Search Algorithms for Regression Test Case Prioritization

Zheng Li, Mark Harman, and Robert M. Hierons

Abstract—Regression testing is an expensive, but important, process. Unfortunately, there may be insufficient resources to allow for the reexecution of all test cases during regression testing. In this situation, test case prioritization techniques aim to improve the effectiveness of regression testing by ordering the test cases so that the most beneficial are executed first. Previous work on regression test case prioritization has focused on Greedy Algorithms. However, it is known that these algorithms may produce suboptimal results because they may construct results that denote only local minima within the search space. By contrast, metaheuristic and evolutionary search algorithms aim to avoid such problems. This paper presents results from an empirical study of the application of several greedy, metaheuristic, and evolutionary search algorithms to six programs, ranging from 374 to 11,148 lines of code for three choices of fitness metric. The paper addresses the problems of choice of fitness metric, characterization of landscape modality, and determination of the most suitable search technique to apply. The empirical results replicate previous results concerning Greedy Algorithms. They shed light on the nature of the regression testing search space, indicating that it is multimodal. The results also show that Genetic Algorithms perform well, although Greedy approaches are surprisingly effective, given the multimodal nature of the landscape.

Index Terms—Search techniques, test case prioritization, regression testing.

1 Introduction

Regression testing is a frequently applied but expensive maintenance process that aims to (re)verify modified software. Many approaches for improving the regression testing processes have been investigated. Test case prioritization [17], [18], [22] is one of these approaches, which orders test cases so that the test cases with highest priority, according to some criterion (a "fitness metric"), are executed first.

Rothermel et al. [18] define the test case prioritization problem and describe several issues relevant to its solution. The test case prioritization problem is defined (by Rothermel et al.) as follows:

The Test Case Prioritization Problem. Given: *T, a test suite; PT, the set of permutations of T; f, a function from PT to the real numbers.*

Problem: *Find $T' \in PT$ such that*

$$(\forall\, T'' \; (T'' \in PT) \; (T'' \neq T') \; [f(T') \geq (T'')].$$

Here, PT represents the set of all possible prioritizations (orderings) of T and f is a function that, applied to any such ordering, yields an award value for that ordering.

Test case prioritization can address a wide variety of objectives [18]. For example, concerning coverage alone, testers might wish to schedule test cases in order to achieve code coverage at the fastest rate possible in the initial phase of regression testing to reach 100 percent coverage soonest or to ensure that the maximum possible coverage is achieved by some predetermined cut-off point. Of course, the ideal order would reveal faults soonest, but this cannot be determined in advance, so coverage often has to serve as the most readily available surrogate. In the Microsoft Developer Network (MSDN) library, the achievement of adequate coverage without wasting time is a primary consideration when conducting regression tests [13]. Furthermore, several testing standards require branch adequate coverage, making the speedy achievement of coverage an important aspect of the regression testing process.

In previous work, many techniques for regression test case prioritization have been described. Most of the proposed techniques were code-based, relying on information relating test cases to coverage of code elements. In [6], [17], [18], Rothermel et al. investigated several prioritizing techniques, such as total statement (or branch) coverage prioritization and additional statement (or branch) coverage prioritization, that can improve the rate of fault detection. In [22], Wong et al. prioritized test cases according to the criterion of "increasing cost per additional coverage." In [20], Srivastava and Thiagarajan studied a prioritization technique that was based on the changes that have been made to the program and focused on the objective function of "impacted block coverage." Other noncoverage based techniques in the literature include fault-exposing-potential (FEP) prioritization [18], history-based test prioritization [11], and the incorporation of varying test costs and fault severities into test case prioritization [5], [6].

Greedy Algorithms have been widely employed for test case prioritization. Greedy Algorithms incrementally add test cases to an initially empty sequence. The choice of

- *Z. Li and M. Harman are with the Software Engineering Group, Department of Computer Science, King's College London, Strand, London, UK, WC2R 2LS. E-mail: {zheng.li, mark.harman}@kcl.ac.uk.*
- *R.M. Hierons is with the School of Information Systems, Computing, and Mathematics, Brunel University, Uxbridge, Middlesex, UK, UB8 3PH. E-mail: rob.hierons@brune.ac.uk.*

Manuscript received 2 Mar. 2005; revised 12 Apr. 2006; accepted 21 Nov. 2006; published online 1 Mar. 2007.
Recommended for acceptance by H. Muller.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0048-0305.

TABLE 1
A Case in Which the Greedy Algorithm Will Not Produce an Optimal Solution

Test Case	Statement							
	1	2	3	4	5	6	7	8
A	X	X	X			X	X	X
B	X	X	X				X	X
C	X	X	X	X				
D					X	X	X	X

which test case to add next is simple: it is that which achieves the maximum value for the desired metric (e.g., some measure of coverage). However, as Rothermel et al. [18] point out, this greedy prioritization algorithm may not always choose the optimal test case ordering. Greedy Algorithms also require that it is possible to define the improvement in fitness obtained by the addition of a single element to a partially constructed sequence. As this paper will show, in the case of regression test case prioritization, this is not always possible.

A simple example, based on statement coverage, is presented in Table 1. If the aim is to achieve full statement coverage as soon as possible, a Greedy Algorithm may select A, B, C, D (or A, C, D, B, depending upon the greedy strategy). However, the optimal test case orderings for this example are C, D, A, B and D, C, A, B.

Metaheuristic search techniques [15] are high-level frameworks that utilize the automated discovery of heuristics in order to find solutions to combinatorial problems at a reasonable computational cost. Evolutionary algorithms, of which Genetic Algorithms are a subclass, are a form of metaheuristic search that employs a Darwinian Evolutionary metaphor to guide the search by a process of "survival of the fittest." In the case of scheduling problems, of which regression test case prioritization is an example, the application of Genetic Algorithms has been shown to be effective [9]. As such, an empirical study of the effectiveness of these and related metaheuristic techniques is timely. As a by-product of such a study, it is possible to gain an insight into the nature of the search space denoted by test case prioritization and to study the fitness metrics used to guide the search.

This paper focuses on test case prioritization techniques for code coverage, including block coverage, decision (branch) coverage, and statement coverage, which are widely studied in prior work. With different objective functions, techniques will have different complexity and search-space characteristics. Given a function f that assesses the rate of achievement of code coverage, an efficient solution to the test case prioritization problem would provide an efficient solution to the knapsack problem, which is known to be NP-hard [7]. Thus, prioritization techniques for code coverage are necessarily heuristic [18].

In this paper, five search techniques are studied: two metaheuristic search techniques (Hill Climbing and Genetic Algorithms), together with three greedy algorithms (Greedy, Additional Greedy, and 2-Optimal Greedy). Of these five algorithms, only Greedy and Additional Greedy have been studied previously. The paper presents results from an empirical study that compared the performance of the five search algorithms applied to six programs, ranging from 374 to 11,148 lines of code.

The primary findings of the study are as follows:

1. The Additional Greedy and 2-Optimal Algorithms appear to be the best approaches overall. However, the difference between the performances of these and the Genetic Algorithm was not statistically significant for any of the small programs.
2. The results of the empirical study show that the Additional Greedy, 2-Optimal, and Genetic Algorithms always outperform the Greedy Algorithm. These results for the programs studied are shown to be statistically significant for all programs studied.
3. The results for the Hill Climbing algorithm indicate that the fitness landscapes of the search spaces involved are multimodal.
4. This paper also reveals some interesting properties of the fitness metrics.

The rest of this paper is organized as follows: Section 2 describes the five algorithms studied in the paper, while Section 3 presents the details and results of the empirical study of the application of these algorithms to the set of six subject programs. Section 4 presents some related work and Section 5 concludes.

2 Algorithms

This section describes the five algorithms used in the experiments.

2.1 Greedy Algorithm

A Greedy Algorithm is an implementation of the "next best" search philosophy. It works on the principle that the element with the maximum weight is taken first, followed by the element with the second-highest weight, and so on, until a complete, but possibly suboptimal, solution has been constructed. Greedy search seeks to minimize the estimated cost to reach some goal. It is simple, but in situations where its results are of high quality, it is attractive because it is typically inexpensive both in implementation and execution time.

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

For the example in Table 1, test case A is selected first since it covers six statements, the maximum covered by a single test case. Test case B, which covers five statements, is selected next. Test cases C and D cover the same number of statements and so the Greedy Algorithm could return either A, B, C, D or A, B, D, C, depending upon the order in which test cases are considered.

2.2 Additional Greedy Algorithm

The "Additional" Greedy Algorithm is also a kind of Greedy Algorithm, but with a different strategy. It combines feedback from previous selections. It iteratively selects the maximum weight element of the problem from that part that has not already been consumed by previously selected elements.

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

Note that, after 100 percent coverage has been achieved, there are possible remaining unprioritized test cases that cannot add additional coverage. These remaining test cases could be ordered using any algorithm; in this work, the remaining test cases were ordered by reapplying the same algorithm—Additional Greedy Algorithm. However, after 100 percent coverage has been achieved, no further fitness improvement will be possible.

For the example in Table 1, test case A is selected first (it covers six statements) leaving statements 4 and 5 uncovered. Test case B is skipped since it covers neither statement 4 nor statement 5. Test cases C and D cover statements 4 and 5, respectively, so each covers only one uncovered statement. Thus, Additional Greedy would return either A, C, D, B or A, D, C, B.

2.3 2-Optimal Algorithm

The 2-Optimal (Greedy) Algorithm is an instantiation of the K-Optimal Greedy Approach [12] when $K = 2$. The K-Optimal approach selects the next K elements that, taken together, consume the largest part of the problem. In the case of K-Optimal *Additional* Greedy, it is the largest *remaining* part of the problem that is selected. In this research, a 2-Optimal Additional Greedy Algorithm was used, hereinafter referred to as the "2-Optimal" Algorithm for brevity.

The K-Optimal approach has been studied in the area of heuristic search to solve the Traveling Salesman Problem (TSP) that is defined as *"find the cycle of minimum cost that visits each of the vertices of a weighted graph G at least once"* [16]. Extensive experiments suggest that 3-optimal tours are usually within a few percent of the cost of optimal tours for TSP. As shown by Skiena, for $K > 3$, the computation time increases considerably faster than solution quality. The 2-Optimal approach has been found to be fast and effective [19].

Again, consider statement coverage: The 2-Optimal Algorithm updates coverage information for each unselected test case following the choice of each pair of test cases. Given a program containing m statements and a test suite containing n test cases, selecting a pair of test cases and readjusting coverage information has cost $O(m\,n^2)$ and this selection and readjustment must be performed $O(n)$ times. Therefore, the time complexity of the 2-Optimal Algorithms is $O(m\,n^3)$.

As with the other approaches, after complete coverage has been achieved, the 2-Optimal Algorithms were reapplied to the remaining test cases.

For the example in Table 1, test case C and D are chosen first since the combination of test cases C and D covers eight statements and this is the maximum among all pairs of test cases. The 2-Optimal Algorithm could thus return C, D, A, B, the global optimum for this simple example.

2.4 Hill Climbing

Hill Climbing is a well-known local search algorithm. There are two primary variations of this strategy: *steepest ascent* and *next best ascent*. In our empirical study, *steepest ascent* is adopted.

The *steepest ascent* Hill Climbing algorithm for test case prioritization is composed of the following steps:

1. Pick a random solution state and make this the current (i.e., initial) state.
2. Evaluate all the neighbors of the current state.
3. Move to the state with the largest increase in fitness from the current state. If no neighbor has a larger fitness than the current state, then no move is made.
4. Repeat the previous two steps until there is no change in the current state.
5. Return the current state as the solution state.

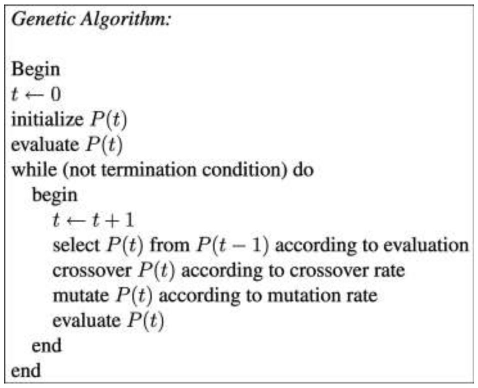
In our experiments, the state above is simply an ordering of a test suite; the neighborhood is defined as any new ordering of a test suite that can be obtained by exchanging the position of the first test case and any other test case. This definition of the neighborhood means that any test suite has $n - 1$ neighbors that must be evaluated for each iteration of the Hill Climbing algorithm. Of course, there are many valid choices of near neighbor. Our approach is just one choice and there is nothing special about the first test case. An alternative approach is to say that any order that can be obtained through swapping two elements is a neighbor. However, this has the disadvantage of dividing $O(n^2)$ neighbors and so is unlikely to scale.

Hill Climbing is simple and cheap to implement and execute. However, it is easy for the search to yield suboptimal results that are merely locally optimal, but not globally optimal.

2.5 Genetic Algorithms

Genetic Algorithms (GAs) represent a class of adaptive search techniques based on the processes of natural genetic selection according to Darwinian theory of biological evolution [8]. Fig. 1 shows a typical GA procedure, which includes initializing a population P, evaluating individuals, selecting pairs of individuals that are combined and mutated to generate new individuals, and forming the next generation. The search proceeds through a number of generations until the termination condition has been met.

The initial population is a set of randomly generated individuals. Each individual is represented by a sequence of variables/parameters (called genes), known as the chromosome. The chromosome encodes a possible solution to a given problem. The encoding can take many forms, for example, binary, real-valued, or character-based. A biased selection depending on the fitness value decides which individuals are to be used as the "parents" for producing the next generation. Crossover is a genetic operator that



```
Genetic Algorithm:

Begin
$t \leftarrow 0$
initialize $P(t)$
evaluate $P(t)$
while (not termination condition) do
  begin
    $t \leftarrow t+1$
    select $P(t)$ from $P(t-1)$ according to evaluation
    crossover $P(t)$ according to crossover rate
    mutate $P(t)$ according to mutation rate
    evaluate $P(t)$
  end
end
```

Fig. 1. Genetic Algorithm structure.

combines two individuals (the parents) to produce a new individual (the offspring). A probability of crossover determines whether crossover should be performed. The mutation operator alters one or more gene values in the individual, depending on the probability of mutation. A GA is not guaranteed to converge upon a single solution: The termination condition is often specified as a maximal number of generations, or as a given value of the fitness function that is deemed to be sufficient.

The detailed settings for the parameters and the operators of the GA in the experiments reported in this paper are described as follows:

Encoding. Binary encoding is not suitable for the sequencing problem, so an ordering chromosome was used. For a test suite with N test cases, the chromosome is encoded as an N-sized array, and the value of a chromosome element indicates the position of a test case in the order.

Selection. Rank-based fitness assignment was used, i.e., the fitness assigned to each individual depends only on its rank position in the current population and not on the concrete value of the overall fitness, as would be the case for proportional fitness assignment. Baker's linear ranking algorithm [2] is employed. The fitness of each individual in the population is defined as

$$Fitness(Pos) = 2 - SP + 2 \cdot (SP - 1) \cdot \frac{(Pos - 1)}{(Nind - 1)},$$

where $Fitness(Pos)$ is the fitness assigned to the individual, Pos is the position of the individual in the population, $Nind$ is the population size, and SP is selective pressure—the ratio of the best individual's selection probability to the average selection probability of all individuals. In tuning experimentation, a value of 2 for SP was found to produce the best results, so Baker's linear ranking algorithm simplifies to the following:

$$Fitness(Pos) = 2 \cdot \frac{(Pos - 1)}{(Nind - 1)}.$$

Crossover (Recombination). Two offspring (o_1 and o_2) are formed from two parents (p_1 and p_2), following the ordering chromosome crossover style adopted by Antoniol et al. [1]:

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

Mutation. The mutation operator randomly selects two genes and exchanges their position in the sequence.

It is common with the application of Genetic Algorithms to find that some "tuning" of the parameters of the algorithm is required to determine the values that yield the best results. In the case of the study reported here, some initial experiments were performed to optimize for the size of programs studied. The size of the population determines the diversity of the initial population. Insufficient diversity can lead to premature convergence to a suboptimal solution. Larger problems denote larger search spaces and therefore require a larger population in order to maintain diversity. The population size was set at 50 individuals for small programs (those with fewer than 1,000 lines of code), and 100 for the large programs (those with 1,000 or more lines of code). The algorithm was terminated after 100 generations for the small programs and 300 for the large programs. The other operators were set to identical values for all program sizes. Stochastic universal sampling was used in selection and mutation, the crossover probability (per individual) was set to 0.8, and the mutation probability (per individual) was set to 0.1.

3 Empirical Study

This section describes the family of empirical studies performed, including the design, subjects, measurements, and results.

3.1 Research Questions

The following two research questions motivated this study:

Q1. *Which algorithm is most effective in solving the test case prioritization problem for regression testing?*

Q2. *What factors could affect the efficiency of algorithms for the test case prioritization problem for regression testing?*

These two questions concern the quality and computational cost of regression testing and so are the two uppermost considerations in deciding upon a particular choice of prioritization algorithm.

3.2 Experimental Design

In order to improve the generality of the results reported in this paper, the basic search experiment was instantiated with several values of the three primary parameters that govern the nature and outcomes of the search, namely:

1. The program to which regression testing is applied. Six programs were studied, ranging from 374 to 11,148 lines of code and including real as well as "laboratory" programs.
2. The coverage criterion to be optimized. The three choices studied were block, decision, and statement coverage for each program.

TABLE 2
Experimental Subjects

subject	LoC	Blocks	Decisions	Test pool size	Average small test suite size	Average large test suite size
Print_tokens	726	126	123	4,130	16	318
Print_tokens2	570	103	154	4,115	12	388
Schedule	412	46	56	2,650	19	228
Schedule2	374	53	74	2,710	8	230
Space	9,564	869	1,068	13,585	155	4,350
Sed	11,148	951	2,331	1,293		1,293

3. The size of the test suite. Two granularities of test suite size were used: small suites of size 8-155 and large suites of size 228-4,350.

Of course, there is a connection between the first and third category since a larger program will typically require more test cases.

3.3 Subjects

In our study, we used two groups of programs, called "small programs" and "large programs,"[1] from a total of six C programs and their corresponding test suites (see Table 2). The programs and test suites were from an infrastructure [4] that is designed to support controlled experimentation with software testing and regression testing techniques.

`Print_tokens2` and `Schedule2` are variations of the programs `Print_tokens` and `Schedule`, respectively. These four programs were assembled by researchers at Siemens Corporate Research for experiments with control-flow and data-flow test adequacy criteria [10]; `Space` and `Sed` are large programs. `Space` was developed for the European Space Agency, and `Sed` is the `Unix` stream editor string processing utility.

For the first four "small" programs, the researchers at Siemens created a test pool containing possible test cases for each program [10]; Rothermel et al. constructed test pools for the two "large" programs following the same approach.

The infrastructure provided by Rothermel et al. [4] was used to obtain test suites in the following manner: In order to produce small test suites, a test case is selected at random from the test pool and is added to the suite only if it adds to the cumulative branch coverage. This is repeated until the test suite achieves full branch coverage. In order to produce large test suites, test cases are randomly selected from the test pool and added to the test suite until full branch coverage is achieved. Both small test suites and large test suites were applied in all three coverage criteria in our studies.

3.4 Effectiveness Measure

The fitness metrics studied are based upon APFD (Average of the Percentage of Faults Detected) [18], which measures the weighted average of the percentage of faults detected over the life of the suite. However, it is not (normally) possible to know the faults exposed by a test case in advance and so this value cannot be estimated before testing has taken place. Therefore, coverage is used as a surrogate measure. It is not the purpose of this paper to enter into the discussion of whether or not coverage is a suitable surrogate for faults found. Coverage is also an important concern in its own right, due to the way in which it has become a part of "coverage mandates" in various testing standards (for example, avionics standards [14]). The presence of these mandates means that testers must achieve the mandated levels of coverage irrespective of whether or not they may believe in coverage *per se*.

1. Although the "large" programs may not be regarded as large for some applications, in terms of the search space they denote, they are very large indeed and so they represent nontrivial applications of these search techniques.

TABLE 3
Test Suites and Block Coverage

Test Case	Block									
	1	2	3	4	5	6	7	8	9	10
A	X		X							
B	X	X	X		X					
C	X	X	X	X	X	X				
D		X		X						
E							X	X	X	X

Depending on the coverage criterion considered, three metrics were used in the paper:

1. APBC (Average Percentage Block Coverage). This measures the rate at which a prioritized test suite covers the blocks.
2. APDC (Average Percentage Decision Coverage). This measures the rate at which a prioritized test suite covers the decisions (branches).
3. APSC (Average Percentage Statement Coverage). This measures the rate at which a prioritized test suite covers the statements.

Consider the APBC metric as an example and 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 order T' is given by the equation

$$APBC = 1 - \frac{TB_1 + TB_2 + \ldots + TB_m}{nm} + \frac{1}{2n}.$$

APBC measures the weighted average of the percentage of block coverage over the life of the suite. APBC values range from 0 to 100; higher numbers imply faster (better) coverage rates.

To illustrate this measure, consider an example program with 10 blocks and a test suite of five test cases, A through E, each with block coverage characteristic as shown in Table 3.

Consider two orders of these test cases, order $T1$: A-B-C-D-E, and order $T2$: C-E-B-A-D. Fig. 2a and Fig. 2b show the percentages of block coverage as a function of the fraction of the test suite used, for the two orders $T1$ and $T2$, respectively. The area under the curve represents the average of the percentage of block coverage over the life of the test suite. APDC and APSC are defined in a similar manner to APBC, except that they measure rate of coverage of decisions and of statements, respectively.

3.5 Analysis Tools

Statement, decision, and block coverage information was obtained using a tailor-made version of the commercial tool,

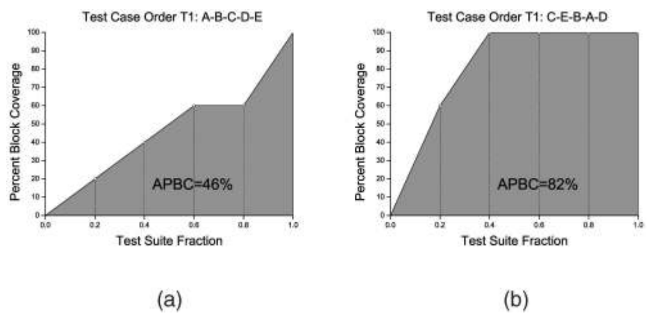


Fig. 2. Example illustrating the APBC measure. (a) APBC for order T1. (b) APBC for order T2.

Cantata++.[2] To analyze experimental data, SPSS was used to generate boxplots and perform an ANOVA (ANalysis Of VAriance) to investigate the statistical significance of any differences observed in the experiments. The "null hypothesis" is that the means of the fitness values (APBC, APDC, and/or APSC) for the five algorithms are equal. In order to decide whether to accept or reject the null hypothesis, the level of significance of the result was set at 0.05.

3.6 Experiments and Discussion

The experiments involved six C programs. For all the programs except Sed, there were 1,000 small test suites and 1,000 large test suites. All test suites were obtained from the same infrastructure [4] as the programs. In order to reduce the computation time for the experiments, without significant loss of generality, half of these test suites were used in the experiments. Thus, the results reported for all programs except Sed are averages obtained over 500 executions of the associated search algorithm, each execution using a different test suite. For the largest program, Sed, no division of all test cases into test suites was available.[3] Therefore, all available test cases were simply combined into a single test suite, and the search algorithms were each applied 100 times to obtain averages that capture typical performance.

For a program p and test case t for p, the blocks, decisions, and statements covered by t were found using Cantata++. For each program used, an instantiation of the experiment was conducted for each of the five algorithms, for each test suite, and for each of the three different coverage criteria.

3.6.1 Experiments with Small Test Suites

Fig. 3 shows boxplots of the fitness metrics APBC, APDC, and APSC for all programs with small test suites. Each row of subfigures indicates the results for one program.

As the graphs in Fig. 3 indicate, with the increase in program size, the differences between the five algorithms become more evident. For small programs, the five algorithms show almost identical performance. The mean fitness value for each program revealed that the Greedy Algorithm is the worst and the Genetic Algorithm is slightly better than the others.

First, consider the small programs with small test suites. The ANOVA results for these experiments, which are summarized in Table 13 in the Appendix, were remarkably consistent across different programs and coverage criteria. For example, in all cases, the difference between the GA and Additional Greedy is not statistically significant. Since these results are so similar, the results for the small programs with small test suites have been combined in order to form one group per algorithm. Table 4 presents an ANOVA analysis between the groups. The F-ratio can be computed from the ratio between the Mean Square for "between" groups and the Mean Square for "within" groups. A larger F-ratio indicates a greater confidence in rejecting the null hypothesis. Significance (p-value) reports the significance level of the F-ratio. The smaller the p-value ($< .05$), the stronger the evidence against the null hypothesis. The results in Table 4 show that there is strong evidence against the null hypothesis; the algorithms do, indeed, have significantly different performance.

However, rejecting the null hypothesis by ANOVA analysis can only tell us whether the five algorithms' performance was significantly different. To explore further, to see where the differences lie, a multiple-comparison procedure was performed. The LSD (Least Significant Difference) method was employed in multiple-comparison to compare the five algorithms pairwise.

Table 5 presents the results of this analysis for all of the small programs and reports the pairwise comparisons. For each pair of algorithms, the mean difference between the results of applying the algorithms is given, as is the significance level of the mean differences. If the significance is smaller than .05, the difference between the algorithms is statistically significant. As shown by the results, there was a significant difference between all pairs except Additional Greedy and GA, i.e., Additional Greedy and GA were not significantly different for small programs with small test suites. Since GA is slightly better than the others, Additional Greedy and GA produce the best results for small programs.

Table 6 and Table 7 present ANOVA results and LSD multiple comparison results, respectively, for the large program Space, considering all coverage criteria. For these experiments, there was no significant difference between the Additional Greedy and 2-Optimal Algorithms, instead of no significant difference between Additional Greedy and GA for small programs.

Returning to the Boxplots for the Space program with small test suites (the last row of subfigures in Fig. 3), the Additional Greedy and 2-Optimal Algorithms both show strong performance. However, the metaheuristic algorithms perform less well. Hill Climbing always appears to obtain a locally optimal value, indicating that there are local optima (foothills in the search landscape). The Genetic Algorithm performs much better than Hill Climbing, suggesting that the local optima are indeed merely local optima and better solutions are available. It is also noteworthy that the Genetic Algorithm is often a little worse than Additional Greedy and 2-Optimal Algorithms,

2. Cantata++ is a professional software testing tool, developed by IPL (http://www.iplbath.com). It has been designed to meet the requirements of the C/C++ languages to produce a tool that allows developers to perform efficient unit and integration testing. The product offers a set of testing, coverage analysis, and static analysis features.

3. Sed and test cases were obtained directly for Rothermel's early unpublished draft infrastructure.

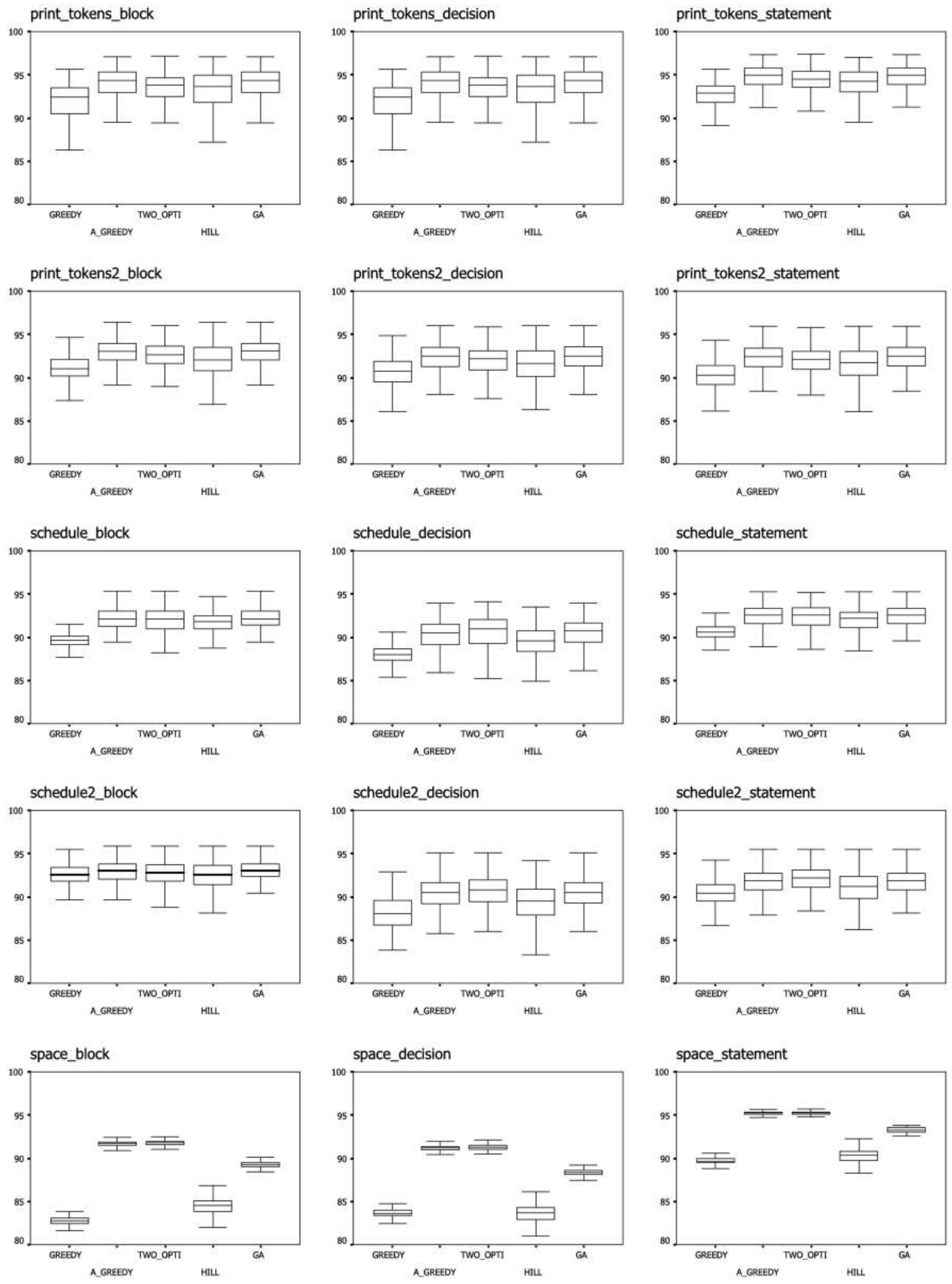


Fig. 3. Boxplots of APBC, APDC, and APSC for all programs with small test suites (vertical axis is the AP*C score) by program and by coverage criteria.

and the difference is significant. This suggests that, where applicable, the cheaper-to-implement-and-execute Additional Greedy and 2-Optimal Algorithms should be used.

The results obtained are similar for the different coverage criteria, providing evidence for the robustness of these results. That is, it would suggest that the results capture something of the nature of the search problem denoted by regression test case prioritization, rather than merely capturing an aspect of a particular choice of coverage criterion.

3.6.2 *Experiments with Large Test Suites*

Fig. 4 presents boxplots of the fitness metrics APBC, APDC, and APSC for all programs with large test suites.

TABLE 4
ANOVA Analysis for Small Programs with Small Test Suites

	Sum of Squares	df	Mean Square	F	Significance
Between Groups	15430.153	4	3857.538	776.233	.000
Within Groups	149062.028	29995	4.970		
Total	164492.181	29999			

TABLE 5
Multiple Comparisons (Least Significant Difference)
for Small Programs with Small Test Suites

Algorithm (x)	Algorithm (y)	Mean Difference $(x - y)$	Significance
Greedy	A–Greedy	-1.88928(*)	.000
	2–Optimal	-1.57361(*)	.000
	HC	-1.01015(*)	.000
	GA	-1.91488(*)	.000
A–Greedy	Greedy	1.88928(*)	.000
	2–Optimal	.31567(*)	.000
	HC	.87913(*)	.000
	GA	-.02560	.529
2–Optimal	Greedy	1.5736(*)	.000
	A–Greedy	-.3157(*)	.000
	HC	.5635(*)	.000
	GA	-.3413(*)	.000
HC	Greedy	1.01015(*)	.000
	A–Greedy	-.87913(*)	.000
	2–Optimal	-.56346(*)	.000
	GA	-.90473(*)	.000
GA	Greedy	1.91488(*)	.000
	A–Greedy	.02560	.529
	2–Optimal	.34127(*)	.000
	HC	.90473(*)	.000
(*) The mean difference is significant at the .05 level.			

Each row of subfigures indicates the results for a single program.

Tables 8 and 9 show the results of ANOVA analysis and LSD multiple comparisons for all small programs with large test suites. Tables 10 and 11 give the corresponding results for the Space program, also with large test suites. The data shows the same results as those produced from programs with small test suites. That is, there is no significant difference between Additional Greedy and the GA for small programs and no significant difference between Additional Greedy and the 2-Optimal Algorithm for Space. All other pairwise comparisons have significant differences.

These experiments with large test suites are consistent with those obtained for the small test suites. This lends additional evidence to support the claim that the results capture properties of the search problem in hand, rather than some artifact of the choice of test suite or the overall size of the test suites considered.

Necessity demanded a different experimental method for the program Sed. That is, the lack of an available subdivision into test suites meant that the single, large test suite was used in 100 repetitions of the experiment (100 repetitions were performed to allow for random effects inherent in search algorithms) executed on the same test suite (for the metaheuristic algorithms). The results produced by Hill Climbing and Genetic Algorithms vary with each experiment, while the results for the Greedy, Additional Greedy, and 2-Optimal Algorithms do not, since these algorithms are deterministic.

TABLE 6
Anova Analysis for the Large Program (Space)
with Small Test Suites

	Sum of Squares	df	Mean Square	F	Significance
Between Groups	75903.588	4	18975.897	2904.341	.000
Within Groups	48969.569	7495	6.534		
Total	124873.157	7499			

TABLE 7
Multiple Comparisons (Least Significant Difference) for the
Large Program (Space) with Small Test Suites

Algorithm (x)	Algorithm (y)	Mean Difference $(x - y)$	Significance
Greedy	A–Greedy	-7.34957(*)	.000
	2–Optimal	-7.41580(*)	.000
	HC	-.76127(*)	.000
	GA	-5.14745(*)	.000
A–Greedy	Greedy	7.34957(*)	.000
	2–Optimal	-.06623	.478
	HC	6.58829(*)	.000
	GA	2.20212(*)	.000
2–Optimal	Greedy	7.41580(*)	.000
	A–Greedy	.06623	.478
	HC	6.65452(*)	.000
	GA	2.26835(*)	.000
HC	Greedy	.76127(*)	.000
	A–Greedy	-6.58829(*)	.000
	2–Optimal	-6.65452(*)	.000
	GA	-4.38617(*)	.000
GA	Greedy	5.14745(*)	.000
	A–Greedy	-2.20212(*)	.000
	2–Optimal	-2.26835(*)	.000
	HC	4.38617(*)	.000
(*) The mean difference is significant at the .05 level.			

Consider the differences in the results for Hill Climbing in Fig. 3 and Fig. 4. Observe that both the size of the program and the size of the test suite do appear to influence the variance in the results. However, larger programs typically have larger test suites, since they usually require more test cases in order to achieve a given level of coverage. The program Space is about 10 times larger than the small programs, and the corresponding test suite is also about 10 times larger than the average size of test suites for small programs. The size of the program does not directly affect the complexity of test case prioritization, whereas the size of the test suite does since it determines the size of the search space. However, it takes longer to determine the fitness of a test case for a larger program. This suggests that the deterministic search techniques are scalable to larger programs, subject to the commensurate increase in the test suite sizes, since the fitness metrics are computed only once. For heuristic algorithms, the cost-benefits should be considered first before they are applied to large programs.

This finding is less pessimistic than it initially appears; it is likely to prove possible to reduce problem complexity by both reducing test suite size and reducing program size. However, it might be easier to reduce the test suite size. For example, techniques for automatically partitioning software into smaller units, such as slicing [21], tend to produce rather large (and overlapping) subprograms [3]. It is to be hoped that techniques for partitioning test data might be far easier to construct and would tend to produce a finer-grained partition. For example, tests might (naturally) be grouped according to the tester, to the revision for which

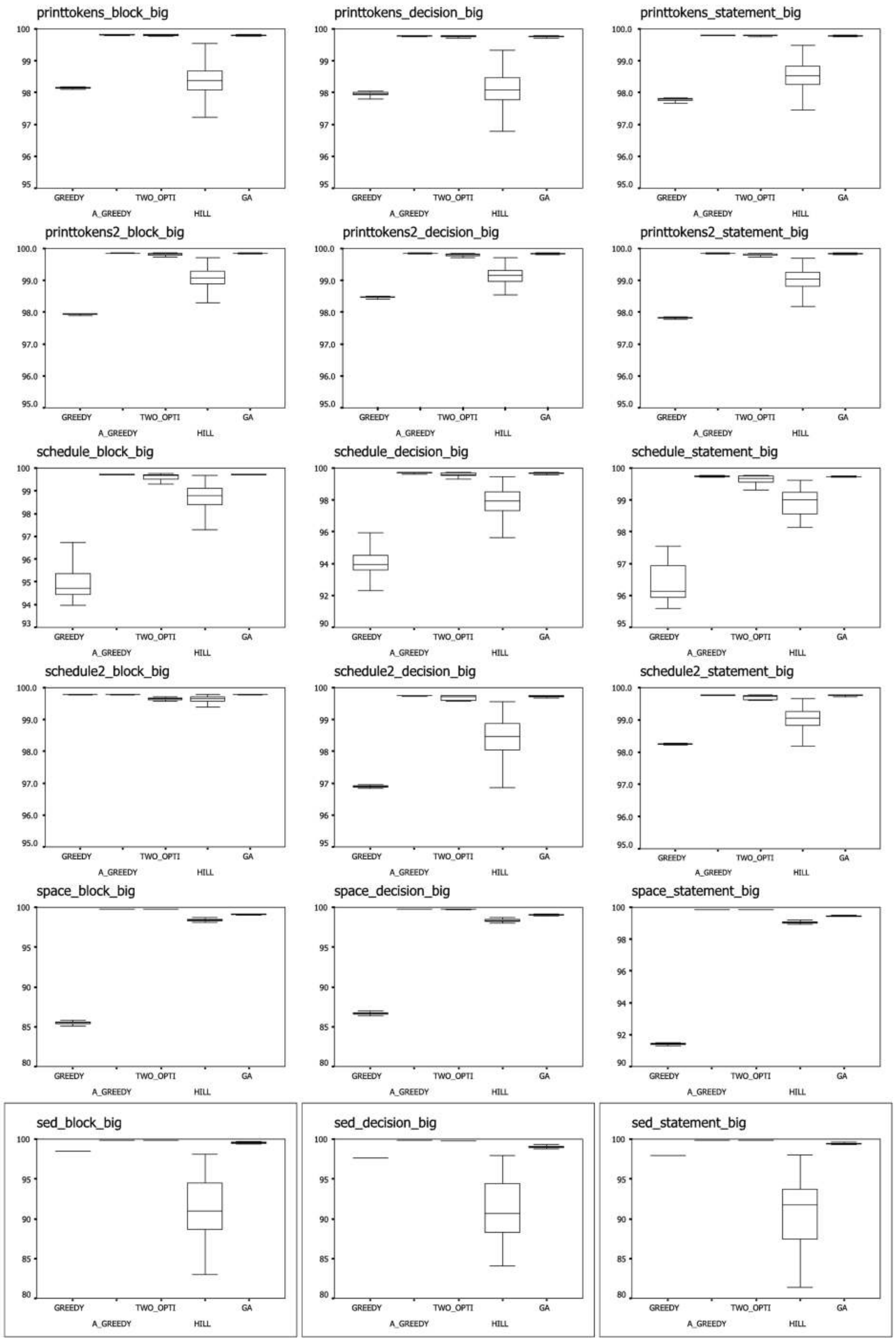


Fig. 4. Boxplots of APBC, APDC, and APSC for all programs with large test suites (vertical axis is AP*C score) by program and coverage criteria.

TABLE 8
ANOVA Analysis for Small Programs with Large Test Suites

	Sum of Squares	df	Mean Square	F	Significance
Between Groups	26571.857	4	6642.964	11923.726	.000
Within Groups	16710.860	29995	.557		
Total	43282.718	29999			

TABLE 9
Multiple Comparisons (Least Significant Difference) for Small Programs with Large Test Suites

Algorithm (x)	Algorithm (y)	Mean Difference ($x-y$)	Significance
Greedy	A–Greedy	-2.41255(*)	.000
	2–Optimal	-2.34864(*)	.000
	HC	-1.35876(*)	.000
	GA	-2.40274(*)	.000
A-Greedy	Greedy	2.41255(*)	.000
	2–Optimal	.06391(*)	.000
	HC	1.05379(*)	.000
	GA	.00982	.471
2–Optimal	Greedy	2.34864(*)	.000
	A–Greedy	-.06391(*)	.000
	HC	.98988(*)	.000
	GA	-.05410(*)	.000
HC	Greedy	1.35876(*)	.000
	A–Greedy	-1.05379(*)	.000
	2–Optimal	-.98988(*)	.000
	GA	-1.04397(*)	.000
GA	Greedy	2.40274(*)	.000
	A–Greedy	-.00982	.471
	2–Optimal	.05410(*)	.000
	HC	1.04397(*)	.000
(*) The mean difference is significant at the .05 level.			

TABLE 10
ANOVA Analysis for the Large Program (`Space`) with Large Test Suites

	Sum of Squares	df	Mean Square	F	Significance
Between Groups	29455.460	4	7363.865	8318.842	.000
Within Groups	1057.818	7495	.885		
Total	30513.278	7499			

TABLE 11
Multiple Comparisons (Least Signifcant Difference) for the Large Program (`Space`) with Large Test Suites

Algorithm (x)	Algorithm (y)	Mean Difference ($x-y$)	Significance
Greedy	A–Greedy	-12.80541(*)	.000
	2–Optimal	-12.80706(*)	.000
	HC	-11.52842(*)	.000
	GA	-12.17675(*)	.000
A-Greedy	Greedy	12.80541(*)	.000
	2–Optimal	-.00165	.985
	HC	1.27699(*)	.000
	GA	.62865(*)	.000
2–Optimal	Greedy	12.80706(*)	.000
	A–Greedy	.00165	.985
	HC	1.27864(*)	.000
	GA	.63031(*)	.000
HC	Greedy	11.52842(*)	.000
	A–Greedy	-1.27699(*)	.000
	2–Optimal	-1.27864(*)	.000
	GA	-.64833(*)	.000
GA	Greedy	12.17675(*)	.000
	A–Greedy	-.62865(*)	.000
	2–Optimal	-.63031(*)	.000
	HC	.64833(*)	.000
(*) The mean difference is significant at the .05 level.			

they were constructed, or for the aspect of the specification for which they were created. Importantly, all of these approaches *partition* the set of test data to be applied, whereas, by contrast, slices of a program may (and often do) contain large amounts of overlap.

3.6.3 Fitness Landscapes

The results obtained using Hill Climbing, both for small and large test suites, shed light on the nature of the search landscape and the factors that govern the properties of the fitness landscape. Naturally, it is important not to read too much into these results. However, some observations are worth noting.

For the problem of test case prioritization, the search space is the set of permutations of the ordering of a test suite. The global optimum is an order, or a set of orders, of a test suite for which no other order has a strictly superior fitness value; a local optimum is an order or a set of interconnected orders with a fitness value that is no worse than those of all its neighbors. For Hill Climbing, if the fitness of the returned order is less than the global optimum, then this order is termed "suboptimal."

In Fig. 3 and Fig. 4, the vertical lengths of the boxplots of Hill Climbing for small programs are similar or just a little longer than those for other algorithms. The variation could be a consequence of the use of 500 test suites, the height of boxplots merely reflecting 500 computations, each using a different test suite.

However, with the increase in program size and test suite size, there is a noticeable difference, i.e., the vertical lengths of the boxplots for Hill Climbing are much longer than those for the other techniques. This indicates that the fitness landscapes become much more complicated with the increase of both program size and test suite size. For the program `Sed`, where the metaheuristic algorithms were executed 100 times using the same test suite, the vertical length for Hill Climbing is much longer than for others. This confirms that there are many local optima with the consequence that Hill Climbing tends to alight upon a local optimum.

The fact that there are many local optima in the search space indicates that, for large test suites, the fitness landscapes are likely to be inherently multimodal. This result suggests that global search techniques could outperform local search techniques for regression testing prioritization problems, particularly for larger test suites.

3.6.4 Analysis of the Fitness Metric

The results presented here confirm that the Additional Greedy Algorithm is superior to the Greedy Algorithm. However, this is unsurprising because the addition of test cases that cover new aspects of the program as yet uncovered is bound to be better than the addition of new test cases that may merely cover what has already been covered.

It appears that the 2-Optimal Algorithm should overcome the weakness of the Greedy or Additional Greedy Algorithms shown by Table 1 in Section 1, so the 2-Optimal Algorithm should be better than, or no worse than, the Greedy and the Additional Greedy Algorithms. However,

TABLE 12
Test Suites and Block Coverage

Test Case	Block							
	1	2	3	4	5	6	7	8
A	X	X	X			X	X	X
B	X	X	X	X				
C					X	X	X	X

in some experiments, the results show the counterintuitive finding that the 2-Optimal Greedy Algorithm produces lower values of the fitness metric than the Additional Greedy Algorithm.

To understand this phenomenon, consider a program with a test suite containing three test cases, A, B, and C, such that the program has eight blocks covered by these test cases, as shown in Table 12. Two orders of these test cases are order $T1$: A-B-C, produced by the Greedy Algorithm or the Additional Greedy Algorithm, and order $T2$: B-C-A, produced by the 2-Optimal Algorithm. Fig. 5a and Fig. 5b show the percentages of blocks covered by the fraction of test suites used, for these two orders, respectively.

As Fig. 5a and Fig. 5b show, test order $T2$ uses two test cases to cover 100 percent of the blocks compared to $T1$ with three test cases, so $T2$ achieves 100 percent block coverage faster than T1. However, in terms of the rates of block coverage for the whole test suite, $T1$ produces an APBC of 70.83 percent, while $T2$ has an APBC of 66.67 percent; that is, $T2$ scores less than $T1$ according to the APBC metric. This sample program shows that the 2-Optimal algorithm can overcome the weakness of Greedy and Additional Greedy Algorithms, while obtaining a lower value from the APBC metric. However, with increasing test suite size, the differences between the Additional Greedy Algorithm and the 2-Optimal Algorithm become insignificant, which suggests that this issue may be ignorable in practice.

This phenomenon illustrates an interesting problem in determining fitness for $T1$ and $T2$. In Fig. 5a and Fig. 5b, the upper border line of the shaded area represents the block coverage rate. These two lines are combined into a single graph in Fig. 5c. The two lines cross at point P. Before the crossover point P, $T1$ denotes a higher coverage attained, but, after P, $T2$ is better than $T1$. The value associated with a result therefore depends upon where the "crossover point" P occurs. If a budget for regression testing is known and P falls after the budget is exhausted, then it is desirable to use $T1$ rather than $T2$.

Determining the point P is thus the key to determining whether $T1$ or $T2$ is the preferred order. Unfortunately, it is necessary to know the complete test case orderings in order to be able to determine which one is better. This situation may be thought of as a characteristic of a related family of regression testing prioritization problems, in which the entire order must be determined before a fitness value may be assigned to a candidate solution.

In such a situation, it will be difficult to define a greedy approach, because a greedy approach must determine which *part* of the solution to add next. For this class of problems, algorithms will be required that can be guided by a fitness function that takes account of the entire ordering. This observation makes Genetic Algorithms attractive, since they can be defined in this way and the results from Section 3.6.1 provide evidence that they can produce results of at least equivalent quality to those produced by Greedy Algorithms.

3.7 Threats to Validity

This section discusses some of the potential threats to the validity of the studies reported in this paper.

Threats to internal validity are influences that can affect the dependent variables without the researchers' knowledge and thereby affect any supposition of a causal relationship between the phenomena underlying the independent and dependent variables. In this study, the greatest concern for internal validity involves instrumentation effects, i.e., the quality of the coverage information measured. To limit problems related to this, a professional commercial software tool was used: Cantata++.

Threats to external validity are conditions that limit generalization from the results. The primary threats to external validity for this study concern the representativeness of the artifacts utilized. Only one approach to test suite generation, based on branch coverage adequacy, was used. This could represent a particular distribution of test cases that does not always occur in practice. This threat can be addressed only by additional studies, using a wider range of artifacts.

Threats to construct validity arise when measurement instruments do not adequately capture the concepts they are supposed to measure. For example, in the experiments, the fitness metrics focus on the measures of effectiveness of algorithms without measuring their cost. The time complexity for the first three algorithms was

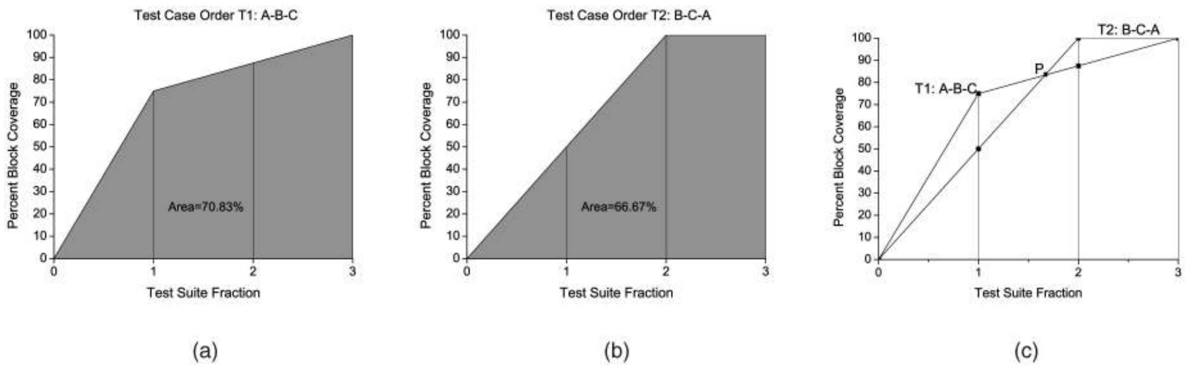


Fig. 5. Example of the pathological case. (a) APBC for order T1 by the Additional Greedy Algorithm. (b) APBC order for order T2 by the 2-Optimal Algorithm. (c) Crossover point.

stated, but no cost analysis was given for the two heuristic algorithms.

TABLE 13
The Summary Results of the ANOVA Analysis per Program with Different Coverage Criteria and Size of Test Suite (Corresponding to Each Subfigure in Fig. 3 and Fig. 4)

Program	Criterion	Algorithms which have no significant differences	
		Small test suite	Large test suite
Printtokens	block	A–Greedy:GA	A–Greedy:2–Optimal:GA
	decision	A–Greedy:GA	A–Greedy:2–Optimal:GA
	statement	A–Greedy:GA	A–Greedy:2–Optimal:GA
Printtokens2	block	A–Greedy:GA	A–Greedy:GA
	decision	A–Greedy:GA	A–Greedy:GA
	statement	A–Greedy:GA	A–Greedy:GA
Schedule	block	A–Greedy:GA 2–Optimal:HC	A–Greedy:GA
	decision	A–Greedy:2–Optimal:GA	A–Greedy:GA
	statement	A–Greedy:GA	A–Greedy:GA
Schedule2	block	A–Greedy:GA Greedy:2–Optimal:HC	A–Greedy:GA
	decision	A–Greedy:2–Optimal:GA	A–Greedy:GA
	statement	A–Greedy:GA	A–Greedy:GA
Space	block	A–Greedy:2–Optimal Greedy:HC	A–Greedy:2–Optimal
	decision	A–Greedy:2–Optimal Greedy:HC	A–Greedy:2–Optimal
	statement	A–Greedy:2–Optimal Greedy:HC	A–Greedy:2–Optimal
Sed	block		A–Greedy:2–Optimal
	decision		A–Greedy:2–Optimal
	statement		A–Greedy:2–Optimal

"`algorithm1:algorithm2`" stands for their being no significant difference between `algorithm1` and `algorithm2`.

4 Related Work

In [6], [17], [18], Rothermel et al. formally defined the test case prioritization problem and empirically investigated six prioritization techniques. Four of the techniques were based on the coverage of either statements or branches for a program and two of the techniques were based on the estimated ability to reveal faults. Several experiments compared these with the use of no prioritization (untreated), random prioritization and optimal prioritization. The experimental results showed that the prioritization techniques can improve the rate of fault detection of test suites. These experiments applied a Greedy Algorithm and an Additional Greedy Algorithm based on code coverage.

In [22], Wong et al. presented a way to combine test suite minimization and prioritization to select test cases, according to the criterion of "increasing cost per additional coverage." Greedy Algorithms were also used and were implemented in a tool named ATAC [22].

In [20], Srivastava and Thiagarajan studied a prioritization technique based on the changes that have been made to the program. Their technique orders the given test cases to maximally cover the affected parts of the program so that defects are likely to be found quickly and inexpensively. A test prioritization system, Echelon, that uses a Greedy Algorithm, was built based on this technique and it is currently being integrated into the Microsoft software development process. Echelon has been tested on large Microsoft product binaries and it has proved to be effective in ordering tests based on changes between two program versions.

The use of Greedy Algorithms for test case prioritization has been widely studied. However, no previous work has investigated the use of metaheuristic and evolutionary algorithms. This is thus the first paper to study metaheuristic and evolutionary algorithms empirically for test case prioritization for regression testing.

5 Conclusions and Future Work

This paper described five algorithms for the sequencing problem in test case prioritization for regression testing. It presented the results of an empirical study that investigated their relative effectiveness.

The data and analysis indicate that the Greedy Algorithm performs much worse than Additional Greedy, 2-Optimal, and Genetic Algorithms overall. Also, the 2-Optimal Algorithm overcomes the weakness of the Greedy Algorithm and Additional Greedy Algorithm (see Table 1) referred to by previous authors. However, the experiments indicate that, in terms of effectiveness, there is no significant difference between the performance of the 2-Optimal and Additional Greedy Algorithms. This suggests that, where applicable, the cheaper-to-implement-and-execute Additional Greedy Algorithm should be used.

The choice of coverage criterion does not affect the efficiency of algorithms for the test case prioritization problem. The size of the test suite determines the size of the search space, thereby affecting the complexity of the test case prioritization problem. The size of the program does not have a direct effect, but increases the difficulty of computing fitness values.

Studies regarding the performance of metaheuristic algorithms led to several conclusions that have practical ramifications. The results produced by Hill Climbing show that the nature of the fitness landscape is multimodal. The results produced by the Genetic Algorithm indicate that it is not the best of the five considered in all cases, but that, in most cases, the differences between the performance and that of the Greedy approach is not significant. However, an analysis of the fitness function shows that there are situations in which it is important to consider the entire ordering and, for such cases, Greedy Algorithms are unlikely to be appropriate (see Fig. 5). Given their generality, the fact that Genetic Algorithms perform so well is cause for encouragement.

The criteria studied were based on code coverage, which is different from criteria based on fault detection. The application of metaheuristic algorithms to fault detection-based prioritization problems could possibly yield different results, but this is a topic for future work.

Appendix

See Table 13.

Acknowledgments

This work was supported in part by a grant from the National Natural Science Foundation of China to the Beijing University of Chemical Technology, Grant No. 60473032. Mark Harman and Rob Hierons are supported by EPSRC grants (GR/R43150, GR/R98938, GR/T22872, and GR/S93684). The authors are very grateful to Gregg Rothermel, who provided the subject programs and many of their test suites and also participated in many valuable discussions about regression testing and related topics. Rothermel also provided many helpful comments on a previous draft of this paper. Finally, the anonymous reviewers provided valuable and detailed comments that have greatly improved the paper.

REFERENCES

[1] G. Antoniol, M.D. Penta, and M. Harman, "Search-Based Techniques Applied to Optimization of Project Planning for a Massive Maintenance Project," *Proc. 21st IEEE Int'l Conf. Software Maintenance (ICSM '05),* pp. 240-249, 2005.

[2] J.E. Baker, "Adaptive Selection Methods for Genetic Algorithms," *Proc. First Int'l Conf. Genetic Algorithms,* pp. 101-111, 1985.

[3] D.W. Binkley and M. Harman, "A Survey of Empirical Results on Program Slicing," *Advances in Computers,* vol. 62, pp. 105-178, 2004.

[4] H. Do, S.G. Elbaum, and G. Rothermel, "Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and Its Potential Impact," *Empirical Software Eng.: An Int'l J.,* vol. 10, no. 4, pp. 405-435, 2005.

[5] S. Elbaum, A. Malishevsky, and G. Rothermel, "Incorporating Varying Test Costs and Fault Severities into Test Case Prioritization," *Proc. 23rd Int'l Conf. Software Eng. (ICSE '01),* pp. 329-338, 2001.

[6] S. Elbaum, A.G. Malishevsky, and G. Rothermel, "Test Case Prioritization: A Family of Empirical Studies," *IEEE Trans. Software Eng.,* vol. 28, no. 2, pp. 159-182, Feb. 2002.

[7] M. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, 1979.

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

[9] P. Husbands, "Genetic Algorithms for Scheduling," *Artificial Intelligence and the Simulation of Behaviour (AISB) Quarterly,* no. 89, 1994.

[10] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments on the Effectiveness of Dataflow- and Control-Flow-Based Test Adequacy Criteria," *Proc. 16th Int'l Conf. Software Eng.,* pp. 191-200, May 1994.

[11] J.-M. Kim and A. Porter, "A History-Based Test Prioritization Technique for Regression Testing in Resource Constrained Environments," *Proc. 24th Int'l Conf. Software Eng.,* pp. 119-129, 2002.

[12] S Lin, "Computer Solutions of the Travelling Salesman Problem," *Bell System Technical J.,* vol. 44, pp. 2245-2269, 1965.

[13] "Regression Testing," Microsoft Corp., http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsent7/html/vxconregressiontesting.asp, 2007.

[14] "RTCA DO178-B: Software Considerations in Airborne Systems and Equipment Certification," Radio Technical Commission for Aeronautics, 1992.

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

[16] G. Reinelt, "TSPLIB—A Traveling Salesman Problem Library," *ORSA J. Computing,* vol. 3, no. 4, pp. 376-384, 1991.

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

[18] G. Rothermel, R. Untch, C. Chu, and M.J. Harrold, "Prioritizing Test Cases for Regression Testing," *IEEE Trans. Software Eng.,* vol. 27, no. 10, pp. 929-948, Oct. 2001.

[19] S.S. Skiena, *The Algorithm Design Manual.* Springer-Verlag, 1998.

[20] A. Srivastava and J. Thiagarajan, "Effectively Prioritizing Tests in Development Environment," *Proc. ACM SIGSOFT Int'l Symp. Software Testing and Analysis (ISSTA '02),* pp. 97-106, 2002.

[21] M. Weiser, "Program Slicing," *IEEE Trans. Software Eng.,* vol. 10, no. 4, pp. 352-357, 1984.

[22] W.E. Wong, J.R. Horgan, S. London, and H. Agrawal, "A Study of Effective Regression Testing in Practice," *Proc. Eighth Int'l Symp. Software Reliability Eng.,* pp. 230-238, Nov. 1997.

Zheng Li received the degree in computer science in 2004 from the Beijing University of Chemical Technology, China, where he also worked from 1996-2004. In 2005, he joined the software engineering group in the Department of Computer Science at King's College London. Currently, he is a research associate and PhD student, working on the EPSRC project ConTRACTs. His present research interests include search-based software engineering, source code analysis and manipulation, and dependence analysis.

Mark Harman is a professor of software engineering and the head of the Software Engineering Group in the Department of Computer Science, at King's College London, where he also directs the work of the Centre for Research on Evolution, Search and Technology (CREST). He has worked extensively on program slicing, transformation, and testing and more recently, he was instrumental in founding the field of search-based software engineering. Professor Harman's work is funded by the UK Engineering and Physical Sciences Research Council (EPSRC), the European Union, DaimlerChrysler, Berlin, and Vizuri Ltd., London.

Robert M. Hierons received the BA degree in mathematics (Trinity College, Cambridge) and the PhD degree in computer science (Brunel University). He then joined the Department of Mathematical and Computing Sciences at Goldsmiths College, University of London, before returning to Brunel University in 2000. He was promoted to full professor in 2003.