2.2. Algorithms. This section describes some algorithms which have been used frequently in literature to deal with the test case prioritization problem. The performance of them will be compared to that of the approach proposed later on this paper.

2.2.1. Greedy Algorithm. The Greedy Algorithm performs in the following way: all candidate test cases are ordered by their coverage. Then, the test case with the highest percentage of coverage is then 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 100 blocks of code. Suppose there are two other test cases that can be added to the solution. The first one covers 80 blocks, but 50 of these were already covered by the current solution. Then, this solution covers 80% of the blocks, but the actual added coverage of this test case is of 30% of coverage (30 blocks). The second test case covers 40 blocks of code, but none of these blocks was covered by the current solution. This means that this solution covers 40% of the blocks. The Greedy algorithm would select the first test case, because it has greater percentage of block coverage overall.

2.2.2. Additional Greedy Algorithm. The Additional 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.

Let us use the same example from Section 2.2.1. Let a partial solution contain two test cases that cover 100 blocks of code. There are two remaining test cases: the first one covers 80 blocks, but 50 of these were already covered; the second one covers 40 blocks of code, none of these already covered. The first solution represents an actual 30% of coverage and the second one represents 40% of coverage. The Additional Greedy algorithm would select the second test case, because that solution has greater coverage factor related to the current partial solution.

2.2.3. Genetic Algorithm. Genetic algorithm is a type of Evolutionary Algorithm which has been employed extensively to solve optimization problems [12]. In this algorithm, an initial population of solutions—in our case a set of test suites—is randomly generated. The procedure then works, until a stopping criterion is reached, as new populations are generated based on the previous one [13]. The evolution from one population to the next one is performed via "genetic operators", including operations of selection, that is, 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 genetic operators are crossover, that is, 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. [6], 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:

$$\text{fitness}(pos) = 2 \cdot \frac{(pos - 1)}{(n - 1)}, \tag{2}$$

where pos is the test case's position in the current test suite and n is the population size.

The crossover algorithm follows the ordering chromosome crossover style adopted by Antoniol [14] and used in [6] by Li et al. for the genetic algorithm in the experiments. It works as follows. Let p_1 and p_2 be the parents, and let 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.

2.2.4. 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.

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

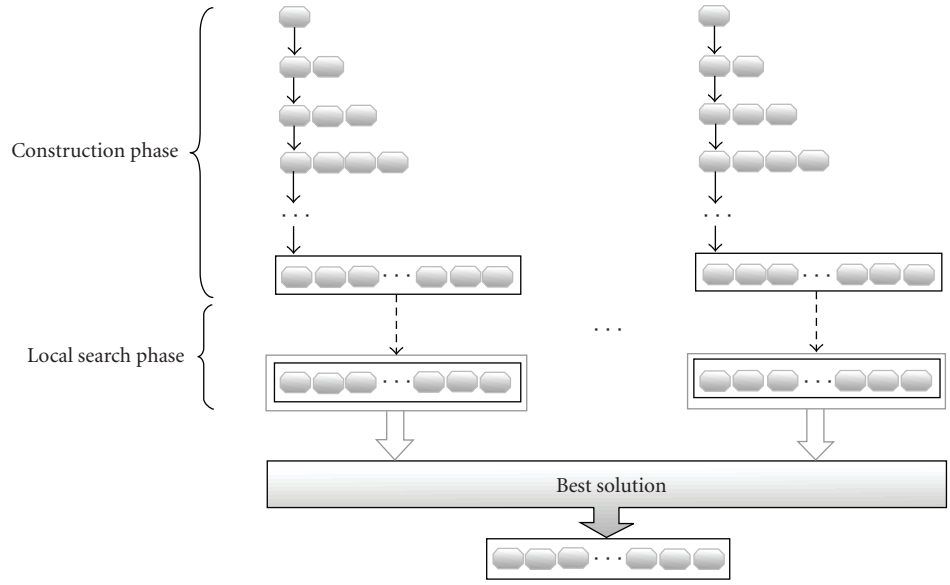

Figure 1: GRASP's phases.

employed to prioritize test cases by the approach proposed later on this paper.

GRASP (Greedy Randomized Adaptative Search Procedures) is a metaheuristic with two phases: construction and local search [15]. This metaheuristic is defined as a multistart algorithm, since the procedure is executed multiple times in order to get the best solution found overall; see Figure 1.

In the construction phase, a feasible solution is built by applying some Greedy algorithm. The greedy strategy used in GRASP is to add 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 greedy 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 $\alpha = 0$, only the best element—with highest coverage—will be present in the RCL, making the construction process a pure Greedy algorithm. Otherwise, if $\alpha = 1$, the construction phase will be completely random, because all possible elements will be in RCL. The parameter α should be set to calibrate how random and greedy the construction process will be. The found solution is then used in the local search phase.

In the local search phase, the aim is to find the best solution in the current solution neighborhood. Indeed, a local search is executed in order to replace the current solution by the local optimum in its neighborhood. After this process, this local optimum is compared with the best local optimum solution found in earlier iterations. If the local optimum just found is better, then this is set to be the best solution already found. Otherwise, there is no replacement.

As can be easily seen, the performance of the GRASP algorithm will strongly depend on the choice of the parameter α. In order to decrease this influence, a GRASP variation named Reactive GRASP [15, 16] has been proposed. This approach performs GRASP while varying the values of α according to their 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 $1/n$, where n is the quantity of α. For each one of the i values of α, the probabilities p_i are reevaluated for each iteration, according to the following equation:

$$p_i = \frac{q_i}{\sum_{j=1}^{n} qj}, \tag{3}$$

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 α generates a good solution, its probability, given by p_i, of being selected in the future is increased. On the other hand, if a bad solution is created, the α value used in the process will have its selection probability decreased.

3.2. The Reactive GRASP Algorithm. The pseudocode below, in Algorithm 1, describes the Reactive GRASP algorithm.

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 α Set, 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 when the new solution is better.

For each iteration, α is selected as follows; see Algorithm 2. Let S^* be the incumbent solution, and let 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. As

```
(1) initialize probabilities associated
with α (all equal to 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;
```

Algorithm 1: Reactive GRASP for Test Case Prioritization.

```
procedure select_α(αSet)
(1) α ← α with probability p_i = q_i / Σ_{j=1}^{m} qj
(2) return α
```

Algorithm 2: Selection of α.

described in Section 3.1, the probabilities p_i are reevaluated at each iteration by taking

$$p_i = \frac{q_i}{\sum_{j=1}^{m} qj}. \quad (4)$$

The pseudocode in Algorithm 3 details the construction phase. For each iteration, one test case which increases the coverage of the current 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, that is, the best coverage values. After the element is incorporated to the partial solution, the RCL is updated. The increment of coverage is then reevaluated.

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 detailed in Algorithm 4.

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 [2] and Li et al. [6], where Greedy algorithms may fail to choose the optimal test case ordering. The pseudocode for the local search is described in Algorithm 5.

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, that is, $n - 1$ new test suites are generated, exchanging the first test case with the ith 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 suites, instead of $n - 1$, in which most of them would exchange the last elements, with no significant difference in coverage.

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

Algorithm 3: Reactive GRASP for Test Case Prioritization, Construction Phase.

```
procedure update_αSet(solution)
(1) update probabilities of all α in αSet, using
                p_i = q_i / Σ_{j=1}^{m} qj
```

Algorithm 4: Update of α.

```
(1) while s not locally optimal do
(2)    Find s' ∈ Neighbour(s) with f(s') < f(s);
(3)    s ← s';
(4) end;
(5) return s;
```

Algorithm 5: Reactive GRASP for Test Case Prioritization, Local Search Phase.

4. Empirical Evaluation

In order to evaluate the performance of the proposed approach, a series of empirical tests was 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 performances—to other search-based algorithms, including Greedy algorithm, Additional Greedy algorithm, genetic algorithm, and Simulated Annealing?

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

Table 1: Programs used in the Evaluation.

Program	LoC	Blocks	Decisions	Test Pool Size
Print_tokens	726	126	123	4,130
Print_tokens2	570	103	154	4,115
Schedule	412	46	56	2,650
Schedule2	374	53	74	2,710
Space	9,564	869	1,068	13,585

4.1. Experimental Design. Four small programs (print_tokens, print_tokens2, schedule, and schedule2) and a larger program (space) were used in the tests. These programs were assembled by researchers at Siemens Corporate Research [17] and are the same Siemens' programs used in Li et al. [6] for the experiments regarding test case prioritization. Table 1 describes the five programs' characteristics.

Besides Reactive GRASP, other search algorithms have also been implemented, in order to compare their effectiveness. They are Greedy algorithm, Additional Greedy algorithm, genetic algorithm, and simulated annealing. These algorithms were implemented exactly as described in Section 3 of this paper. For the genetic algorithm, as presented by Li et al. [6], 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.

For the simulated annealing approach, the initial temperature was set to a random number between 20 and 99. For each iteration, the new temperature is given by the following steps:

(1) $dividend = actualTemperature + initialTemperature$,

(2) $divisor = 1 + \log_{10} 1$,

(3) $new\ temperature = \dfrac{dividend}{divisor}$.

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%, 5% of test cases are randomly chosen from the pool to compare the performance of the algorithms. We tested with 1%, 2%, 3%, 5%, 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90%, and 100% for the four small programs and 1%, 5%, 10%, 20%, 30%, 40%, and 50% for space. Each algorithm was executed 10 times for the four small programs and 1 time for the space program, for each coverage criterion and each percentage.

The pools of test cases used in the experiments were collected from SEBASE [18]. 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. In the APDC, a "0" for the first decision means that the first decision is not covered by the test suite and 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 using the Java programing language.

4.2. Results. The results are presented in Tables 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, and 18 and Figures 2 to 17, separating the four small programs from the space program. Tables 2, 3, 4, and 5 detail the average of 10 executions of the coverage percentage achieved for each coverage criterion and each algorithm for printtokens, printtokens2, schedule, and schedule2, respectively. Table 12 has this information regarding the space program. The TSSp column is the percentage of test cases selected from the test case pool. The mean differences on time execution in seconds are also presented in Tables 6 and 16, for small programs and space, respectively.

Tables 7 and 14 show the weighted average for the metrics (APBC, APDC, and APSC) for each algorithm. Figures 2 to 17 demonstrate a comparison among the algorithms for the metrics APBC, APDC, and APSC, for the small programs and space program.

4.3. Analysis. Analyzing the results obtained from the experiments, which are detailed in Tables 2, 3, 4, 5, and 9 and summarized in Tables 6 and 13, several relevant results can be pointed out. First, the Additional Greedy algorithm had the best performance in effectiveness of all tests. It performed significantly better than the Greedy algorithm, the genetic algorithm, and simulated annealing, both for the four small programs and for the space program. The good performance of the Additional Greedy algorithm had already been demonstrated in several works, including Li et al. [6] and Yoo and Harman [9].

4.3.1. Analysis for the Four Small Programs. The Reactive GRASP algorithm had the second best performance. This approach also significantly outperformed the Greedy algorithm, the genetic algorithm, and simulated annealing, considering the coverage results. When compared to the Additional Greedy algorithm, there were no significant differences in terms of coverage. Comparing the metaheuristic-based approaches, the better performance obtained by the Reactive GRASP algorithm over genetic algorithm and simulated annealing was clear.

In 168 experiments, the genetic algorithm generated a better coverage only once (block criterion, the schedule program, and 100% of tests being considered). The two algorithms tied also once. For all other tests, the Reactive GRASP outperformed the genetic algorithm. The genetic algorithm approach performed the fourth best in our evaluation. In Li et al. [6], the genetic algorithm was also worse than the Additional Greedy algorithm. The results

Table 2: Results of Coverage Criteria (Average of 10 Executions), Program Print-tokens.

Block Coverage %					
TSSp	Greedy	Additional Greedy	Genetic Algorithm	Simulated Annealing	Reactive GRASP
1%	96.6591	98.235	96.6893	96.1242	97.808
2%	98.3209	99.2101	98.3954	98.3113	99.0552
3%	98.6763	99.5519	98.5483	98.5553	99.3612
5%	98.5054	99.6909	98.8988	98.9896	99.5046
10%	98.2116	99.8527	99.2378	99.3898	99.7659
20%	98.266	99.9317	99.2378	99.6414	99.8793
30%	98.3855	99.9568	99.6603	99.6879	99.9204
40%	98.3948	99.9675	99.7829	99.736	99.9457
50%	98.4064	99.9747	99.8321	99.8213	99.9627
60%	98.4097	99.979	99.8666	99.8473	99.9622
70%	98.4133	99.9818	99.8538	99.8698	99.9724
80%	98.4145	99.9841	99.8803	99.8657	99.9768
90%	98.4169	99.9859	99.9013	99.8958	99.9783
100%	98.418	99.9873	99.9001	99.8895	99.9775
Decision Coverage %					
TSSp	Greedy	Additional Greedy	Genetic Algorithm	Simulated Annealing	Reactive GRASP
1%	96.7692	98.3836	96.9125	95.9213	98.1204
2%	98.0184	99.1429	97.9792	98.2299	98.8529
3%	98.5569	99.4499	98.3785	98.0762	99.2886
5%	98.4898	99.6971	98.7105	98.7513	99.4631
10%	98.1375	99.8462	98.8659	99.1759	99.697
20%	98.2486	99.928	99.3886	99.5111	99.8668
30%	98.3131	99.952	99.587	99.6955	99.9061
40%	98.3388	98.3388	99.7137	99.7505	99.9237
50%	98.3437	99.9712	99.7305	99.78	99.9386
60%	98.358	99.9766	99.817	99.8235	99.959
70%	98.3633	99.9799	99.8109	99.7979	99.9543
80%	98.3651	99.9821	99.8631	99.8447	99.9663
90%	98.4169	99.9859	99.9013	99.8541	99.9783
100%	98.418	99.9873	99.9001	99.869	99.9775
Statement Coverage %					
TSSp	Greedy	Additional Greedy	Genetic Algorithm	Simulated Annealing	Reactive GRASP
1%	97.2989	98.3561	97.0141	97.251	98.0439
2%	97.7834	99.2557	98.0175	98.576	98.9675
3%	98.0255	99.4632	98.5163	98.5633	99.2356
5%	97.8912	99.6826	98.5167	99.0268	99.4431
10%	97.8137	99.8534	99.1497	99.3131	99.681
20%	98.0009	99.9264	99.5024	99.5551	99.8554
30%	98.0551	99.954	99.6815	99.7151	99.9079
40%	98.0661	99.9656	99.7342	99.7677	99.9296
50%	98.0705	99.9724	99.8123	99.8108	99.9464
60%	98.0756	99.9773	99.8348	99.8456	99.9598
70%	98.0887	99.9805	99.8641	99.8633	99.9704
80%	98.088	99.9831	99.89	99.8649	99.9682
90%	98.0924	99.985	99.9026	99.8819	99.9709
100%	98.0943	99.9865	99.8998	99.8897	99.977

Table 3: Results of Coverage Criteria (Average of 10 Executions), Program Print-tokens2.

Block Coverage%					
TSSp	Greedy	Additional Greedy	Genetic Algorithm	Simulated Annealing	Reactive GRASP
1%	97.233	98.3518	97.6629	98.042	98.1576
2%	98.3869	99.2665	98.6723	98.8302	99.208
3%	97.9525	99.5122	98.8576	99.1817	99.3274
5%	98.1407	99.711	99.2379	99.3382	99.5932
10%	98.131	99.8564	99.5558	99.6731	99.7994
20%	98.01	99.9293	99.7894	99.8015	99.8689
30%	98.0309	99.9535	99.8269	99.839	99.9239
40%	98.0462	99.9656	99.8602	99.8957	99.9495
50%	98.0569	99.9727	99.9166	99.9106	99.9653
60%	98.0589	99.977	99.9165	99.9269	99.9689
70%	98.0611	99.9805	99.9264	99.9236	99.9756
80%	98.0632	99.9828	99.9383	99.9261	99.9778
90%	98.0663	99.9849	99.9543	99.9385	99.9796
100%	98.0671	99.9864	99.9562	99.9434	99.9811
Decision Coverage %					
TSSp	Greedy	Additional Greedy	Genetic Algorithm	Simulated Annealing	Reactive GRASP
1%	97.05	98.3055	97.2108	97.6375	98.0859
2%	98.6637	99.2489	98.4368	98.5244	99.0987
3%	98.5798	99.5496	98.8814	99.0411	99.4344
5%	98.5903	99.7371	99.3676	99.2289	99.6772
10%	98.5673	99.8628	99.5183	99.6528	99.8118
20%	98.6351	99.9353	99.7939	99.8084	99.913
30%	98.6747	99.9593	99.8615	99.8405	99.9482
40%	98.6837	99.9692	99.8836	99.8802	99.9556
50%	96.1134	99.9552	99.8269	99.8992	99.9318
60%	98.6948	99.9795	99.9181	99.9109	99.9751
70%	98.6964	99.9826	99.9358	99.9302	99.9768
80%	98.6985	99.9848	99.9478	99.931	99.9788
90%	98.0663	99.9849	99.9543	99.9409	99.9796
100%	98.0671	99.9864	99.9562	99.9424	99.9811
Statement Coverage %					
TSSp	Greedy	Additional Greedy	Genetic Algorithm	Simulated Annealing	Reactive GRASP
1%	97.4458	98.3742	97.8234	97.453	98.2804
2%	98.7611	99.2552	98.755	98.5444	99.0653
3%	98.3634	99.4745	98.9385	98.9165	99.3279
5%	97.8694	99.6856	99.1507	99.3858	99.5327
10%	98.0271	99.8494	99.5268	99.6258	99.7906
20%	98.1264	99.927	99.7455	99.7283	99.9086
30%	97.9467	99.9518	99.8533	99.8297	99.9328
40%	97.9653	99.9645	99.8833	99.864	99.9564
50%	97.9762	99.9717	99.9126	99.8891	99.9584
60%	97.9792	99.9768	99.9162	99.905	99.9644
70%	97.9851	99.9803	99.9265	99.9156	99.9708
80%	97.9854	99.9827	99.9399	99.9187	99.9759
90%	97.9877	99.9847	99.9399	99.9288	99.9789
100%	97.9894	99.9863	99.9477	99.9262	99.9791

Table 4: Results of Coverage Criteria (Average of 10 Executions), Program Schedule.

Block Coverage %					
TSSp	Greedy	Additional Greedy	Genetic Algorithm	Simulated Annealing	Reactive GRASP
1%	96.6505	98.2286	98.2286	97.9387	98.2286
2%	96.5053	99.0237	98.9499	98.8596	99.0073
3%	96.451	99.3315	99.2336	99.1955	99.2445
5%	95.6489	99.5652	99.2481	99.4066	99.3233
10%	95.2551	99.767	99.5586	99.6455	99.7013
20%	95.9548	99.8884	99.7604	99.7497	99.8589
30%	95.8225	99.9224	99.8219	99.8442	99.8918
40%	96.0783	99.9429	99.8995	99.8982	99.9163
50%	96.3159	99.9553	99.9051	99.899	99.9396
60%	96.9283	99.9644	99.918	99.9156	99.9546
70%	97.0744	99.9695	99.9235	99.9322	99.9643
80%	97.0955	99.9733	99.9464	99.9411	99.9649
90%	97.1171	99.9763	99.9474	99.946	99.9704
100%	97.0495	99.9786	99.9573	99.9454	99.7013
Decision Coverage %					
TSSp	Greedy	Additional Greedy	Genetic Algorithm	Simulated Annealing	Reactive GRASP
1%	96.3492	98.2671	98.0952	98.0092	98.2407
2%	95.9838	99.0566	98.7129	98.7364	98.9218
3%	95.933	99.3303	98.9107	98.9575	99.2589
5%	95.1047	99.5327	98.6224	99.0856	99.2427
10%	94.8611	99.7668	99.2237	99.3422	99.6159
20%	94.75	99.8739	99.4858	99.6266	99.7749
30%	95.3616	99.9241	99.7047	99.7181	99.8757
40%	95.3396	99.9413	99.7944	99.7871	99.9144
50%	96.1134	99.9552	99.8269	99.8515	99.9318
60%	96.3241	99.9627	99.852	99.8541	99.9416
70%	96.5465	99.968	99.8673	99.8927	99.9586
80%	96.9312	99.9722	99.88	99.8868	99.9553
90%	97.1171	99.9763	99.9474	99.9077	99.9704
100%	97.0495	99.9786	99.9573	99.9118	99.9701
Statement Coverage %					
TSSp	Greedy	Additional Greedy	Genetic Algorithm	Simulated Annealing	Reactive GRASP
1%	96.747	98.1768	98.0792	98.1911	98.1596
2%	97.0323	99.039	98.8108	98.8664	99.0273
3%	96.937	99.3284	99.1366	99.1927	99.2257
5%	96.3181	99.5751	99.2731	99.4252	99.4398
10%	96.1091	99.782	99.452	99.6635	99.6428
20%	96.9909	99.8945	99.7965	99.8168	99.8693
30%	97.2931	99.9307	99.8703	99.8683	99.9112
40%	97.0724	99.9471	99.9003	99.8983	99.9358
50%	97.4288	99.9584	99.9214	99.9146	99.9445
60%	97.4015	99.9653	99.932	99.9281	99.9594
70%	97.6458	99.9707	99.9374	99.931	99.9653
80%	97.8832	99.9748	99.9399	99.9273	99.9722
90%	97.8907	99.9777	99.9496	99.9471	99.9653
100%	97.8901	99.9799	99.9627	99.9494	99.978

Table 5: Results of Coverage Criteria (Average of 10 Executions), Program Schedule2.

Block Coverage %					
TSSp	Greedy	Additional Greedy	Genetic Algorithm	Simulated Annealing	Reactive GRASP
1%	97.2708	98.1199	98.066	98.167	98.1064
2%	98.2538	99.0566	98.9605	99.0325	99.036
3%	98.6447	99.3764	99.3304	99.3464	99.3534
5%	99.4678	99.6184	99.5851	99.5879	99.6184
10%	98.2116	99.8527	99.2378	99.7869	99.7659
20%	99.9056	99.907	99.8952	99.893	99.907
30%	99.9385	99.9385	99.9348	99.9267	99.9385
40%	99.9538	99.9538	99.9476	99.9418	99.9538
50%	99.963	99.963	99.9586	99.9535	99.963
60%	99.9692	99.9692	99.9676	99.9612	99.9692
70%	99.9736	99.9736	99.9702	99.9584	99.9736
80%	99.9769	99.9769	99.972	99.9641	99.9769
90%	99.9794	99.9794	99.9779	99.9735	99.9794
100%	99.9815	99.9815	99.9796	99.9701	99.9815
Decision Coverage %					
TSSp	Greedy	Additional Greedy	Genetic Algorithm	Simulated Annealing	Reactive GRASP
1%	95.6563	98.3687	97.9922	98.1129	98.3301
2%	96.1375	98.9533	98.2113	98.5404	98.8501
3%	95.5965	99.3111	98.4344	98.9122	99.0936
5%	97.6887	99.6164	99.058	99.2189	99.4773
10%	97.1277	99.7985	99.4385	99.4873	99.7057
20%	97.2249	99.9027	99.7033	99.7575	99.8713
30%	97.2647	99.9352	99.8177	99.8224	99.9126
40%	97.2726	99.9513	99.8145	99.8673	99.9144
50%	97.2823	99.9712	99.8745	99.8907	99.9411
60%	97.2869	99.9676	99.8827	99.9143	99.9584
70%	97.2981	99.9722	99.915	99.9013	99.9595
80%	97.3005	99.9756	99.9311	99.915	99.9695
90%	99.9794	99.9794	99.9779	99.9304	99.9794
100%	99.9815	99.9815	99.9796	99.9297	99.9815
Statement Coverage %					
TSSp	Greedy	Additional Greedy	Genetic Algorithm	Simulated Annealing	Reactive GRASP
1%	97.7116	98.2883	98.1984	98.0316	98.2777
2%	97.4612	99.1097	98.9346	98.6235	99.0208
3%	97.1499	99.336	98.9259	99.0397	99.1481
5%	97.7227	99.6029	99.3066	99.428	99.5114
10%	98.3422	99.8072	99.6104	99.6295	99.734
20%	98.4317	99.9014	99.7765	99.7866	99.8815
30%	98.474	99.9363	99.8543	99.8455	99.9074
40%	98.4861	99.9525	99.8892	99.8833	99.9441
50%	98.4988	99.962	99.9159	99.9055	99.9568
60%	98.5041	99.9683	99.9251	99.9123	99.9626
70%	98.5109	99.9728	99.9345	99.9278	99.9663
80%	98.512	99.9762	99.9429	99.9291	99.9725
90%	98.5166	99.9788	99.9549	99.9463	99.9757
100%	98.521	99.981	99.9583	99.9453	99.9783

Table 6: Coverage Significance and Time Mean Difference, Small Programs.

Algorithm (x)	Algorithm (y)	Mean Coverage Difference (%) ($x - y$)	Coverage Difference Significance (t-test)	Time Mean Difference (s) ($x - y$)
Greedy Algorithm	Additional Greedy Algorithm	−1.9041	0.0000	−1.4908
	Genetic Algorithm	−1.8223	0.0000	−8.4361
	Simulated Annealing	−1.8250	0.0000	−0.0634
	Reactive GRASP	−1.8938	0.0000	−100.0312
Additional Greedy Algorithm	Greedy Algorithm	1.9041	0.0000	1.4908
	Genetic Algorithm	0.0818	0.0000	−6.9452
	Simulated Annealing	0.0790	0.0000	1.4274
	Reactive GRASP	0.0103	0.1876	−98.5403
Genetic Algorithm	Greedy Algorithm	1.8223	0.0000	8.4361
	Additional Greedy Algorithm	−0.0818	0.0000	6.9452
	Simulated Annealing	−0.0026	0.4918	8.3727
	Reactive GRASP	−0.0715	0.0000	−91.5951
Simulated Annealing	Greedy Algorithm	1.8250	0.0000	0.0634
	Additional Greedy Algorithm	−0.0790	0.0000	−1.4274
	Genetic Algorithm	0.0026	0.4918	−8.3727
	Reactive GRASP	−0.0688	0.0000	−99.9679
Reactive GRASP	Greedy Algorithm	1.8938	0.0000	100.0312
	Additional Greedy Algorithm	−0.0103	0.1876	98.5403
	Genetic Algorithm	0.0715	0.0000	91.5951
	Simulated Annealing	0.0688	0.0000	99.9679

Table 7: Weighted Average for the Metrics, Small Programs.

Coverage Criterion	Greedy Algorithm	Additional Greedy Algorithm	Genetic Algorithm	Simulated Annealing	Reactive GRASP
Block Coverage	98.2858	**99.9578**	99.8825	99.8863	99.9335
Decision Coverage	97.8119	99.9276	99.8406	99.8417	**99.9368**
Statement Coverage	98.0328	**99.9573**	99.8743	99.8706	99.9417

Table 8: Difference in Performance between the Best and Worst Criteria, Small Programs.

	Greedy Algorithm	Additional Greedy Algorithm	Genetic Algorithm	Simulated Annealing	Reactive GRASP
Difference in performance between the best and worst criteria	0.4739	0.0302	0.0419	0.0446	0.0082

Table 9: Average for Each Algorithm (All Metrics), Small Programs.

	Greedy Algorithm	Additional Greedy Algorithm	Genetic Algorithm	Simulated Annealing	Reactive GRASP
Final Average	98.0435	99.9476	99.8658	99.8662	99.9373

Table 10: Standard Deviation of the Effectiveness for the Four Algorithms, Small Programs.

	Greedy Algorithm	Additional Greedy Algorithm	Genetic Algorithm	Simulated Annealing	Reactive GRASP
Standard Deviation	0.002371	0.000172	0.000222	0.000226	0.000041

Table 11: Summary of Results, Small Programs.

Algorithm	Coverage Performance	Execution Time	Observations
Greedy Algorithm	The worst performance	Fast	
Additional Greedy Algorithm	Best performance of all	Fast	
Genetic Algorithm	Fourth best performance	Medium	It generated a better coverage only once.
Simulated Annealing	Third best performance	Fast	No significant difference to genetic algorithm.
Reactive GRASP	Second best performance	Slow	No significant difference to Additional Greedy Algorithm.

Table 12: Results of Coverage Criteria (1 Execution), Program Space.

Block Coverage %					
TSSp	Greedy	Additional Greedy	Genetic Algorithm	Simulated Annealing	Reactive GRASP
1%	87.4115	96.4804	92.6728	91.4603	95.6961
5%	85.8751	98.5599	94.8614	94.9912	98.0514
10%	85.5473	99.1579	95.9604	96.7242	98.6774
20%	86.5724	99.6063	98.0118	97.991	99.4235
30%	86.9639	99.7423	98.5998	98.6937	99.6431
40%	87.3629	99.811	98.9844	98.9004	99.7339
50%	87.8269	99.842	99.1271	99.216	99.7755
Decision Coverage %					
TSSp	Greedy	Additional Greedy	Genetic Algorithm	Simulated Annealing	Reactive GRASP
1%	88.753	96.9865	91.6811	92.0529	96.4502
5%	85.5131	98.553	93.6639	94.9256	97.8443
10%	86.9345	99.1999	95.9172	96.6152	98.358
20%	87.9909	99.6074	98.0217	97.7348	99.2446
30%	88.4008	99.7464	98.4662	98.5373	99.3256
40%	88.6799	99.8074	98.9283	98.8599	99.7149
50%	88.6635	99.8476	99.0786	98.84	99.7469
Statement Coverage %					
TSSp	Greedy	Additional Greedy	Genetic Algorithm	Simulated Annealing	Reactive GRASP
1%	92.8619	97.7642	94.3287	93.5957	97.0516
5%	90.9306	99.1171	95.7946	96.4218	98.4031
10%	91.3637	99.5086	97.5863	97.7154	99.3172
20%	91.7803	99.7598	98.6129	98.6336	99.6214
30%	92.1344	99.8473	99.0048	99.2151	99.6555
40%	92.1866	99.8859	99.3106	99.2963	99.8365
50%	92.2787	99.9117	99.4053	99.4852	99.8517

in the present paper demonstrate that this algorithm is also worse than the proposed Reactive GRASP approach. The simulated annealing algorithm had the third best performance, outperforming only the Greedy algorithm.

Figures 2, 3, and 4 demonstrate a comparison among the five algorithms used in the experiments. It is easy to see that the best performance was that of the Additional Greedy algorithm, followed by that of the Reactive GRASP algorithm. Reactive GRASP surpassed the genetic algorithm and simulated annealing in all coverage criteria, and it had the best performance at APDC criterion. The Additional Greedy algorithm was better at APBC and APSC criteria and Greedy algorithm was the worst of all.

For better visualization, consider Figures 5 and 6 that show these comparisons among the used algorithms. To make the result clearer, Figures 7 and 8 have this information regarding the 3 more efficient algorithms in this experiment. Figure 9 shows the final coverage average for each algorithm.

To investigate the statistical significance, we used t-test, which can be seen in Table 6. For each pair of algorithms, the mean coverage difference is given, and the significance level. If the significance is smaller than 0.05, the difference between the algorithms is statistically significant [6]. As can be seen, there is no significant difference between Reactive GRASP and Additional Greedy, in terms of coverage. In addition, one can see that there is no significant difference between simulated annealing and genetic algorithm, also in accordance with Table 6.

We can also notice in Table 6 the time mean difference for execution, for each pair of algorithms. It is important to mention that the time required to execute Reactive GRASP was about 61.53 larger than the time required to execution for Additional Greedy algorithm.

Another conclusion that can be drawn from the graphs is that the performance of the Reactive GRASP algorithm has remained similar for all metrics used, while Additional Greedy algorithm was a slightly different behavior for each metric.

Table 7 shows the weighted average of the algorithms, for each coverage criterion. The best results are highlighted in the table (bold). Table 8 shows the difference in performance between the best and the worst metric regarding the coverage percentage. In this experiment, Reactive GRASP had the minor difference in performance between the best and the worst coverage criterion, which demonstrates an interesting characteristic of this algorithm: its stability.

Table 9 contains the effectiveness average for all coverage criteria for each algorithm (APBC, APDC, and APSC).

Table 13: Coverage Significance and Time Mean Difference, Program Space.

Algorithm (x)	Algorithm (y)	Mean Coverage Difference (%) ($x - y$)	Coverage Difference Significance (t-test)	Time Mean Difference (s) ($x - y$)
Greedy Algorithm	Additional Greedy Algorithm	−10.5391	0.0000	−16.643
	Genetic Algorithm	−9.4036	0.0000	−495.608
	Simulated Annealing	−9.4459	0.0000	−5.339
	Reactive GRASP	−10.3639	0.0000	−36, 939.589
Additional Greedy Algorithm	Greedy Algorithm	10.5391	0.0000	16.643
	Genetic Algorithm	1.1354	0.0000	−478.965
	Simulated Annealing	1.0931	0.0000	11.303
	Reactive GRASP	0.1752	0.0613	−36, 922.945
Genetic Algorithm	Greedy Algorithm	9.4036	0.0000	495.608
	Additional Greedy Algorithm	−1.1354	0.0000	478.965
	Simulated Annealing	−0.0423	0.4418	490.268
	Reactive GRASP	−0.9602	0.0000	−36, 443.980
Simulated Annealing	Greedy Algorithm	9.4459	0.0000	5.339
	Additional Greedy Algorithm	−1.0931	0.0000	−11.303
	Genetic Algorithm	0.0423	0.4418	−490.268
	Reactive GRASP	−0.9180	0.0000	−3, 6934.249
Reactive GRASP	Greedy Algorithm	10.3639	0.0000	36, 939.589
	Additional Greedy Algorithm	−0.1752	0.0613	36,922.945
	Simulated Annealing	0.9180	0.0000	3,6934.249
	Genetic Algorithm	0.9602	0.0000	36,443.980

Table 14: Weighted Average for the Metrics, Program Space.

Coverage Criterion	Greedy Algorithm	Additional Greedy Algorithm	Genetic Algorithm	Simulated Annealing	Reactive GRASP
Block Coverage	87.1697	**99.6781**	98.4650	98.53273	99.5424
Decision Coverage	88.3197	**99.6856**	98.3631	98.33361	99.4221
Statement Coverage	92.0653	**99.8081**	98.9375	99.02625	99.6819

Together with Figure 9, Table 9 reinforces that the best performance was obtained by Additional Greedy algorithm, followed by that of the Reactive GRASP algorithm. Notice that Reactive GRASP algorithm has little difference in the performance compared with that of Additional Greedy algorithm.

The standard deviation shown in Table 10 refers to the 3 metrics (APBC, APDC, and APSC). It was calculated using the weighted average percentage of each algorithm. According to data in Table 10, the influence of the effectiveness performance regarding the coverage criterion is the lowest in the proposed Reactive GRASP algorithm, since its standard deviation value is the minimum among the algorithms. These data mean that the proposed technique is the one that less varies its performance related to the coverage criteria, which, again, demonstrates its higher stability.

4.3.2. Analysis for the Space Program. The results for space program were similar to results for the four small programs. The Reactive GRASP algorithm had the second best performance. Additional Greedy algorithm, genetic algorithm, simulated annealing, and Reactive GRASP algorithms significantly outperformed the Greedy algorithm. Comparing both metaheuristic-based approaches, the better performance obtained by the Reactive GRASP algorithm over the genetic algorithm and simulated annealing is clear.

The Reactive GRASP algorithm was followed by genetic algorithm approach, which performed the fourth best in our evaluation. The third best evaluation was obtained by simulated annealing.

Figures 10, 11, and 12 demonstrate a comparison between the five algorithms used in the experiments, for the space program. Based on these figures, it is possible to conclude that the best performance was that of the Additional Greedy algorithm, followed by the Reactive GRASP algorithm. Reactive GRASP surpassed the genetic algorithm, simulated annealing, and Greedy algorithm. One difference between the results for space program and the small programs is that Additional Greedy algorithm was better for all criteria, while, for small programs, Reactive GRASP had the best results for the APDC criteria. Another difference is the required execution time. As the size of the

Table 15: Difference in Performance between the Best and the Worst Criteria, Program Space.

	Greedy Algorithm	Additional Greedy Algorithm	Genetic Algorithm	Simulated Annealing	Reactive GRASP
Difference in performance between the best and worst criteria	4.8956	0.1300	0.5744	0.6926	0.2598

Table 16: Average for Each Algorithm (All Metrics), Program Space.

	Greedy Algorithm	Additional Greedy Algorithm	Genetic Algorithm	Simulated Annealing	Reactive GRASP
Final Average	89.1849	99.7240	98.5885	98.6308	99.5488

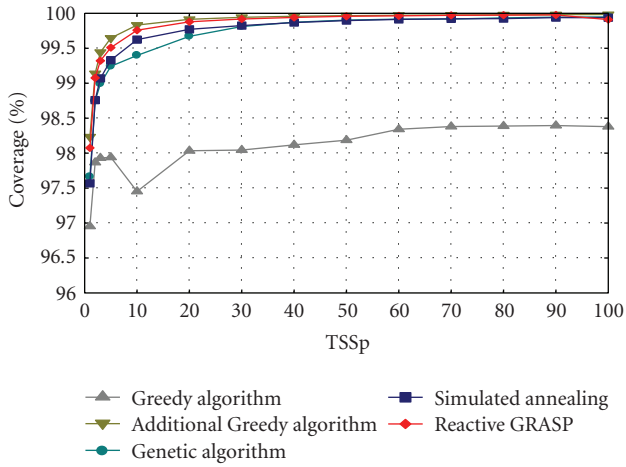

Figure 2: APBC (Average Percentage Block Coverage), Comparison among Algorithms for Small Programs.

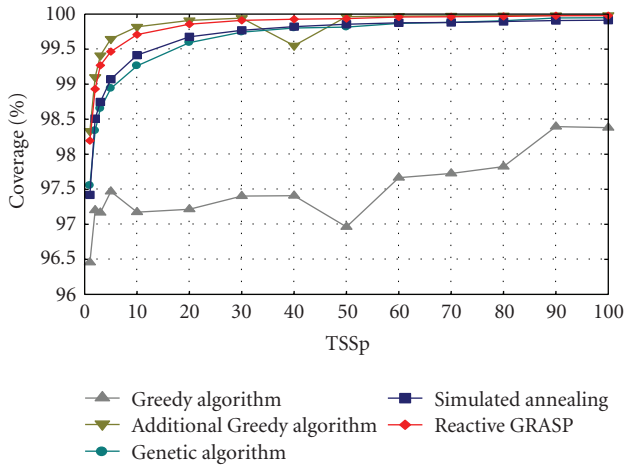

Figure 3: APDC (Average Percentage Decision Coverage), Comparison among Algorithms for Small Programs.

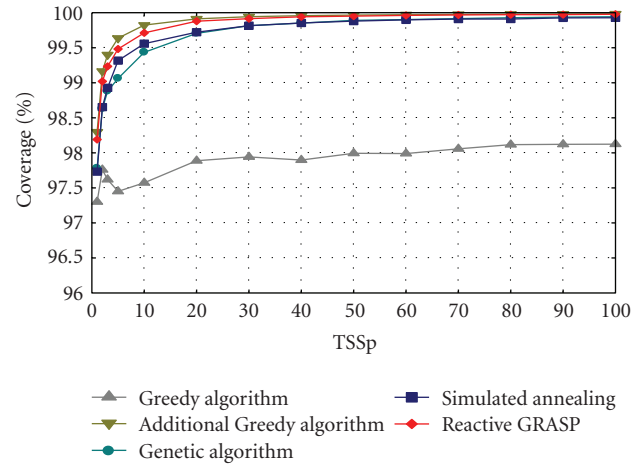

Figure 4: APSC (Average Percentage Statement Coverage), Comparison among Algorithms for Small Programs.

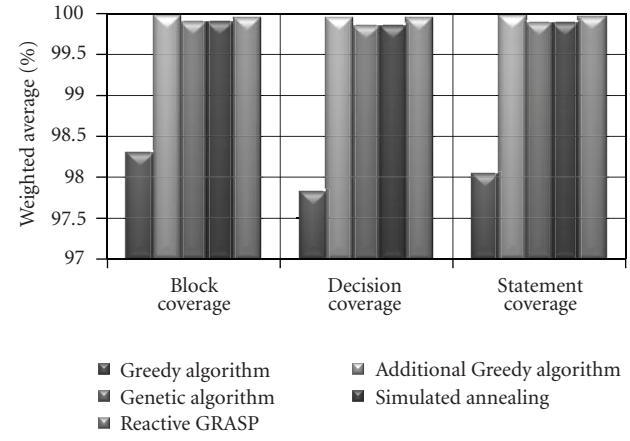

Figure 5: Weighted Average for the Metrics (Comparison among the Metrics), Small Programs.

program increases, the Reactive GRASP algorithm has its time relatively less slow compared with the others.

For better visualization, consider Figures 13 and 14 that show these comparisons among the used algorithms. To make the result clearer, Figures 15 and 16 have this information regarding the 3 more efficient algorithms in this experiment. Figure 17 shows the coverage average for each algorithm.

The t-test was used to investigate the statistical significance for space program, which can be seen in Table 13. As in the analysis for the small programs, the level of significance of the result was set to 0.05. In the same way to the small

Table 17: Standard Deviation of the Effectiveness for the Four Algorithms, Program Space.

	Greedy Algorithm	Additional Greedy Algorithm	Genetic Algorithm	Simulated Annealing	Reactive GRASP
Standard Deviation	0.025599	0.000730	0.003065	0.003566	0.001300

Table 18: Summary of Results, Program Space.

Algorithm	Coverage Performance	Execution Time	Observations
Greedy Algorithm	The worst performance.	Fast	
Additional Greedy Algorithm	Best performance of all.	Fast	
Genetic Algorithm	Fourth best performance.	Medium	
Simulated Annealing	Third best performance.	Fast	No significant difference to genetic algorithm.
Reactive GRASP	Second best performance	Slow	No significant difference to Additional Greedy Algorithm.

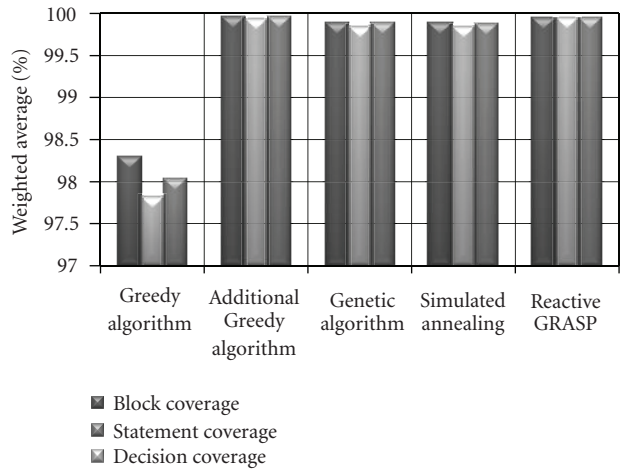

Figure 6: Weighted Average for the Metrics (Comparison among the Algorithms), Small Programs.

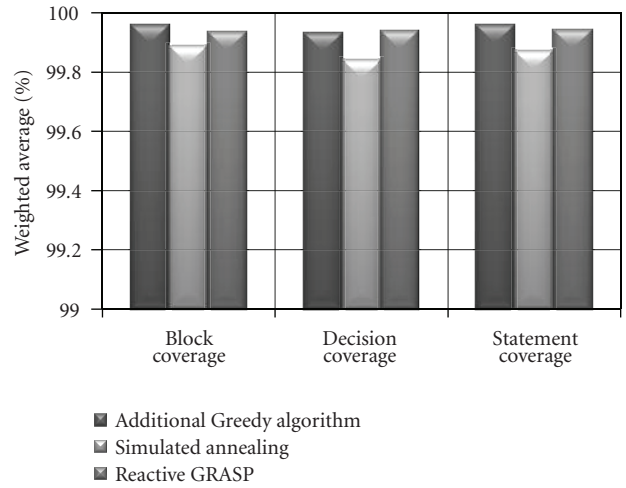

Figure 7: Weighted Average for the 3 More Efficient Algorithms (Comparison among the Metrics), Small Programs.

programs, there is no significant difference between Reactive GRASP and Additional Greedy, in terms of coverage, for space program, neither for simulated annealing nor for genetic algorithm.

4.3.3. Final Analysis. 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. [6]. 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 have tried from 1% to 100% of test cases for each program and criterion for the four small programs, and the performances of all algorithms remained unaltered. This demonstrated that the ability of the five algorithms discussed here is not deeply related to the number of test cases required to order.

In terms of time, 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 Tables 6 and 13 that the Greedy algorithm performed more efficiently than all other algorithms. This algorithm was, on average, 1.491 seconds faster than Additional Greedy algorithm, 8.436 faster than the genetic algorithm, 0.057 faster than the simulated annealing, and almost 50 seconds faster than the Reactive GRASP approach, for the small programs. In terms of relative values, Reactive GRASP was 61.53 times slower than Additional Greedy, 11.68 slower than genetic algorithm, 513.87 slower than simulated annealing, and 730.92 slower than Greedy algorithm. This result demonstrates, once again, the great performance obtained by the Additional Greedy algorithm compared to that of the Greedy algorithm, 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 Additional Greedy algorithm and 41,511 seconds slower than the genetic algorithm. In favor of both metaheuristic-based approaches is the fact that one may calibrate the time

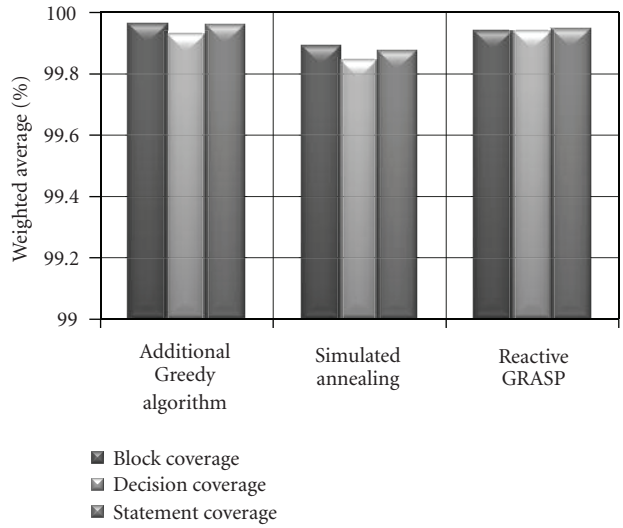

Figure 8: Weighted Average for the 3 More Efficient Algorithms (Comparison among the Algorithms), Small Programs.

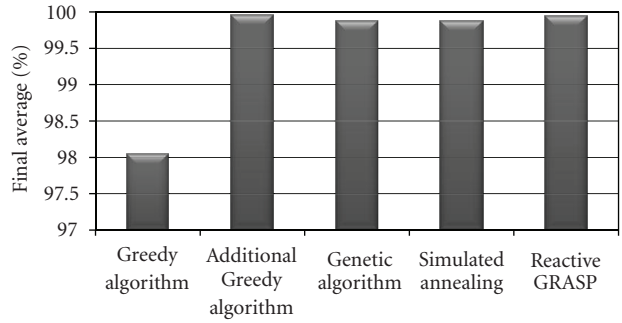

Figure 9: Average for Each Algorithm (All Metrics), Small Programs.

required for prioritization depending on time constraints and characteristics of programs and test cases. This flexibility is not present in the Greedy algorithms.

Tables 11 and 18 summarize the results described above.

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 literature. Since the Reactive GRASP algorithm performed significantly better—in terms of coverage performance—than the genetic algorithm, Simulated Annealing, and similarly to the Greedy algorithm and it avoids the problems mentioned by Rothermel [2] and Li et al. [6], where Greedy algorithms may fail to choose the optimal test case ordering, the use of the Reactive GRASP algorithm is

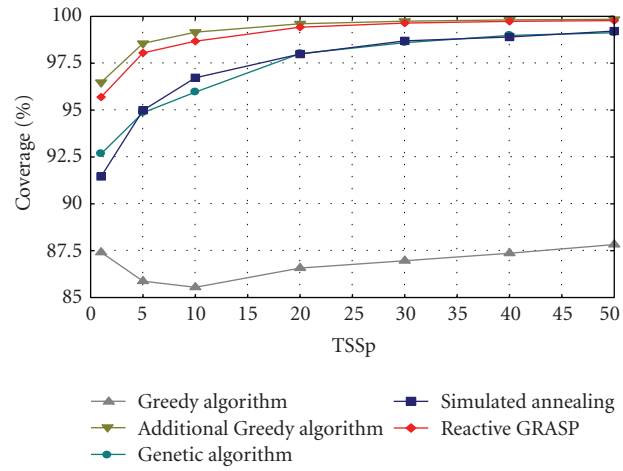

Figure 10: APBC (Average Percentage Block Coverage), Comparison among Algorithms for Program Space.

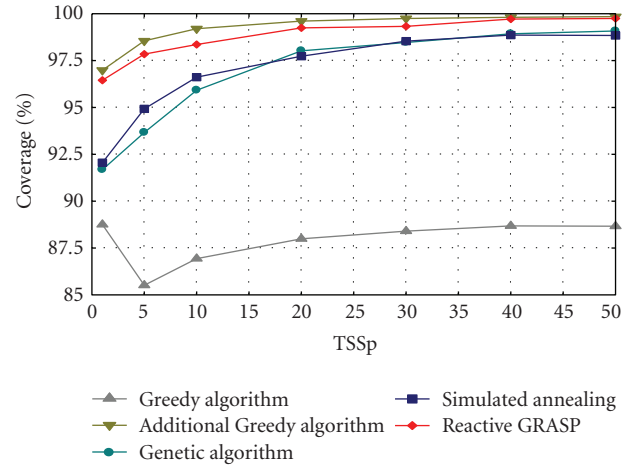

Figure 11: APDC (Average Percentage Decision Coverage), Comparison among Algorithms for Program Space.

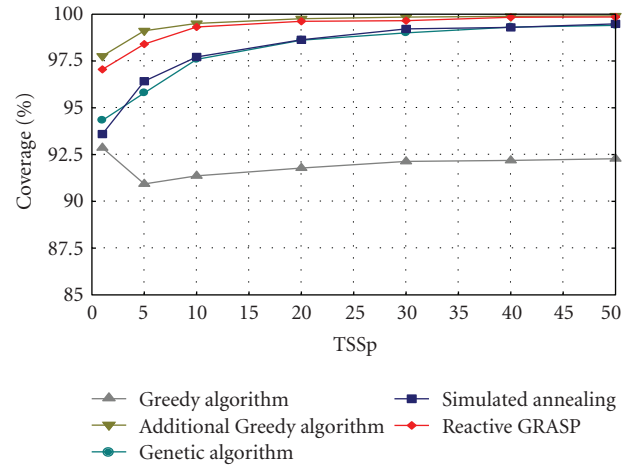

Figure 12: APSC (Average Percentage Statement Coverage), Comparison among Algorithms for Program Space.

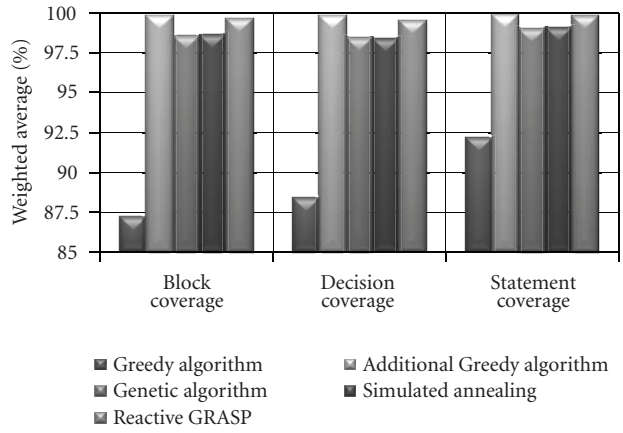

Figure 13: Weighted Average for the Metrics (Comparison among the Metrics), Program Space.

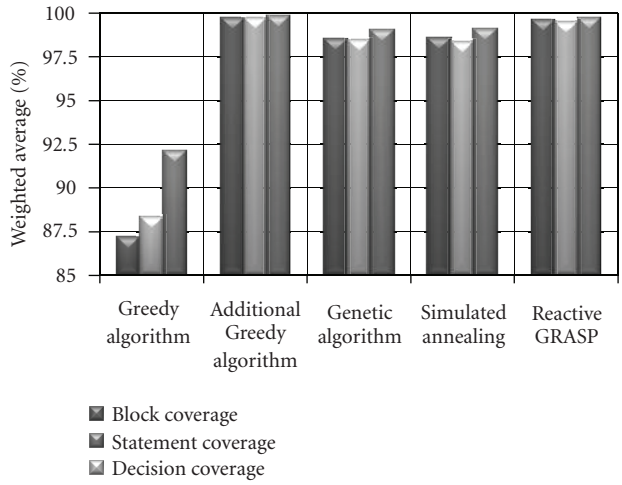

Figure 14: Weighted Average for the Metrics (Comparison among the Algorithms), Program Space.

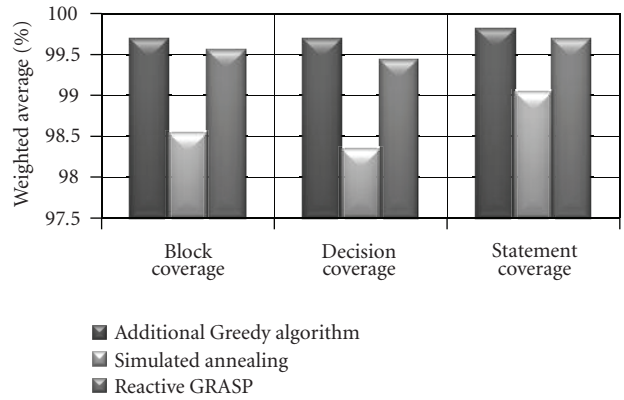

Figure 15: Weighted Average for the 3 More Efficient Algorithms (Comparison among the Metrics), Program Space.

indicated to the problem of test case prioritization, especially when time constraints are not too critical, since the Reactive GRASP added a considerable overhead.

Our experimental results confirmed also the previous results reported in literature regarding the good performance of the Additional Greedy algorithm. However, some results

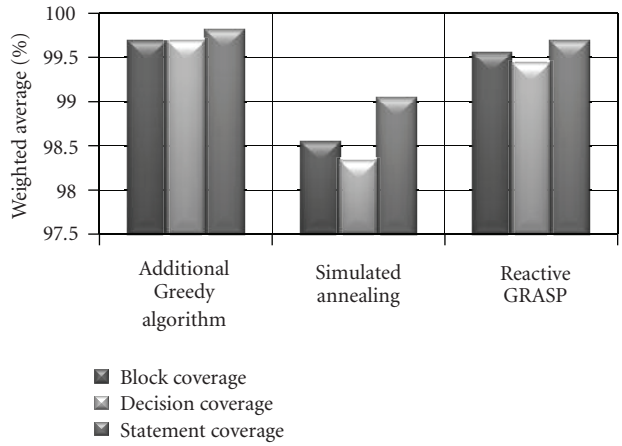

Figure 16: Weighted Average for the 3 More Efficient Algorithms (Comparison among the Algorithms, Program Space.

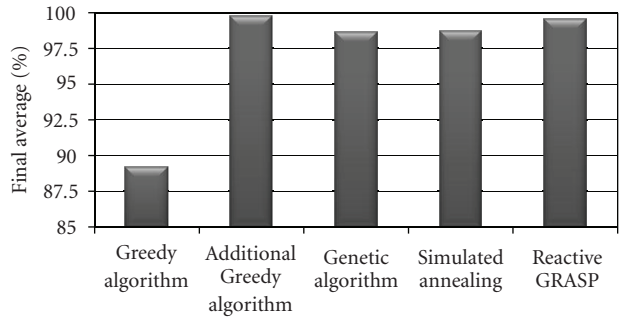

Figure 17: Final Average for Each Algorithm, Program Space.

point out to some interesting characteristics of the Reactive GRASP solution. First, the coverage performance was not significantly worse when compared to that of the Additional Greedy algorithm. In addition, the proposed solution had a more stable behavior when compared to all other solutions. Next, GRASP can be set to work with as many or as little time as available.

As future work, 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 and VNS, among others.

References

[1] M. Fewster and D. Graham, *Software Test Automation*, Addison-Wesley, Reading, Mass, USA, 1st edition, 1994.

[2] G. Rothermel, R. H. Untcn, 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.

[3] G. J. Myers, *The Art of Software Testing*, John Wiley & Sons, New York, NY, USA, 2nd edition, 2004.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, MIT Press, Cambridge, Mass, USA; McGraw-Hill, New York, NY, USA, 2nd edition, 2001.

[5] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Test case prioritization: an empirical study," in *Proceedings of the International Conference on Software Maintenance (ICSM '99)*, pp. 179–188, Oxford, UK, September 1999.

[6] 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.

[7] F. Glover and G. Kochenberger, *Handbook of Metaheuristics*, Springer, Berlin, Germany, 1st edition, 2003.

[8] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos, "Time-aware test suite prioritization," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '06)*, pp. 1–12, Portland, Me, USA, July 2006.

[9] S. Yoo and M. Harman, "Pareto efficient multi-objective test case selection," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '07)*, pp. 140–150, London, UK, July 2007.

[10] S. Elbaum, G. Rothermel, S. Kanduri, and A. G. Malishevsky, "Selecting a cost-effective test case prioritization technique," *Software Quality Journal*, vol. 12, no. 3, pp. 185–210, 2004.

[11] K. Deb, S. Agrawal, A. Pratab, and T. Meyarivan, "A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II," in *Proceedings of the 6th Parallel Problem Solving from Nature Conference (PPSN '00)*, pp. 849–858, Paris, France, September 2000.

[12] J. H. Holland, *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*, University of Michigan, Ann Arbor, Mich, USA, 1975.

[13] M. Harman, "The current state and future of search based software engineering," in *Proceedings of the International Conference on Software Engineering—Future of Software Engineering (FoSE '07)*, pp. 342–357, Minneapolis, Minn, USA, May 2007.

[14] 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.

[15] 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.

[16] M. Paris and C. C. Ribeiro, "Reactive GRASP: an application to a matrix decomposition problem in TDMA traffic assignment," *INFORMS Journal on Computing*, vol. 12, no. 3, pp. 164–176, 2000.

[17] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria," in *Proceedings of the 16th International Conference on Software Engineering (ICSE '99)*, pp. 191–200, Los Angeles, Calif, USA, 1999.

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