Optimizing for the Number of Tests Generated in Search Based Test Data Generation with an Application to the Oracle Cost Problem

Mark Harman[1], Sung Gon Kim[1], Kiran Lakhotia[1], Phil McMinn[2] and Shin Yoo[1]
[1]King's College London, CREST centre, Strand, London, WC2R 2LS, UK.
[2]The University of Sheffield, Regent Court, 211 Portobello, Sheffield, S1 4DP, UK.

***Abstract*—Previous approaches to search based test data generation tend to focus on coverage, rather than oracle cost. While there may be an aspiration that systems should have models, checkable specifications and/or contract driven development, this sadly remains an aspiration; in many real cases, system behaviour must be checked by a human. This painstaking checking process forms a significant cost, the oracle cost, which previous work on automated test data generation tends to overlook. One simple way to reduce oracle cost consists of reducing the number of tests generated. In this paper we introduce three algorithms which do this without compromising coverage achieved. We present the results of an empirical study of the effectiveness of the three algorithms on five benchmark programs containing non trivial search spaces for branch coverage. The results indicate that it is, indeed, possible to make reductions in the number of test cases produced by search based testing, without loss of coverage.**

I. Introduction

Previous work on Search Based Software Testing (SBST) has addressed many different programming paradigms and languages, including conventional 3GL code [1], Object Oriented systems [2], [3], Aspect Oriented systems [4] and Model Based systems [5], [6]. The SBST approach has proved to be highly generic, leading to its incorporation in many different testing scenarios including Stress Testing [7], Exception Testing [8], Mutation Testing [9], [10] Functional [11] and Non–Functional Testing [12].

The field is growing rapidly with over 340 papers according to a recent survey [13]. However, despite this considerable publication output, there is very little work on the oracle cost problem. That is, previous work concentrates on the problem of searching for good test inputs that can be used in these very many previously mentioned scenarios, but it does not address the equally important problem of reducing the cost of checking the output produced in response to the inputs generated. This observation also applies to work on Random Testing [14] and constraint solving [15] where the goal is also the pursuit of coverage, without regard for oracle cost.

One obvious way to reduce checking effort consists of finding ways to reduce the size of the test suite produced as a result of automated test data generation. However, the research challenge is to develop ways of achieving this goal without sacrificing the equally important goal of achieving coverage of the System Under Test (SUT). In order to do this, we seek test inputs that cover a targeted branch in the SUT, while also maximizing the so-called 'collateral' coverage [4]; coverage of branches not targeted, but hitherto, uncovered by any other test case. In this way we can reduce the overall number of test cases required to achieve full coverage.

Of course, one might hope that the SUT has been developed with respect to excellent design–for–test principles, so that there might be a detailed, and possibly formal, specification of intended behaviour. One might also hope that the code itself contains pre– and post– conditions that implement well-understood contract–driven development approaches [16]. In these situations, the oracle cost problem is ameliorated by the presence of an automatable oracle to which a testing tool can refer to check outputs, free from the need for costly human intervention

However, for many real systems, the tester has the luxury of neither formal specification nor assertions and must therefore face the potentially daunting task of manually checking the system's behaviour for all test cases generated. In such cases, it is essential that automated software testing approaches address the oracle cost problem. This paper takes some initial steps towards tackling this re-formulated version of the automated test data generation problem, making the following contributions:

1) We introduce a new formulation of the search based structural test data generation problem in which the goal is to maximize coverage, while simultaneously minimizing the number of test cases, with a view to taking into account the human oracle cost effort involved in checking the behaviour of the SUT for a given test suite.
2) We introduce three algorithms for addressing this extended re-formulation of the test data generation problem for search based software testing.
3) We present the results of an empirical study of the effectiveness of the three algorithms when applied to five programs containing search space sizes from 10^9 for the trivial benchmark `triangle` program to 10^{98} for the real-world function `check_isbn`, part of `Bibclean`, a BibTex file pretty printer and syntax checker.

The rest of the paper is organized as follows: Section II presents a brief overview of the background of search based software testing and the static analysis techniques used in this paper. Sections III and IV present the three algorithms used and a brief description of their implementation respectively. Section V presents the results of the empirical study. Section VI presents related work while Section VII concludes.

II. Background

Control Dependence: A node i *dominates* a node j if and only if every path from the entry node to the node j passes through node i. Conversely, a node j *post-dominates* a node i if and only if every path from the node i to the exit node traverses the node j. A node k *post-dominates* a branch $e = (i, j)$ if and only if every path from the node i to the exit node through e contains the node k. A node j is *control dependent* on a node i if and only if the node i dominates the node j and the node j post-dominates one and only one of the branches of the node i. A Control Dependence Graph (CDG) is a directed graph that captures control dependence [17]. Sibling nodes in a CDG that belong to one parent node, connected through the same edge, must, by definition, either have a dominance or a post-dominance relationship between each other. That is, should one of the nodes be executed, then the other sibling nodes must also be executed.

Search Based Software Testing: Meta-heuristic search techniques are methods which adopt heuristic mechanisms as the principal search strategies. The techniques are generally applied to complex problems when there exists no satisfactory algorithm for the problem or an existing algorithm is not practical with respect to computation time. Meta-heuristic techniques have also been applied to testing problems in a field known as Search Based Software Testing [12], [18], a sub-area of Search Based Software Engineering (SBSE) [19], [20]. Evolutionary algorithms are one of the most popular meta-heuristic search algorithms and are widely used to solve a variety of problems [13].

A fitness function for covering a target branch requires two principal components: an approach level and branch distance. The approach level [21], [18] indicates how close a candidate solution came to a target node in terms of control flow. This is achieved by counting the number of unexecuted nodes on which the target node is control dependent. Branch distance is a measure of how close a candidate solution came to satisfying the conditional expression in the last predicate executed on a sub-path to the target (that is, figuratively, where the path 'went wrong' and missed the target). The combination of these two components as a fitness function has repeatedly been demonstrated to be capable of guiding a search technique towards finding an input that covers a target branch [18].

Set Cover Problem: Set cover is one of the classic problems in complexity theory. The goal is to find a collection of minimal subsets of a set S that cover S. More formally:

Definition Let X be a finite set of size n, and let $\mathcal{F} = \{\mathcal{S}_1, \ldots, \mathcal{S}_k\}$ be a family of subsets of X, that is $\mathcal{S}_i \subseteq X$ such that $\bigcup_{i=1}^{k} \mathcal{S}_i = X$. A collection of subsets $C \subseteq \mathcal{F}$ is a *set cover* of X if $X = \bigcup_{\mathcal{S} \in C} \mathcal{S}$.

Though the set cover problem is NP hard, greedy algorithms are known to provide fast and relatively accurate approximations to the optimal solution [22]. That is, greedy algorithms can produce solutions of size n that are within $\log n$ of the optimal solution [23]. Figure 1 illustrates the set cover problem.

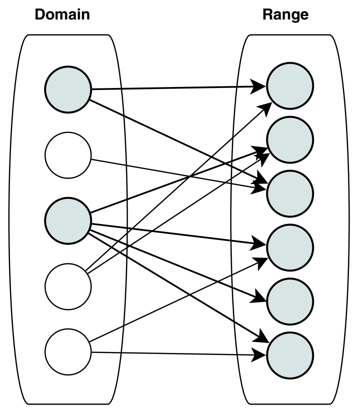


Fig. 1. An example of a set cover problem and its solution. Minimal subsets are obtained by choosing the 1st and 3rd elements from the domain. They cover all the elements of the range.

III. Algorithms for Reduced Oracle Cost Search Based Testing

This section introduces the three algorithms for 'Reduced Oracle Cost Search Based Software Testing' (ROC-SBST) studied in this paper. The memory based approach is, effectively, a codification of common sense and serves merely as a baseline against which to compare the other two algorithms based on greedy set cover and CDG analysis.

A. Memory-Based Test Data Reduction

In standard approaches to SBST, each currently uncovered branch is targeted by a distinct search process. The goal of previous work has been largely to cover branches, at any cost.

This is clearly sub-optimal from the point of view of reducing the number of test cases required to cover the program under test. In order to reduce the number of test cases it makes sense to record all branches hit by a test case that covers some particular branch of interest.

Algorithm 1 formalizes this observation as an algorithm, referred to as the memory-based approach. This algorithm is similar to tracking of 'serendipitous' coverage in the methods of Wegener et al. [21] and McGraw et al. [24]. Here, *any input* generated in the course of searching for a target that hits a previously uncovered branch may be inserted into the test suite as a separate test case. The aim is to ensure as high a coverage level as possible, rather than reduce the size of the test suites, as it is possible that several additional test cases may be inserted into the test suite that are found to cover different branches while attempting to execute the target.

```
Algorithm 1: Outline of the memory-based approach
Input $P$: target program; $\mathcal{B}$: set of all branches in $P$
Output $\mathcal{C}$: set of test cases
MEMORY-BASED-APPROACH($P$, $\mathcal{B}$)
(1)  $\mathcal{U} \Leftarrow \mathcal{B}$
(2)  $\mathcal{C} \Leftarrow \emptyset$
(3)  while $\mathcal{U} \neq \emptyset$
(4)      select a target branch, $t \in \mathcal{U}$
(5)      search for $x$ s.t. $t \in P(x)$
(6)      if $x$ is found
(7)          $\mathcal{U} \Leftarrow \mathcal{U} - P(x)$
(8)          $\mathcal{C} \Leftarrow \mathcal{C} \cup \{x\}$
(9)      else
(10)         $\mathcal{U} \Leftarrow \mathcal{U} - \{t\}$
(11) return $\mathcal{C}$
```

Let $\mathcal{B}$ be the set of all branches in the target program, P. Let $P(x)$ denote the set of branches covered by the execution of P with the input x. The memory-based approach maintains a set of remaining target branches, $\mathcal{U}$. Whenever an input x is generated for a specific target branch $t \in \mathcal{U}$, the approach also removes from $\mathcal{U}$ other branches that were covered by the execution of P with x, i.e. $\mathcal{U} \Leftarrow \mathcal{U} - P(x)$. Therefore, if x achieved collateral coverage of branches other than t, those branches will not be targeted again. In case the search fails to find an input x which covers the target branch t, the algorithm removes the branch t from the set of remaining branches U.

B. Greedy Algorithm for Set Cover Problem

In regression testing, the goal of test suite minimization [25], [26], [27] is to find a minimal collection of test data whose paths cover all of the reachable branches of the program. This is one kind of the set cover problem that can be solved by greedy algorithms. In the set cover–based approach to ROC-SBST, we first generate as many test cases as we can to cover the target branches (possibly multiple times and in different ways) and then select from these a subset that achieves coverage with the fewest test cases. This approach is attractively simple and (because of its use of greedy algorithms, that are known to be effective) it also generates small covering sets. However, the approach is computationally costly, because it requires the repeated use of search based test data generation as a pre–requisite for selection. The set cover–based algorithm is presented as Algorithm 2.

The greedy approach shown in Algorithm 2 consists of two stages. From line 1 to 6, the algorithm prepares the pool of test cases; for each branch in the target program, the algorithm generates a set of test cases using a search algorithm. The number of generated test cases for a specific branch is controlled by the stopping criterion used in line 6, which can be either a set number of test cases generated or a set number of fitness evaluations (i.e. computational resource) spent by the search algorithm for test data generation. In the second stage, from line 7 to 13, the algorithm applies a greedy-based test suite minimisation to the pool of test cases, $\mathcal{T}$. Note that $\mathcal{U}$ is initialised with all branches for which one or more test inputs exist. While there exist remaining branches, the algorithm selects a test case x from $\mathcal{T}$ such that x covers as many uncovered branches as possible. Then x is added to $\mathcal{C}$ and the branches covered by x, $P(x)$, are removed from the set of remaining branches, $\mathcal{U}$. Once all branches are covered, $\mathcal{C}$ contains a set of test cases that provides a maximum set cover of $\mathcal{B}$.

```
Algorithm 2: Outline of the greedy approach
Input $P$: target program; $\mathcal{B}$: set of all branches in $P$
Output $\mathcal{C}$: set of test cases
GREEDY-APPROACH($P$, $\mathcal{B}$)
(1)  $\mathcal{T} \Leftarrow \emptyset$
(2)  repeat
(3)      foreach $t \in \mathcal{B}$
(4)          search for $x$ s.t. $t \in P(x)$
(5)          $\mathcal{T} \Leftarrow \mathcal{T} \cup \{x\}$
(6)  until stopping criterion is met
(7)  $\mathcal{U} \Leftarrow \bigcup_{x \in \mathcal{T}} P(x)$
(8)  $\mathcal{C} \Leftarrow \emptyset$
(9)  while $\mathcal{U} \neq \emptyset$
(10)     select $x$ in $\mathcal{T}$ s.t. maximises $|P(x) \cap \mathcal{U}|$
(11)     $\mathcal{U} \Leftarrow \mathcal{U} - P(x)$
(12)     $\mathcal{C} \Leftarrow \mathcal{C} \cup \{x\}$
(13) return $\mathcal{C}$
```

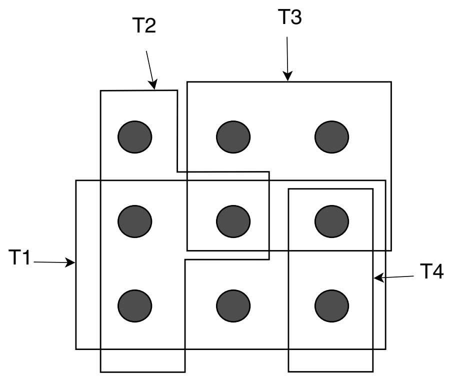


Fig. 2. An example of greedy algorithm for set cover problem. Here, a program consists of 9 branches and $T = T_1, T_2, T_3, T_4$. The greedy algorithm produces set cover of size 3 by selecting the test data: T_1, T_3 and T_2 in this order.

C. CDG-Based Test Data Reduction

While the standard approach to the fitness function, as described in Section II, is known to be effective for achieving structural coverage, it only concerns a single branch at a time. The resulting test suite, i.e. the collection of test cases that are generated using this fitness function, would naturally contain some redundant test cases, which in turn results in extra test oracle cost. If we want to reduce the size of the resulting test suite, each search process for test data should not only consider the achievement of a specific structural target but also the amount of *extra* structural coverage that the candidate test case can achieve. Combined with the memory-based approach, rewarding higher collateral coverage achieved by a test case would produce a smaller test suite.

```
Algorithm 3: Outline of the CDG-based approach
Input $P$: target program; $\mathcal{B}$: set of all branches in $P$
Output $\mathcal{C}$: set of test cases
CDG-BASED APPROACH($P$, $\mathcal{B}$)
(1)   $\mathcal{U} \Leftarrow \mathcal{B}$
(2)   $\mathcal{C} \Leftarrow \emptyset$
(3)   while $\mathcal{U} \neq \emptyset$
(4)       select $t \in \mathcal{U}$
(5)       search for $x$ s.t. $t \in P(x) \wedge$
          maximizes $|P(x) \cap \mathcal{U}|$
(6)       if $x$ is found
(7)           $\mathcal{U} \Leftarrow \mathcal{U} - P(x)$
(8)           $\mathcal{C} \Leftarrow \mathcal{C} \cup \{x\}$
(9)       else
(10)          $\mathcal{U} \Leftarrow \mathcal{U} - \{t\}$
(11)  return $\mathcal{C}$
```

Algorithm 3 formalizes this approach as a top level algorithm. The main difference between Algorithm 3 and the memory-based approach is in line 5. The CDG-based approach actively seeks to maximise the increase in coverage as well as achieving coverage of the target branch, t. The algorithm depends on the CDG representation of the target problem in order to accurately calculate the possible collateral coverage.

1) CDG and Coverable Branches: Consider a program as in Figure 3, where the two graphs - Control Flow Graph (CFG) on the left and CDG on the right - represent the same program. Suppose that edges in grey are previously covered, and the next target branch is 1F. The set of 1F's postdominating nodes contains node 6 and the exit node. Further, the edges 1F, 3T and 3F are control dependent on node 1, and edge 6F is control dependent on node 6. Those are amongst the branches that can be covered 'serendipitously' after targeting the original branch, 1F. However, with a single test input, only one of the two branches 3T and 3F can be covered. Therefore, the number of additional coverable branches when attempting 1F is 2 (1F and either 3T or 3F). We formalise this for programs without loops, then relax the definition of the collateral coverage to cater for loop control structures.

For programs without loops, the CDG allows an elegant recursive calculation of the number of potentially coverable edges. Take a node n in a CDG representation of a program. For branching nodes, let E represent the `true` and the `false` branches, i.e., $E = \{e_t, e_f\}$. For $e \in E$, Let N_e be the set of nodes that are dominated by the edge e of the node n. Similarly, let $L_e(n)$ be the number of potentially coverable edges when targeting the edge e. Finally, let M be the set of edges that are already covered. Then $L_e(n)$ is defined as follows:

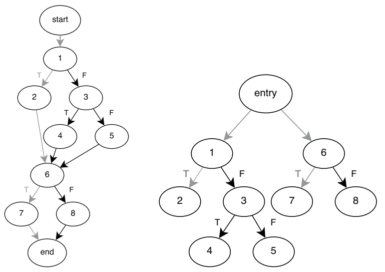


Fig. 3. Control Flow Graph (left) and Control Dependence Graph (right) which are used to demonstrate how coverable branches are calculated.

$$L_e(n) = \begin{cases} 0 \text{ if } n \text{ is not a branching node} \\ \displaystyle\sum_{n_i \in N_e} max(L_{e_t}(n_i), L_{e_f}(n_i)) \text{ if } e \in M \\ 1 + \displaystyle\sum_{n_i \in N_e} max(L_{e_t}(n_i), L_{e_f}(n_i)) \text{ if } e \notin M \end{cases}$$

2) Fitness Function for Collateral Coverage: Once the number of potentially coverable branches is calculated, it is possible to express the collateral coverage in more precise terms. Suppose that the CDG-based approach is evaluating the fitness of a candidate input x for covering a branch, e, which in turn belongs to a node n. Then the collateral coverage of x regarding the branch e of node n, $C(x, n, e)$, is defined as follows:

$$C(x, n, e) = \frac{|P(x) - M|}{L_e(n)}$$

The edges in $P(x) - M$ are the edges that are newly covered by x. If the edges in the CDG are targeted in a top-down order (i.e. the ones closer to the entry node are targeted first), then any edges that are newly covered by x should be also control dependent on e. This ensures that $|P(x) - M|$ is less than or equal to $L_e(n)$.

Using the definition of collateral coverage, we extend the standard definition of fitness function for test data generation. Let $f(x, n, e)$ represent the overall fitness of an input, x, that covers branch e of node n. Further, let $a(x)$ be the approach level and $b(x)$ the branch distance for the input x. Then $f(x, n, e)$ is defined as follows:

$$f(x, n, e) = (a(x) + normalize(b(x))) + (1 - C(x, n, e))$$

where

$$normalize(d) = 1 - 1.001^{-d}$$

The fitness function is to be minimised. The ideal fitness value is zero, which is achieved when x covers e and the maximum possible number of potentially coverable branches. In practice, the fitness function was split into two parts, $a(x) + b(x)$ and $1 - C(x, e)$, with the first part being the primary fitness and the second part the secondary. This is due to the fact that only the first part provides the actual guidance towards the search of a test input that will satisfy the specific condition required for the execution of a branch. The second part on the other hand is merely a *post-hoc* measurement of the collateral coverage that has been achieved; if the second part were to act as the primary guidance, the overall coverage achieved by the search algorithm will be significantly less than ideal.

3) Relaxation for Loops: The definition of $L_e(n)$ relies on the assumption that no single test case can execute both the `true` and the `false` branch of a predicate node. This assumption only holds when the predicate node is not a loop predicate or contained within a loop. For such predicates, a single test case can execute both the `true` and the `false` branch. This means that, for loop predicates, $|P(x) - M|$ for a node n can be greater than $L_{e_t}(n)$ or $L_{e_f}(n)$, resulting in a collateral coverage value higher than 1.

However, it is not feasible to determine in general whether there exists a test input that will complete a loop; otherwise we would solve the halting problem. Therefore, instead of doing the exact analysis for the number of coverable branches for loops, we relax the definition of $C(x, n, e)$ to cater for loops as follows:

$$C(x,n,e) = \begin{cases} 1, \text{ if } |P(x) - M| > L_e(n) \\ \dfrac{|P(x) - M|}{L_e(n)}, \text{ otherwise} \end{cases}$$

IV. Implementation

Each of the three approaches described in Section III was implemented on top of IGUANA [28], a test data generation framework using search based approaches. It provides useful features, such as code instrumentation, control flow analysis, and a variety of search algorithms.

A. Greedy Approach

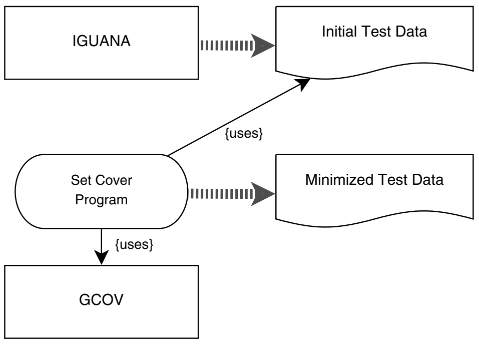


Fig. 4. Flowchart of the greedy algorithm program.

Figure 4 is a diagrammatic representation of the implementation of the greedy approach. The greedy approach uses IGUANA for its first stage; the stopping criterion was set according to the size of the resulting test suites, which ranged from 200 to 500 depending on the program. The second stage of the greedy approach, a solver for the set cover problem, was written as a `perl` script. Since the set cover solver exists outside IGUANA, it uses GNU `gcov` profiler to collect the execution path information of each test case.

B. Memory-based and CDG-based approach

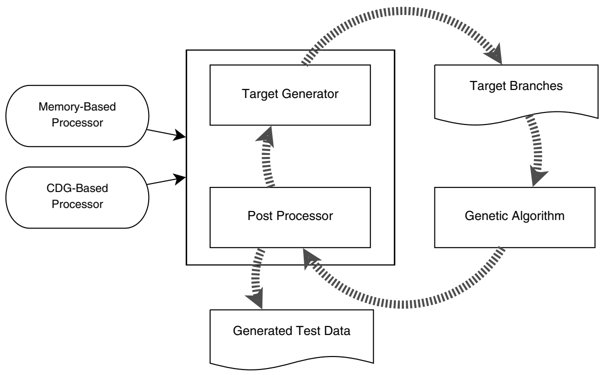


Fig. 5. Flowchart of the memory-based approach and the CDG-based approach.

Both the memory-based and CDG-based approaches were implemented as extensions to IGUANA [28]. The overall process is depicted in Figure 5. The `MemoryBasedTargetGenerator` implementation is merely a straightforward optimisation of the existing IGUANA test data generation approach. If a search algorithm requests a new target branch, it returns the first in the list. Once the search algorithm found the ideal solution which traverses the target branch, the path is inspected in order to identify collaterally covered branches. These are marked as visited. Should the algorithm fail to find a covering input, then the fittest candidate solution is added to the test suite if it covers any remaining uncovered branches. Otherwise, the candidate solution is discarded. The search continues until no more target branches remain.

The `CDGBasedTargetGenerator` implementation initially generates a control dependence graph of the program under test. Each time a search algorithm requests a new target branch, the target generator first updates the coverable branch value for all remaining target branches. Then, it returns the target closest to the entry node with the associated number of potentially coverable branches. This number is used by the genetic algorithm inside IGUANA to calculate the collateral coverage fitness. Once the algorithm finds a solution, the trace is inspected and accepted if the solution covers any remaining branches. Note that the trace does not have to be *ideal* in a sense that it covers the original target branch and also achieves 100% collateral coverage possible. However, if the trace fails to cover any remaining branch, it is discarded. The search attempt continues until no more target branches are left.

C. Genetic Algorithm Setup

A genetic algorithm is used to search for test data to cover a given target branch. This section details the parameter settings used in our study in order to facilitate replication of our results. For the selection operation, stochastic universal sampling [29] was used, where the probability of individual selection is biased according to an individual's fitness value. This means the higher the fitness value an individual has, the higher its chance to be chosen, but less fit individuals may still be selected with a low probability. The approach seeks to maintain the diversity within the population. Before selection takes place for crossover, individuals within the population are ranked according to their fitness value. Linear ranking [30] is used to rank the individuals, while discrete recombination [31] is used to generate offspring from the selected parent individuals. The mutation operation is based on the breeder genetic algorithm [31]. For covering a specific branch, the combination of approach level and branch distance measure is applied to the algorithm as a fitness function, as described in Section II.

For the CDG-based approach, the fitness function described in Section III-C2 was split into primary and secondary fitness. The primary fitness alone is applied until the original target branch is covered. However, once the search algorithm finds more than one test input that achieves the coverage of the original target branch, it applies the secondary fitness to evaluate the candidate test inputs according to the amount of collateral coverage achieved.

V. Empirical Study

An empirical study was performed that compared the standard search based test data generation algorithm, which generates test data for each branch individually, against the three reduced oracle cost algorithms detailed in Section III.

A. Test Subjects

Five programs were used as test subjects. `check_isbn` is a function from the open source `bibclean` program, which validates ISBNs in BibTex files. `clip_to_circle` is part of the `spice` open source analogue circuit simulator, while `euchk` is an open source program used to validate serial numbers on European bank notes. `gdkanji` is a function that forms part of the `gdLibrary`, an open source graphic package library. Finally, `triangle` is the well-known triangle classification program, often used as a benchmark program in automatic test data generation studies.

Further details regarding each program are recorded in Table I. The test suite size for the standard approach is equal to each program's number of branches if 100% coverage is obtained. Cyclomatic complexity gives an upper bound on the number of test cases required to cover all feasible branches if collateral coverage is taken into account. Cyclomatic complexity is therefore a useful statistic to compare against the test suite sizes generated by the ROC-SBST algorithms.

TABLE I
Details of the test subjects used in the empirical study

Program	Lines of code	Branches	Cyclomatic complexity	Approx. Domain Size (10^n)
check_isbn	144	44	23	98
clip_to_circle	156	42	22	35
euchk	74	18	10	23
gdkanji	140	58	30	12
triangle	60	20	11	9

B. Experimental Setup

The three reduced oracle cost algorithms (memory-based, CDG-based and greedy set cover algorithm) were tested alongside the standard search based approach, which attempts to cover each branch individually. Due to the stochastic nature of the algorithms, the experiments were repeated 15 times and the numbers reported are averages over these 15 runs.

In order for the set cover approach to have a good chance of achieving high coverage, it is necessary for the test case generation phase (implemented using IGUANA) to generate a large number of test cases. These test cases must cover branches in the SUT multiple times and in different ways in order to ensure that the set cover algorithm has a good range of options from which to construct a good minimal set cover.

This is the primary reason why the set cover approach is inefficient (though effective); the inefficiency lies in the generation of a sufficiently large initial pool of test case from which the selection phase can choose. Of course, the whole process is entirely automated and so it is only inefficient in machine time, and not in human analysis time, which is the more precious commodity and that which we wish to preserve in order to reduce oracle cost. In our experiments we set the maximum number of test cases to be generated to 500. The choice of upper limit has to be determined for the SUT in question. In our experiments this number was chosen based upon initial experimentation, from which we found that the set cover approach failed to noticeably increase effectiveness for larger pools of test data.

It is these limitations of the set cover approach that motivate the introduction of the more computationally efficient CDG–based approach. The CDG–based approach, performs a static analysis to determine the branches to cover and uses a secondary fitness to increase the collateral coverage achieved when targeting a branch in the SUT. This obviates the need for a large pre-generated test pool and thereby also removes the need for the determination of this initial test pool size.

C. Results

The average test suite size and branch coverage achieved by each algorithm for each test subject can be seen in the bar charts of Figure 6. The average test suite sizes produced by the reduced oracle cost algorithms were significantly smaller than that of the standard approach. For every test subject, average test suite size was smaller than the subject's cyclomatic complexity number. As can also be seen in the figure, this reduced test suite size did not have a compromising effect on branch coverage of the program. For certain subjects and algorithms, the reduced cost algorithms managed to exceed the average level of coverage obtained by the standard individual-branch approach.

Figure 7 shows the additive branch coverage of each test case search performed by the reduced oracle cost algorithms. Generally speaking, the memory-based approach has to initiate the most searches to achieve coverage levels comparable with the CDG-based and greedy set cover method. The memory-based approach steadily covers a small number of branches in each search, whilst the CDG-based method and set cover

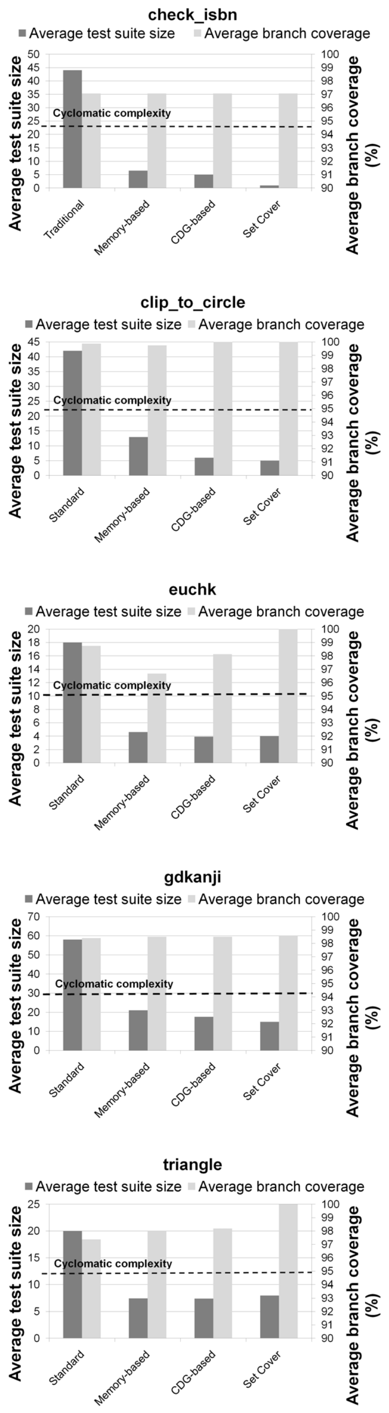


Fig. 6. Average test suite size and branch coverage using the different algorithms. All reduced oracle cost algorithms are successful at reducing test suite size without compromising coverage and are smaller than each program's cyclomatic complexity number.

algorithm achieve most of their coverage in the early searches, covering the proportionally fewer remaining branches that remain after this 'initial burst'.

The results are now discussed in detail for each test subject.

check_isbn. The `check_isbn` program contains a large loop with small nested statements within it. All methods achieved approximately 97% coverage, with set cover producing the smallest test suite (consisting of just 1 test case).

clip_to_circle. Again, set cover produced the smallest test suite, achieving 100% coverage at the same time. The average test suite produced by the CDG-based approach is less than half the size of that produced by the memory-based approach. Figure 7 for `clip_to_circle` shows how the CDG-based and set cover methods attempt to achieve as much branch coverage as early as possible, whilst the memory-based approach maintains steady coverage, finding test cases that execute only a small number of branches with each additional search.

euchk. In terms of test suite size, both the memory-based and the CDG-based approaches are able to find test suites that are almost as small as that found by the set cover approach. When considering the performance over each individual consecutive search (Figure 7) with respect to branch coverage, the CDG-based approach achieves its comparable level of coverage in almost half as many searches as the memory-based method for `euchk`.

gdkanji. The CDG-based approach performs better than the memory-based approach with respect to test suite size as well as the number of search attempts. The clear overall 'winner' however, is the greedy set cover method. It achieves the smallest test suite using the fewest number of searches[1].

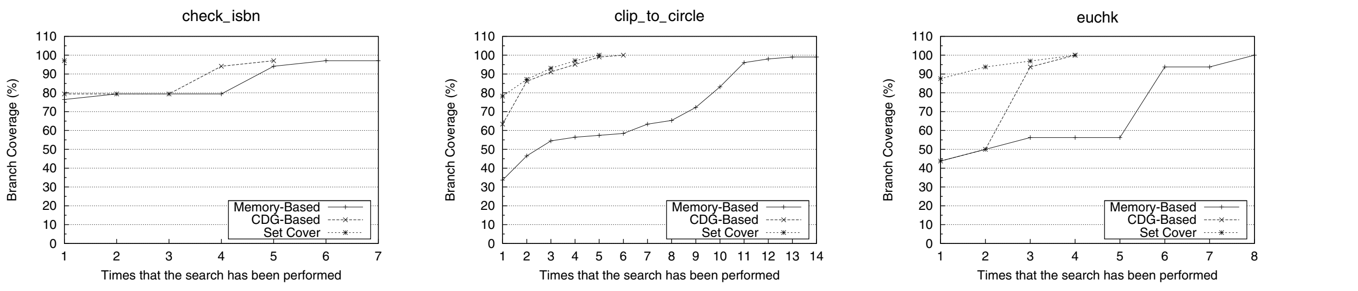
triangle. For the `triangle` program there is one branch that is difficult for search based approaches, resulting in less than 100% coverage for all algorithms except the greedy set cover algorithm. Failure of the memory and CDG-based approaches to cover this branch results in slightly smaller test suites for this algorithm compared to greedy set cover.

D. Analysis

In terms of test suite size, the standard approach is the worst; although this is not surprising, given that it attempts to find a separate test case for each individual branch. If the search is able to keep track of collateral coverage, as with the memory-based approach, the number of test cases in the test suite is always reduced to a size that is less than the program's cyclomatic complexity. However, our results indicate that this situation can be further improved by effectively targeting more deeply nested branches using the CDG-based approach, which results in smaller test suite sizes using fewer distinct searches to do so. The clear winner with respect to test suite size is the greedy set cover algorithm. There was only one program (`triangle`) for which the greedy set cover method had a larger test suite size than the CDG-based and memory-based approaches, and this was because an additional 'hard-to-cover' branch had been covered that the other algorithms had not.

The set cover approach is very effective at generating small test suites, since it is based on a known near optimal greedy algorithm. However it has the (non trivial) drawback that it requires a 'suitable initial test pool' of test cases to be pre–generated. This is a research problem in itself, since test case generation for multiple coverage remains a topic of current

[1]The number of searches reported exludes the number of searches required to generate the initial pool of test data.



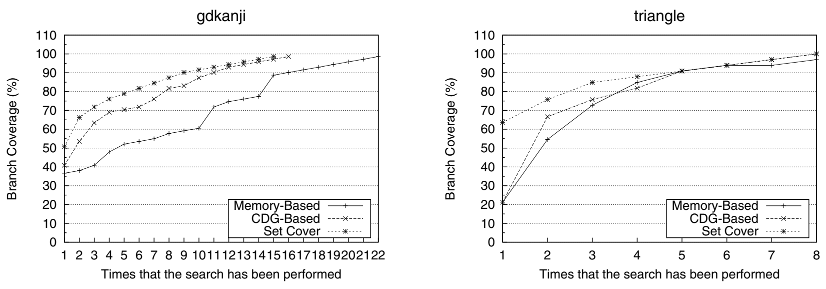


Fig. 7. Coverage performance of each reduced oracle cost algorithm relative to the number of searches that needed to be initiated. For the set cover method only the searches initiated by the greedy algorithm are reported. The tendency of the CDG-based and set cover methods is to achieve as much branch coverage as early as possible, whilst the memory-based approach steadily improves coverage over a longer period; each test case found executes a relatively smaller number of branches.

research. Our experimental results do, however, confirm, that with a suitable test pool, the set cover approach can generate good results. They also indicate, that by repeatedly executing a 'standard' SBST test data generation tool (IGUANA in this case) it is possible, for those programs studied herein, to create such a suitable initial test pool.

The results for the `check_isbn` program highlight the potential drawback of the CDG-based approach, which is unable to achieve results comparable to the set cover algorithm in terms of the size of reduced test suite. This is due to the fact that the `check_isbn` program has a large loop structure containing within, a large number of small nested conditionals. This structure offers considerable potential for test suite reduction, since a single test case may be able to traverse the loop multiple times, covering different nested predicate combinations on each iteration. If the 'suitable initial test pool' used by the set cover approach is rich enough, it will contain test cases that perform different traversals of the nested predicates. The set cover algorithm will then be able to select from these, a minimal (or near minimal) set that achieves coverage of all (or nearly all) nested predicates with fewest test cases.

E. Limitations and Threats to Validity

The empirical study in this paper has been primarily concerned with determining the feasibility of reducing test suite size, while maintaining coverage and in evaluating the effectiveness of the set cover–based and CDG–based approaches. The CDG–based and set–cover–based approaches are novel to this paper, but the memory based approach is merely a codification of 'common sense', similar to that used in existing implementations; it simply applies existing SBST test data generation techniques in a 'sensible' way to avoid unnecessarily large test suite sizes.

The empirical study results are promising, but further empirical studies are required to evaluate the algorithms proposed here (and also to develop and evaluate other, possibly hybrid, approaches).

The results presented attract the threats to validity that are commonly found in empirical studies of software test data generation techniques. This section considers these threats to validity and limitations and their implications.

There is a threat to external validity that limits the extent to which the results can be generalized. We have selected five programs for the study. These include the widely studied (but relatively trivial) `triangle` program. This is included merely because of the wide use of this example in other studies. Due to its small size and synthetic (as opposed to real world) application domain, results concerning this subject are included merely for 'historical compatibility'.

We have also selected four other programs for study. This was far from a 'random sample' of all possible programs. In choosing these four, we were careful to select those that denoted challenging problems, with large search space sizes, non–trivial branch nesting and real world applications.

However, like any set of programs used in a study of this nature, results obtained from these subjects cannot necessarily be generalized to other programs, languages or programming paradigms. This is particularly true, precisely because these subjects were not chosen at random, but were selected more as a set of case studies that denote challenging search problems. All that can be said with absolute certainty from our results is that there is evidence to support the claim that the size of test suites can be reduced from that produced by the current state–of–the–art in search based test data generation, while maintaining coverage.

The results concerning test suite size are relatively free from threats to construct validity, since the measurement used is straightforward and intuitive (set size). However, as is well known, branch reachability is undecidable [32] and so it cannot be known whether uncovered branches are uncovered because they are infeasible, or whether the test suite is simply insufficiently powerful to cover them. As a result, the findings relating to coverage are affected by a threat to construct validity; low coverage results may appear to be artificially low for subjects with a large number of infeasible branches. Fortunately, as can be observed from our results, the coverage for these subject programs is extremely high and so we can conclude that there are very few, if any, infeasible branches present.

The results of the study are also vulnerable to threats to internal validity. We only make the claim that there is evidence to indicate that both algorithms, CDG–based and set cover–based, are capable of producing smaller covering test suites than existing approaches. We do not seek to make any conclusive claims regarding the relative performance of each, but prefer to use our study to present descriptive statistics concerning their behaviour on the programs studied.

For these, we can say that there is evidence to suggest that the algorithms do behave in different ways and that the set cover approach can achieve smaller test suites. However, there are too many confounding factors to be able to make more definitive claims. For example, the performance of the set cover approach is strongly influenced by the quality and diversity of the test pool from which it draws a subset. In order to fully evaluate its performance while taking into account these potential confounding effects, a more detailed and sophisticated controlled trial would be required; one which would take more space to present than is possible in a ten–page conference paper.

VI. Related Work

The work reported in this paper draws from two sources; search based software test data generation and test suite minimization. The former is used to generate test data, while the latter is used to reduce the size of the test suites so–generated. Test suite minimization concerns reducing the size of a regression test suite which grows over time as the software evolves [25], [26]. The majority of the literature on test suite minimization [33], [34], [35] differs from the work in this paper as the existing minimization techniques are *post-hoc* processes applied to existing test suites. This paper incorporates the minimization within the test data generation phase. Leitner et al. introduced a technique for minimizing, i.e. shortening unit test cases in order to reduce testing cost [36]. While this paper shares a similar goal, the work reported in this paper operates on the generation of a coverage-adequate test suite rather than a single unit test.

The work reported here also draws on the widely–studied Search Based Software Engineering approach to software test data generation, in which the test data generation problem is reformulated as a search problem [13], [18].

Previous work on SBST has addressed many different programming paradigms and languages, including the C language [1], [37], [21], object-oriented systems [2], [3], aspect-oriented systems [4] and model-based systems [5], [6]. The SBST approach has proved to be highly generic, leading to its incorporation in many different testing scenarios including stress testing [7], exception testing [8], mutation testing [9], [10], functional [11] and non–functional testing [12].

However, as mentioned in the introduction to this paper, there is very little work on the oracle cost problem.

'Serendipitous coverage' has been taken advantage of by some techniques [21], [24], and is a similar concept to collateral coverage as described in this paper. However, serendipitous coverage occurs when any input generated in the course of covering the target executes a previously uncovered branch. Collateral coverage records only branches covered in the same path as the target in a single test case, and therefore has greater potential for test suite reduction (and thus reduced oracle cost). Serendipitous coverage is largely concerned with ensuring that the highest possible level of coverage is achieved. This is also the primary concern of the so-called 'coverage-oriented' fitness functions of Roper [38] and Watkins [39], where inputs are rewarded on the basis of the number of targets they execute. While such fitness function formulations may inherently incorporate some degree of oracle-cost-reducing capability, new experiments would be required to validate any such potential.

Reducing oracle cost is the primary novelty of the present paper; it re–formulates the test data generation problem as one in which the human oracle cost is reduced by minimizing the number of test cases generated, while attempting to achieve maximal coverage of the SUT.

VII. Conclusions and Future Work

This paper motivated a reformulation of automated test data generation from achievement of test goals (without consideration for oracle cost) to the problem of balancing cost and benefit. The paper sought to determine the extent to which search based testing techniques could be adapted to produce fewer test cases without loss of coverage, presenting empirical results to support the claim that the approach can reduce cost without an impact on the benefits that accrue from coverage. The paper argues that more work is required in this area of search based testing.

Our results and the observations of the relative strengths and weakness of the CDG–based approach and the set cover based approach allow us to make some suggestions for future work. In order to achieve better results for nested predicate structures inside loops, a form of multi objective search may be suitable for extending and developing the CDG–based approach. In such an approach, it may be possible to use a concept similar to the 'crowding distance' measurement used in the multi objective search algorithm NSGA–II [40] in order to maintain a diversity of branch coverage within loops.

It also suggests that some form of hybrid algorithm may be required for optimal effectiveness, in which the CDG–based approach is used to generate a small initial test pool which is highly optimized for coverage diversity as well as maximal collateral coverage, from which the set cover approach could select. Such a hybrid may be capable of combining the best features of both CDG and set cover approaches. However, this remains a topic for future work.

VIII. Acknowledgments

Mark Harman is supported by EPSRC Grants EP/G060525/1, EP/D050863, GR/S93684 & GR/T22872 and also by the kind support of Daimler Berlin, BMS and Vizuri Ltd., London. Kiran Lakhotia is funded by EPSRC grant EP/G060525/1. Phil McMinn is supported in part by EPSRC grants EP/G009600/1 (Automated Discovery of Emergent Misbehaviour) and EP/F065825/1 (REGI: Reverse Engineering State Machine Hierarchies by Grammar Inference). Shin Yoo is supported by the EPSRC SEBASE project (EP/D050863).

References

[1] M. Harman and P. McMinn, "A theoretical and empirical study of search based testing: Local, global and hybrid search," *IEEE Transactions on Software Engineering*, vol. 36, no. 2, 2010, To Appear.

[2] P. Tonella, "Evolutionary testing of classes," in *ISSTA*, 2004, pp. 119–128.

[3] A. Arcuri and X. Yao, "Search based software testing of object-oriented containers," *Inf. Sci*, vol. 178, no. 15, pp. 3075–3095, 2008.

[4] M. Harman, F. Islam, T. Xie, and S. Wappler, "Automated test data generation for aspect-oriented programs," in *AOSD*, 2009, pp. 185–196.

[5] K. Derderian, R. M. Hierons, M. Harman, and Q. Guo, "Input sequence generation for testing of communicating finite state machines (CFSMs)," in *GECCO*, 2004, pp. 1429–1430.

[6] Y. Zhan and J. A. Clark, "Search-based mutation testing for simulink models," in *GECCO*, 2005, pp. 1061–1068.

[7] V. Garousi, L. C. Briand, and Y. Labiche, "Traffic-aware stress testing of distributed real-time systems based on UML models using genetic algorithms," *Journal of Systems and Software*, vol. 81, no. 2, pp. 161–185, 2008.

[8] N. Tracey, J. Clark, and K. Mander, "The way forward for unifying dynamic test-case generation: The optimisation-based approach," in *International Workshop on Dependable Computing and Its Applications (DCIA)*, 1998, pp. 169–180.

[9] Y. Jia and M. Harman, "Constructing subtle faults using higher order mutation testing," in *SCAM*, 2008, pp. 249–258.

[10] Y. Zhan and J. A. Clark, "The state problem for test generation in simulink," in *GECCO*, 2006, pp. 1941–1948.

[11] J. Wegener and O. Bühler, "Evaluation of different fitness functions for the evolutionary testing of an autonomous parking system," in *GECCO*, 2004, pp. 1400–1412.

[12] W. Afzal, R. Torkar, and R. Feldt, "A systematic review of search-based testing for non-functional system properties," *Information and Software Technology*, vol. 51, no. 6, pp. 957–976, Jun. 2009.

[13] M. Harman, A. Mansouri, and Y. Zhang, "Search based software engineering: A comprehensive analysis and review of trends techniques and applications," Department of Computer Science, King's College London, Tech. Rep. TR-09-03, April 2009.

[14] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," *ACM SIGPLAN Notices*, vol. 40, no. 6, pp. 213–223, Jun. 2005.

[15] R. A. Demillo and A. J. Offutt, "Experimental results from an automatic test case generator," *ACM Transactions on Software Engineering and Methodology*, vol. 2, no. 2, pp. 109–127, Apr. 1993.

[16] B. Meyer, "Eiffel: A language and environment for software engineering," *The Journal of Systems and Software*, vol. 8, no. 3, pp. 199–246, Jun. 1988.

[17] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 3, pp. 319–349, Jul. 1987.

[18] P. McMinn, "Search-based software test data generation: a survey," *Softw. Test, Verif. Reliab*, vol. 14, no. 2, pp. 105–156, 2004.

[19] M. Harman and B. F. Jones, "Search-based software engineering," *Information & Software Technology*, vol. 43, no. 14, pp. 833–839, 2001.

[20] M. Harman, "The current state and future of search based software engineering," in *FOSE*, 2007, pp. 342–357.

[21] J. Wegener, A. Baresel, and H. Sthamer, "Evolutionary test environment for automatic structural testing," *Information & Software Technology*, vol. 43, no. 14, pp. 841–854, 2001.

[22] A. M. Smith and G. M. Kapfhammer, "An empirical study of incorporating cost into test suite reduction and prioritization," in *SAC*, 2009, pp. 461–467.

[23] D. S. Johnson, "Approximation algorithms for combinatorial problems," in *Proceedings of the 5th annual ACM Symposium on Theory of Computing (STOC 1973)*, 1973, pp. 38–49.

[24] C. C. Michael, G. McGraw, and M. Schatz, "Generating software test data by evolution," *IEEE Trans. Software Eng*, vol. 27, no. 12, pp. 1085–1110, 2001.

[25] M. J. Harrold, R. Gupta, and M. L. Soffa, "A methodology for controlling the size of a test suite," *ACM Transactions on Software Engineering and Methodology*, vol. 2, no. 3, pp. 270–285, Jul. 1993.

[26] J. Offutt, J. Pan, and J. Voas, "Procedures for reducing the size of coverage-based test sets," in *Proceedings of the 12th International Conference on Testing Computer Software*, 1995, pp. 111–123.

[27] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur, "Effect of test set minimization on fault detection effectiveness," *Softw, Pract. Exper*, vol. 28, no. 4, pp. 347–369, 1998.

[28] P. McMinn, "IGUANA: Input generation using automated novel algorithms. A plug and play research tool," Department of Computer Science, University of Sheffield, Tech. Rep. CS-07-14, 2007.

[29] J. E. Baker, "Reducing bias and inefficienry in the selection algorithm," in *Proceedings of the 2nd International Conference on Genetic Algorithms and their Applications*, 1987, pp. 14–21.

[30] D. Whitley, "The GENITOR algorithm and selection pressure: Why rank-based allocation," in *Proc. of the Third Int. Conf. on Genetic Algorithms*, 1989, pp. 116–121.

[31] H. Mühlenbein and D. Schlierkamp-Voosen, "Predictive models for the breeder genetic algorithm: I. continuous parameter optimization," *Evolutionary Computation*, vol. 1, no. 1, pp. 25–49, 1993.

[32] E. J. Weyuker, "Program schemas with semantic restrictions," Ph.D. dissertation, Rutgers University, June 1977.

[33] M. Harder, J. Mellen, and M. D. Ernst, "Improving test suites via operational abstraction," in *Proceedings of the 25th International Conference on Software Engineering*, 2003, pp. 60–71.

[34] J. Black, E. Melachrinoudis, and D. R. Kaeli, "Bi-criteria models for all-uses test suite reduction," in *ICSE*, 2004, pp. 106–115.

[35] S. Yoo and M. Harman, "Pareto efficient multi-objective test case selection," in *ISSTA*, 2007, pp. 140–150.

[36] A. Leitner, M. Oriol, A. Zeller, I. Ciupa, and B. Meyer, "Efficient unit test case minimization," in *ASE*, 2007, pp. 417–420.

[37] B. Jones, H.-H. Sthamer, and D. Eyres, "Automatic structural testing using genetic algorithms," *The Software Engineering Journal*, vol. 11, pp. 299–306, 1996.

[38] M. Roper, "Computer aided software testing using genetic algorithms," in *10th International Software Quality Week*, San Francisco, USA, 1997.

[39] A. Watkins, "A tool for the automatic generation of test data using genetic algorithms," in *Proceedings of the Software Quality Conference '95*, 1995, pp. 300–309.

[40] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Trans. Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, 2002.