Open access • Journal Article • DOI:10.1109/TSE.2011.18

Input Domain Reduction through Irrelevant Variable Removal and Its Effect on Local, Global, and Hybrid Search-Based Structural Test Data Generation

Phil McMinn, Mark Harman, Kiran Lakhotia, Youssef Hassoun ...+1 more authors

Institutions: University of Sheffield, University College London, King's College London

Published on: 01 Mar 2012 - IEEE Transactions on Software Engineering (IEEE)

Topics: Beam search, Random search, Test data generation, Fitness function and Automatic test pattern generation

Input Domain Reduction through Irrelevant Variable Removal and its Effect on Local, Global and Hybrid Search-Based Structural Test Data Generation

Phil McMinn,
University of Sheffield,
Regent Court,
211 Portobello,
Sheffield, S1 4DP, UK.

Mark Harman, Kiran Lakhotia,
CREST,
University College London,
Gower Street, London,
WC1E 6BT, UK.

Youssef Hassoun,
King's College London,
Strand, London,
WC2R 2LS, UK.

Joachim Wegener,
Berner & Mattner Systemtechnik GmbH,
Gutenbergstr. 15,
D-10587 Berlin,
Germany.

Abstract

Search-Based Test Data Generation reformulates testing goals as fitness functions so that test input generation can be automated by some chosen search-based optimization algorithm. The optimization algorithm searches the space of potential inputs, seeking those that are 'fit for purpose', guided by the fitness function. The search space of potential inputs can be very large, even for very small systems under test. Its size is, of course, a key determining factor affecting the performance of any search-based approach. However, despite the large volume of work on Search-Based Software Testing, the literature contains little that concerns the performance impact of search space reduction. This paper proposes a static dependence analysis derived from program slicing that can be used to support search space reduction. The paper presents both a theoretical and empirical analysis of the application of this approach to open source and industrial production code. The results provide evidence to support the claim that input domain reduction has a significant effect on the performance of local, global and hybrid search, while a purely random search is unaffected.

Keywords. Search-Based Software Testing, Evolutionary Testing, Automated test data generation, input domain reduction.

1 Introduction

Software testing is an important but currently expensive activity. The expense of testing derives largely from the human costs of designing good test cases and examining their output. This has led to an enduring interest in methods to automate both construction of good test input and the determination of output correctness. The latter problem is known as the oracle problem and it has been the subject of much work in its own right. This paper is concerned with the former problem of automated test input construction.

The last decade has witnessed a considerable increase in research in the area of applying search-based optimization methods to this problem. Evidence for this increase in activity comes from the publication data presented in Figure 1.

This area of research has come to be known as Search-Based Software Testing (SBST) [2, 3, 33], an instance of Search-Based Software Engineering (SBSE) [17, 22, 40]. The flexibility of the approach lends itself to many testing problems, including structural [26, 32, 36, 38, 48, 52], functional [11], temporal [49] and integration [10] testing.

Although Search-Based Software Testing has received a great deal of attention, there has been almost no work [19], investigating the relationship between search space size and search performance – the ability

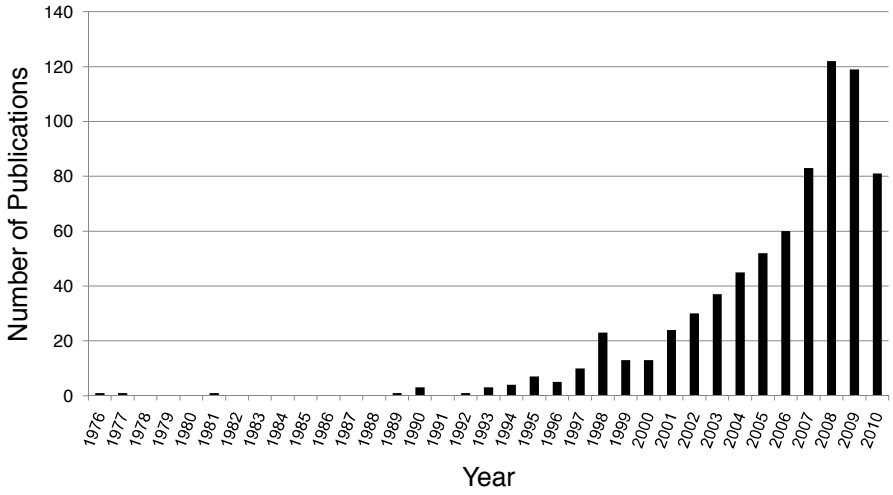


Figure 1: The growth trend of publications on Search-Based Software Testing. The publication number for 2010 is based on only partially complete data at the time of writing and so the figure reveals a dramatic growth in publications overall. Despite a rapid growth trend, there has been very little work on search space reduction strategies.

(Source: the SEBASE publications repository, `http://www.sebase.org/sbse/publications`)

of a search technique to find test data and the effort expended to do so. This paper seeks to address this problem. It presents both theoretical and empirical results concerning the impact of search space reduction on performance. The Search-Based Software Testing problem studied herein is the search for test data for full branch coverage of a program. This test goal was chosen because it has proved to be the most widely studied in the existing literature [22], thereby making the paper's findings more widely applicable.

The key observation underpinning this work is that only *some* input variables affect whether a branch will be executed. Others can statically be determined to have no effect on the branch in question and can, therefore, be safely removed from the search space. The removal of any such irrelevant input variables removes dimensions from the search space that would otherwise have been unnecessarily explored.

The paper presents a theoretical analysis of the potential impact of irrelevant input variable removal on Random Search, and three meta-heuristic methods guided by a fitness function; namely

1. A Hill Climbing algorithm. This is an optimization technique based on local search. The version used in this paper is the well–known Alternating Variable Method, due to Korel [26].

2. An Evolutionary Testing algorithm. This is an optimization technique based on global search. The version used in this paper is an implementation based on the work done by DaimlerChrysler, which has also been widely studied [48].

3. A Memetic Algorithm. This is a hybrid optimization technique based on a combination of global and local search. The version implemented for this paper is similar to that used by Wang and Jeng [47]. The use of Memetic Algorithms in Search-Based Software Testing is comparatively less well studied, but is relevant to the work reported here because it completes the overall picture; covering local, global and combined approaches to search-based optimization.

The theoretical analysis proves that there should be no relationship between irrelevant input variable removal and Random Search, while performance improvements should be possible for the three meta-heuristic searches. The theoretical analysis only shows the possibility of improvements for guided search. In order to assess the likely impact on real systems, an empirical study is required. The paper presents the results of a detailed, large–scale empirical study on both open source and real world production code for all three of these approaches: local, global and hybrid search. The results of the empirical study reveal that the theoretically–predicted improvement on search performance is statistically significant in many cases, including when the time cost of performing the upfront analysis for search space reduction is taken into account.

More specifically, the primary contributions of this paper are as follows:

1. The paper presents a theoretical analysis of irrelevant input variable removal on each of the algorithms studied in the paper. The predictions of this analysis are:
   (a) For Random Search: No significant change will be observed. The empirical results should bear out this prediction.
   (b) For Hill Climbing: Irrelevant input variable removal may cause a reduction in unfruitful manipulations of irrelevant input variables. An empirical study is required to determine the size of the effect in practice.
   (c) For Evolutionary Testing: Irrelevant input variable removal may also cause a reduction in unfruitful manipulations of irrelevant input variables. These manipulations are due to the mutation operator used by Evolutionary Testing. However, this positive effect might be counteracted by the increased disruptive effect on useful recombinations of input vectors produced by the crossover operator. Therefore, an empirical study is required to determine whether there is an effect and, if so, how large it could be.
   (d) Since Memetic Algorithm combines features of both Hill Climbing and Evolutionary Testing, there is also potential for improvement and this also requires empirical investigation.
2. To answer the empirical questions raised by the theoretical study, the paper performs a large–scale empirical study using open source programs and industrial production code. The results do indeed show that irrelevant input variable removal has no effect on Random Search, but it does have a positive effect on the performance of fitness–guided search algorithms. Specifically, the study finds that:
   (a) The success rate of finding inputs is significantly improved in several cases when the search operates in a reduced space. This is also true when the practical time cost of performing the up-front static analysis for search space reduction is taken into consideration, by allowing test data searches in unreduced search spaces to run for considerably longer.
   (b) Search space reduction results in a significantly higher quantity of additional branches to be serendipitously executed during the search for one specific target branch. This is because input variables deemed irrelevant for the target branch can be freely assigned to random values in the course of fitness evaluation.

The rest of this paper is organized as follows. Section 2 introduces Search-Based Software Testing, giving a detailed overview of the fitness function used and the search algorithms studied in this paper. Section 3 introduces the method for search space reduction; the removal of irrelevant input variables for the coverage of individual branches. Section 4 provides a theoretical analysis of the impact of irrelevant input variable removal on each of the search techniques. Section 5 introduces the empirical study and the research questions under investigation. The results obtained are presented, along with their corresponding analysis. Any empirical study such as this inherently involves potential threats to validity, and the section concludes with a discussion of these and how they were handled. Section 6 presents related work, while Section 7 concludes.

2 Search-based structural test data generation

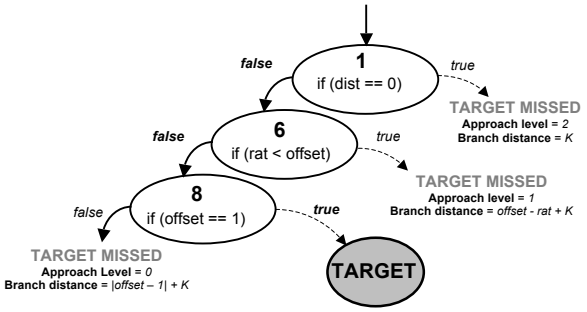
Search-Based Test Data Generation uses optimization techniques, such as Genetic Algorithms, to automatically generate test data. This paper focuses on structural testing, in particular branch coverage. In order to cover a particular branch in a unit under test, the goal of the search is to find an input vector for a function that drives the path of execution down the branch of interest. The search space is formed from the input domain of the function under test.

2.1 Basic concepts

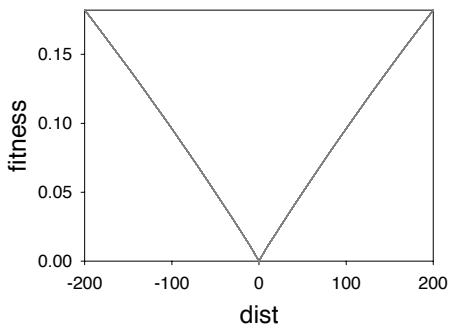
Let $I = (i_1, i_2, ...i_{len})$ be a vector of the input variables of the function under test. The domain D_{i_n} of the input variable i_n is the set of all values that i_n can hold, $1 \leq n \leq len; len = card(I)$. The **input domain** of the function under test, therefore, is a cross product of the domains of each of the individual input variables: $D = D_{i_1} \times D_{i_2} ... \times D_{i_{len}}$. An **input** to the function **i** is a specific element of the function's input domain, that is, $\mathbf{i} \in D$.

```
     double gradient_calc_radial_factor(double dist,
                                        double offset,
                                        double x,
                                        double y) {
         double r, rat;

(1)      if (dist == 0.0) {
(2)          rat = 0.0;
         } else {
(3)          offset = offset / 100.0;
(4)          r   = sqrt (x * x + y * y);
(5)          rat = r / dist;

(6)          if (rat < offset) {
(7)              rat = 0.0;
(8)          } else if (offset == 1.0) {
(9)              rat = (rat >= 1.0) ? 1.0 : 0.0;
                 // ...
```

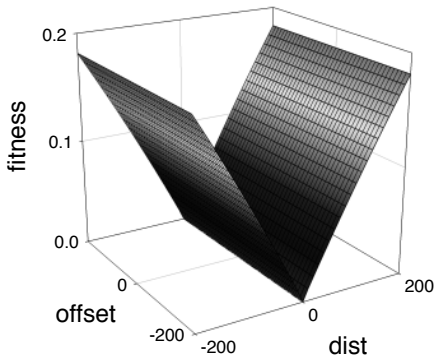
(a) *Code snippet*



(b) *Fitness calculation for the true branch from node 8. K is a constant added to the branch distance when the undesired alternate branch is taken*



(c) *Fitness landscape for true branch from node 1, plotted for the* `dist` *input variable only*



(d) *Fitness landscape for true branch from node 1, plotted for the* `dist` *and* `offset` *input variables only*

Figure 2: Code from the `gimp` open source graphics package and corresponding fitness analysis

2.2 Fitness function

Meta-heuristic optimization techniques require a numerical formulation of the test goal, from which a 'fitness function' is formed. The purpose of the fitness function is to guide the search into promising, unevaluated areas of a potentially vast input domain, in order to find required test data.

For branch coverage, each branch is taken as the focus of a separate test data search. The **fitness function** is a function $fit(t, \mathbf{i}) \rightarrow \mathbb{R}$, that takes a structural target t and individual input $\mathbf{i}$, and returns a real number that scores how close the input was to executing the required branch. This calculation has two components [48], the so-called *approach level* and the *branch distance*.

The approach level assesses the path taken by the input with respect to the target branch by counting the target's control dependencies that were not executed by the path. For structured programs, the approach level reflects the number of unpenetrated levels of nesting levels surrounding the target. For example, in Figure 2b, if the target is the true branch from node 8, the approach level is 2 if the execution path misses the target by taking the true branch at node 1. If the false branch is taken at node 1, but the target is then missed because the true branch was taken at node 6, the approach level is 1, and so on.

When execution of a test case diverges from the target branch, the second component, the *branch distance*, expresses how close an input came to satisfying the condition of the predicate at which control flow for the test case went 'wrong'; that is, how close the input was to descending to the next approach level. For example, for the coverage of the true branch from node 8 in Figure 2, the branch distance is computed using the formula $|\texttt{offset}-1| + K$, where K is a constant added when the undesired, alternate branch is taken. The closer `offset` is to 1.0, the 'closer' the desired true branch is to being taken. A different branch distance formula is applied depending on the type of relational predicate. In the case of relational equals, this is $|a - b| + K$. For a full list of branch distance formulae for different relational predicate types, see Tracey *et al.* [45].

The complete fitness value is computed by normalizing the branch distance and adding it to the approach level:

$$fit(t, \mathbf{i}) = approach\_level(t, \mathbf{i}) + normalize(branch\_distance(t, \mathbf{i}))$$

Since the maximum branch distance is generally not known, the standard approach to normalization cannot be applied [6]; instead the following formula is used:

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

A *fitness landscape* is a useful visualization of the surface of the fitness landscape. Figures 2c plots the fitness landscape for the coverage of the true branch from node 1 in the `gimp` example, for the variable `dist` only. The landscape shows how the value of `dist` can be minimized to zero for coverage of the branch. Figures 2d adds the variable `offset` to the plot, giving a 3-dimensional surface, and showing how changes to the `offset` variable have no effect on fitness.

The following sections provide a detailed overview along with the parameters used for the search techniques in this paper, in order to facilitate replication of the empirical study which appears in Section 5.

2.3 Random Search

Random Search does not involve a fitness function to guide the optimization process, and so is technically not a 'meta-heuristic' search technique. However, Random Search has been shown to be a surprisingly effective way of generating test data [15], and as such makes a useful baseline comparison to more complex and computationally intensive methods.

It is possible to cover many structural targets using random search, because there are often several input vectors that can be selected that are good enough to execute most of the structures of a program. For example, the false branch from node 1 is easily covered in Figure 2, because it is executed by all input vectors apart from those for which `dist` is zero. However, the chances of executing the alternative true branch at random however are significantly lower; for such test targets, more intelligent techniques are required.

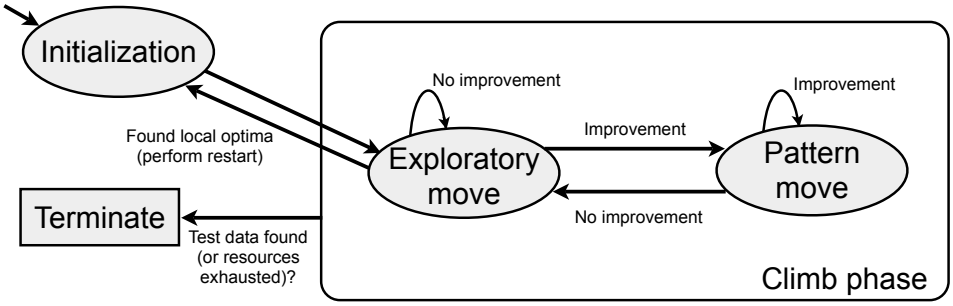


Figure 3: The main steps of a Hill Climbing algorithm using the Alternating Variable Method

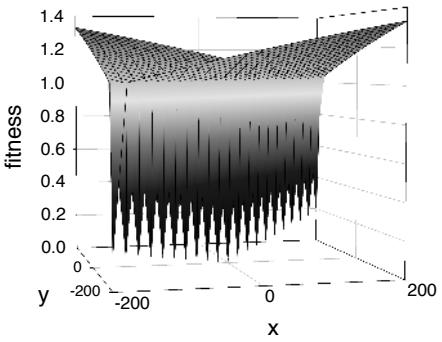
2.4 Hill Climbing

Hill Climbing is a meta-heuristic search technique that seeks to improve one candidate solution by exploring its neighbouring search space. Hill Climbing involves two stages; a simple *initialization* stage, where an initial candidate solution is chosen from the search space at random. Assuming the randomly-chosen solution does not satisfy the search criteria, the *climb* phase then commences, where solutions neighbouring the current solution are evaluated (Figure 3). If one of these neighbouring solutions offers an improved fitness, it replaces the current solution. The neighbourhood of this new solution is explored, and so on, until no neighbouring solutions can be found that improve the fitness of the current solution. This progressive pattern of moving to points in the search space with improved fitness values is likened to climbing hills on the surface of the fitness landscape, despite the fact the fitness function is actually minimized for search-based structural test data generation; and the optimization process figuratively descending rather than ascending into local optima.

The implementation of the Hill Climbing algorithm for this paper is based on the 'Alternating Variable Method' first introduced for test data generation by Korel [26]. This method explores the neighbourhood of each individual input variable in the input vector in turn. If changes in the values of the input variable do not result in an increased fitness, the search considers the next input variable, and so on; recommencing from the first input variable if necessary, until test data is found or no further improvements can be made to the current input vector.

Consider the example shown in Figure 2. Suppose the target is the true branch from node 8, and the initial random input is $(1, 50, 1, 1)$. The input reaches node 8 but takes the false branch. The Alternating Variable Method begins to perform 'exploratory moves' on each input variable by inducing a small increase, followed by a small decrease. Suppose changes of ± 0.1 are made to each double variable in the example of Figure 2. The moves made around the `dist` variable are 0.9 and 1.1. However, no improvement is made; `offset` is no closer to becoming 1. The search continues on to consider the next variable in the input vector, `offset`. An increased value of 50.1 moves `offset` closer to the value of 1 required at node 8 via the division statement at node 3. At this point, the search makes accelerated 'pattern' moves in the direction of improvement. In this paper, the value of the x^{th} move $move_x$ made in the direction of improvement, $dir \in \{-1, 1\}$ is computed using $move_x = s^x \cdot 10^{-prec_n} \cdot dir$, where s is the *repeat base* ($s = 2$ for experiments in this paper), and $prec_n$ is the number of decimal places defining the precision of exploratory moves to be made to the n^{th} input variable. Successive moves are made for `offset` until it overshoots the required value of 100, and the new value generated represents a decrease in fitness. At this point, exploratory moves are recommenced in order to establish a new direction. The search continues in this fashion until the test data is found, or the current input vector cannot be improved because local moves do not offer an improved fitness. The latter case is a common problem for local search techniques - the tendency to converge on a sub-optimal solution. This may be, for example, at the peak of locally optimal 'hills' (*i.e.* in the trough of pits where fitness is being minimized); or along ridges or plateaux in the fitness landscape, where there is no variation in fitness value. An example of this can be seen for the coverage of the target branch in Figure 4. As soon as the search moves to the bottom of one of the locally optimal spikes, it is effectively 'stuck', as there is no input vector in the direct vicinity with a better fitness value. When this happens, the search is restarted at another randomly chosen point in the search space. The number of restarts is potentially unlimited and only restricted by the limit on the overall budget allowance of fitness evaluations permitted.

```
void local_optima(double x, double y) {
    if (x == y) {
        if (x == 0) {
            // target
        }
    }
    // ...
}
```



(a) *Code snippet containing a nested target* (b) *Corresponding fitness landscape*

Figure 4: Code snippet with a target whose fitness landscape contains several local optima

2.5 Evolutionary Testing

Because of the problem of local search techniques like Hill Climbing becoming trapped in local optima, many authors have considered global search techniques, most notably Genetic Algorithms [31, 32, 34, 38, 48]. This has given rise to the so-called 'Evolutionary Testing' approach.

For Evolutionary Testing, input vectors compete with one another according to their ability to satisfy the test objective. The aim of Evolutionary Testing is that, as the algorithm iterates through increasingly fit generations of candidate test cases, it steadily and automatically approaches achievement of the test goal through optimization of the fitness function.

The Genetic Algorithm used for Evolutionary Testing in this paper is based on the approach described by Wegener *et al.* [48]. Genetic Algorithms differ from local search techniques in that they maintain a population of candidate solutions, also referred to as 'individuals', rather than just one solution. The population is therefore capable of sampling many points in the search space at once, and as a result is more robust to entrapment in local optima.

The *initialization* phase of a Genetic Algorithm involves constructing the initial population with randomly generated individuals. The Wegener model splits the overall population of 300 individuals into six competing subpopulations, which begin with 50 individuals each. The Genetic Algorithm then enters a loop involving stages referred to as *fitness evaluation*, *selection*, *crossover*, *mutation* and *reinsertion* (Figure 5). The Genetic Algorithm used in Evolutionary Testing also undergoes a further stage of *competition and migration* amongst the subpopulations.

For selection, the Wegener model uses linear ranking [51], which sorts the population into fitness order, assigning a ranked fitness to each individual based on a selection pressure value $Z = 1.7$. Ranked fitnesses are allocated in such a way that the current best individual in a population has a fitness of Z, the median individual a fitness of 1.0 and the worst candidate solution a fitness of $2 - Z$. Once individuals have been assigned a fitness, a selection operator is applied to the population. The method used is stochastic universal sampling [8], where the probability of an individual being selected is proportionate to its fitness value. This type of selection favours 'fitter' individuals. Weaker individuals retain a selection chance, but with a relatively smaller probability. Individuals are then removed two-by-two at random from the selection pool for recombination. A discrete recombination [37] strategy is used to produce new 'offspring'. Discrete recombination operates such that offspring individuals receive 'genes' (*i.e.* input variable values) from either parent with an equal probability.

The next stage in the Genetic Algorithm is to apply a mutation operation to the offspring, based on the Breeder Genetic Algorithm [37]. Mutation is applied with a probability p_m of $1/len$, where len is the length of the input vector. The mutation operator applies a different mutation step size $10^{-pop}, 1 \leq pop \leq 6$ for each of the six subpopulations. A mutation range r is defined for each input parameter by the product of pop and the domain size of the parameter. The 'mutated' value of an input parameter can thus be computed as $v' = v \pm r \cdot \delta$. Addition or subtraction is chosen with an equal

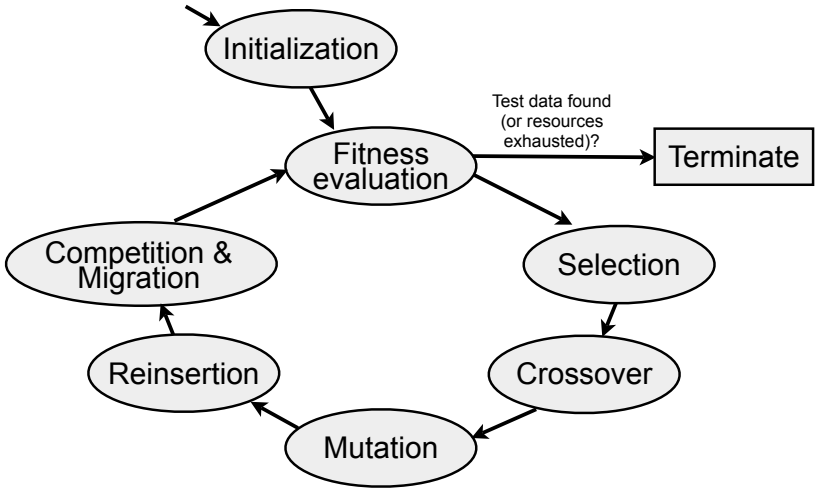


Figure 5: The main steps of a Genetic Algorithm

probability. The value of δ is defined to be $\sum_{y=0}^{15} \alpha_y \cdot 2^{-y}$, where each α_y is 1 with a probability of 1/16 else 0. If a mutated value is outside the allowed bounds of a variable, its value is set to either the minimum or maximum value for that variable.

An elitist reinsertion strategy is used in the following *reinsertion* stage. The top 10% of the current generation is retained and the remaining individuals discarded and replaced by the best offspring.

In the final stage of the loop, *competition and migration*, subpopulations compete with one another for a slice of individuals. A progress value, *prog*, is computed for each population at the end of a generation. This value is obtained using the formula $0.9 \cdot prog + 0.1 \cdot rank$. The average fitness $rank$ for a population is obtained by linearly ranking its individuals as well as the populations amongst themselves (again with $Z = 1.7$). After every 4 generations, the populations are ranked according to their progress value and a new slice of the overall population is computed for each, with weaker subpopulations transferring individuals to stronger ones. However, no subpopulation can lose its last 5 individuals, preventing it from dying out. Finally, a general migration of individuals takes place after every 20^{th} generation, where subpopulations randomly exchange 10% of their individuals with one another.

2.6 Memetic Algorithms

Memetic Algorithms combine the best features of both global and local search: robustness to local optima through the *diversification* of Genetic Algorithms with the fine-tuning *intensification* power of Hill Climbing. The Memetic Algorithm employed in this paper is essentially a Genetic Algorithm with a stage of Hill Climbing at the end of each generation to improve each individual of the population as much as possible. It was first proposed by Wang and Jeng [47].

Because Hill Climbing is used to intensify the search on an explicit sub-region of the search space, Memetic Algorithms tend to require smaller population sizes than Genetic Algorithms. For experiments in this paper, a flat population size of 20 is employed, without division into subpopulations as for Evolutionary Testing. Thus there is no competition or migration stage; however, the selection, crossover and reinsertion mechanisms are the same as those used for Evolutionary Testing. The Breeder Genetic Algorithm mutation operator is not used; since it is intended to work with the subpopulations to produce a process akin to intensification in a Genetic Algorithm. Instead, uniform mutation is employed at the usual mutation rate of $p_m = 1/len$. Uniform mutation overwrites an input variable value with a new randomly-generated value from its domain, thus encouraging greater search diversification. The Alternating Variable Method is applied to improve each input vector at the end of each generation, but without restarts. That is, as soon as a local optima is hit, the intensification phase for that individual terminates.

3 Irrelevant input variable removal

The test data search space for the coverage of an individual branch of the function under test is the input domain of that function. However it is possible that not every input variable will be responsible for determining whether every branch in the function under test will be executed or not. For example, in Figure 2, only the input variable `dist` is relevant for the predicate at node 1. Therefore, when attempting to cover branches from this node, search effort on the values of the other variables (`offset`, `x` and `y`) is wasted, since these variables cannot influence coverage of the branch. Thus removing *irrelevant* input variables from the test data search, and only concentrating on input variables that are *relevant* for the current branch of interest may improve the performance of the search process. Relevant and irrelevant input variables are defined as follows.

Definition 1. *Relevant input variable.*
A *relevant input variable* is an input variable $i_{rel} \in I$ that is capable of influencing whether a particular structural target t will be executed or not. That is, there exists an $\mathbf{i} \in D$ where t is not executed, for which there exists an $\mathbf{i}' \in D$ where $\mathbf{i}'$ is a copy of $\mathbf{i}$ but for the value of i_{rel} in $\mathbf{i}$ having been changed to some other $v \in D_{i_{rel}}$, where t is executed with $\mathbf{i}'$ as the input.

Definition 2. *Irrelevant input variable.*
An *irrelevant input variable* is an input variable $i_{irrel} \in I$ that is not capable of influencing whether a particular structural target t will be executed. That is, for all $\mathbf{i} \in D$ where t is not executed, there does not exist an $\mathbf{i}' \in D$ such that $\mathbf{i}'$ is a copy of $\mathbf{i}$ but for the value of i_{irrel} in $\mathbf{i}$ having been changed to some other $v \in D_{i_{irrel}}$, for which t is executed with $\mathbf{i}'$ as the input.

In general, the classification of an input variable as relevant or irrelevant for a particular target is an undecidable problem, since the target structure may appear in a program after a looping construct, for which termination is unknown. However it is possible to obtain a conservative estimate of irrelevancy using static data flow techniques. That is, fewer variables may be deemed to be irrelevant than is actually the case but, as a result, several variables may be removed from the search process.

3.1 Static analysis for removal of irrelevant input variables with CodeSurfer

For experiments in this paper, the implementation of variable dependence analysis was constructed on top of the CodeSurfer commercial program analysis tool from Grammatech Inc. The dependence analysis module[1], 'wraps' the CodeSurfer slicing functionality to produce variable dependence information in a similar format to that used by the Vada dependence analysis tool [18]. The decision to use the commercial product CodeSurfer, rather than the research prototype, was motivated by the need for scalability and robustness. CodeSurfer offers scalable dependence analysis, based on the System Dependence Graph (SDG) of Horwitz, Reps and Binkley [24] and offers various pointer analysis implementations, such as Steensgaard's [44] and Andersen's [4] scalable (but flow and context insensitive) points-to analysis. The default global pointer analysis algorithm provided by CodeSurfer is that due to Andersen and this is the one used in our study.

Of course, like any static analysis, the results of CodeSurfer are conservative approximations to true dependence, which is uncomputable. This means that the set of variables returned by the variable dependence wrapper module are a superset of the relevant variables. This is not a problem for the application to input domain reduction because the analysis is safe. It guarantees that any variables identified as irrelevant truly are irrelevant and, therefore, it is safe to remove them from the input vector when searching for input to traverse a chosen target branch. The choice of points–to analysis is not the only aspect of the dependence analysis that is a safe approximation. The results of the dependence analysis are also affected by the manner in which composite data types such as `struct`s and arrays are handled. In CodeSurfer, a variable type containing a `struct` can be expanded to give a more precise analysis of dependence. In essence, each field of the `struct` is treated as a separate variable in its own right and dependences that arise through one field do not become merged (or collapsed) into those that flow through the other fields. The wrapper uses this approach to avoid the imprecision that would arise when `struct`s become collapsed into a singleton variable [9]. However, for array types, there is no such expansion possible because the index type is a computed field and it is not possible to easily distinguish a dependence flowing through one index from those flowing through the others. There has been work

[1] In order to facilitate replication, the code of the dependence analysis module is available from `http://www.dcs.kcl.ac.uk/staff/youssef/software/wrapper-0.1/`.

on dependence analysis that can handle the complex constraints on array indices, in order to separate out dependence flowing through individual array elements [27]. However, this work has yet to find its way into industrial strength scalable dependence analysis tools such as CodeSurfer upon which this work relies. For this reason, our results for the improvements that can be achieved by domain reduction, can be thought of as a lower bound on what could be achieved with a more fine grained analysis.

The GrammaTech tool CodeSurfer is provided with a Scheme–based scripting language [16]. The scripting language extends standard Scheme with additional data types for computing the dependence-graph representation of C programs as well as the program's individual structures. This provides a programming level interface to the underlying implementation of the System Dependence Graph. Operations on the SDG are built into the extended language to express program elements and their dependencies. Scripts use this API to inspect C programs according to their structure.

The implementation used to calculate variable dependencies is based on slicing functions provided by the tool's scripting language. The correspondence between the problem of slicing and that of variable dependence analysis is described in more detail elsewhere [18]. This correspondence is used to manipulate the results of slicing, using the scripting language, in order to compute a conservative and safe approximation to the relevant input variable set.

To provide variable dependence information for branch coverage, conditional statements such as if-statements and loops, are identified. CodeSurfer provides function calls in Scheme to return all vertices in the SDG and to select from these, those of a certain type (for example predicate nodes, as required for branch coverage).

CodeSurfer also provides a set of Scheme functions for computing the backward slice of a node. The result of a backward slice is a set of vertices from the SDG that are contained in the slice. From the computed slice node set, all relevant variables can be derived. These are the variables that affect the computation of the program point at the given line. Input variables (global and formal parameters) can be then filtered out from this set. For each branch that forms a target for the Search-Based Test Data Generation, the CodeSurfer wrapper module provides a set of relevant variables (global variables and formal parameters that may affect the outcome of the predicate evaluation). This set is guaranteed to contain the true relevant variables. The search for test data can then operate on a reduced input vector to find inputs to execute the current branch of interest.

4 Theoretical analysis

This section presents a theory of the impact of irrelevant input variable removal on each of the search techniques discussed and applied in this paper. The theoretical analysis performed in this section demonstrates that:

1. Irrelevant input variable removal has no mean effect on Random Search.
2. Irrelevant input variable removal has the potential to improve the efficiency of the meta-heuristic searches considered; *i.e.* Hill Climbing, Evolutionary Testing and the Memetic Algorithm.

The analysis starts from first principles; that is by discussing the question of what factors contribute to the difficulty of a search-based test data generation problem. The impact of irrelevant input variable removal on these factors is then studied, and from this the potential effect on each of the individual search techniques is derived. The analysis assumes a search target of one specific branch, and that other branches executed serendipitously whilst searching for the target branch are ignored.

4.1 Problem difficulty for search-based test data generation

A common metric for measuring the effort expended by a search is the number of trial solutions (fitness evaluations) that the search had to consider in order to reach a final solution.

This paper contends that there are two main factors that contribute to the amount of effort a search needs to expend in order for search-based methods to find test data for a program structure. These are as follows:

1. **Ratio of domain size to target-executing inputs**

   The higher the ratio of domain size to target-executing inputs (or more simply the 'domain-to-execution ratio'), the 'scarcer' the existence of test data for its coverage, and the more effort the optimization technique is likely to have to expend in order to find an input vector that executes it.

Take for example the program of Figure 2, and the coverage of the true branch from node 1. For sake of argument, suppose each floating point variable has the range -499.9 to 500, with a precision of 0.1, and thus an individual domain size of 10000. The true branch is only executed by one specific value of `dist` along with any value of the other three input variables, giving a high domain-to-execution ratio of 10000^4:10000^3 or 10000:1. The domain-to-execution ratio of the alternative false branch, however, is extremely low; there is only one input from the domain size of 10000^4 that *does not* execute it, resulting in test data that is very easy to find.

Domain-to-execution ratio is related to the concept of domain-to-range ratio, the ratio of the cardinality of the set of all inputs to the cardinality of the set of all outputs for a program, identified by Voas [46] as a general source of poor testability in programs.

2. **Fitness landscape**.

   The shape of the fitness landscape is also a factor that affects the progress of a search. If the fitness landscape is free of plateaux and local optima, providing smooth guidance to the location of the desired inputs, the search will be relatively easy; as with the landscape of Figure 2d. However, if the search problem has a difficult landscape, that contains several local optima (for example as with Figure 4) or is largely flat, then the search may be either misled or offered little guidance at all.

It can be theorised from the above, that test data search problems with a high domain-to-execution ratio and difficult landscapes will be problematic for search-based test data generation techniques. The paper now investigates whether, from a theoretical stand-point, irrelevant input variable removal can have an impact on these attributes and whether it is capable of making test data generation problems *easier*. These results are then applied to each of the individual search techniques studied in this paper.

4.2 Impact of irrelevant input variable removal on search problem difficulty

4.2.1 Ratio of domain size to target-executing inputs

It transpires that irrelevant input variable removal has no effect on the domain-to-execution ratio for a target, *i.e.* irrelevant input variable removal has no impact on the first attribute of problem difficulty for search-based test data generation:

Proposition 1. *Irrelevant input variable removal has no impact on the domain-to-execution ratio for a target.*

Proof. The domain-to-execution ratio *der* for a target can be expressed as

$$der = card(D) : card(E)$$

where $card(D)$ is the cardinality of the input domain of the function under test, and $card(E)$ is the size of the set of target-executing inputs.
Input variables can be classified as either *relevant* or *irrelevant* to the target. Thus D can be split into two independent sets; the set of relevant input variables, and the set of irrelevant input variables. Let D_{rel} equal the cross product of the domains of the relevant variables, and D_{irrel} the cross product of the domains of the irrelevant variables. In accordance with this, D can be expressed as

$$D = D_{rel} \times D_{irrel}$$

and it follows that

$$card(D) = card(D_{rel}) \times card(D_{irrel})$$

In a similar fashion, the set E may itself be split into two sets of 'partial' input vectors:

$$E = E_{rel} \times E_{irrel}$$

where E_{rel} is the set of target-covering partial input vectors made up of relevant variables only, and E_{irrel} is the set of target-covering partial input vectors made from irrelevant variables.
By definition, in order to cover the target, specific values are required of relevant variables; while any value may be chosen from the domain of an irrelevant variable. Therefore it follows that $card(E_{irrel}) = card(D_{irrel})$, and that correspondingly

$$card(E) = card(E_{rel}) \times card(D_{irrel})$$

and the domain-to-execution ratio for a target can be re-expressed as

$$der = (card(D_{rel}) \times card(D_{irrel})) : (card(E_{rel}) \times card(D_{irrel}))$$

Since $card(D_{irrel})$ appears on both sides of the ratio, it can be cancelled, thus

$$\begin{aligned} der &= (card(D_{rel}) \times card(D_{irrel})) : (card(E_{rel}) \times card(D_{irrel})) \\ &= card(D_{rel}) : card(E_{rel}) \end{aligned}$$

That is, the domain-to-execution ratio of a target is not actually affected by irrelevant variables.

□

The above result can be demonstrated with an example. Take again the program of Figure 2, and the coverage of the true branch from node 1; the domain-to-execution ratio of which is 10000^4:10000^3 or 10000:1. Removing irrelevant variables gives a domain size of 10000 and one target-executing input vector (*i.e.* `dist` equal to 0), and despite irrelevant variable removal, the same domain-to-execution ratio of 10000:1.

4.2.2 Fitness landscape

It can be shown that irrelevant input variables identified by static analysis can have no effect on fitness values, and this has an interesting impact on the overall fitness landscape.

Proposition 2. *Statically-identified irrelevant input variables cannot have an effect on the fitness value for a target branch.*

Proof. The fitness function for covering an individual branch relies on structural control dependency information, and the values of variables appearing in conditions in the branch's control dependent nodes. This set of variables cannot be influenced by irrelevant input variables identified through static analysis. If they were, some irrelevant variable may exist capable of influencing whether the branch is executed or not. This however is contrary to the definition of an irrelevant input variable (Definition 2).

□

As a consequence of this, fitness invariant lines exist in the fitness landscape for the target's fitness function in each dimension representing an irrelevant variable revealed by static analysis. These dimensions of fitness-invariance are of no help in guiding the search. This observation can be seen in Figure 2d, where the irrelevant variable `offset` is responsible for a constant fitness for each value of `dist`. When `offset` is not considered, a useless dimension is removed from the fitness landscape, which does not give any guidance to the search (Figure 2c). The removal of these useless dimensions has the potential to improve the efficiency of a meta-heuristic search, albeit in different ways. Thus, the ramifications of these sections are now investigated in the light of the individual search techniques that are the focus of this paper.

Note that the proof only extends to the set of variables identified as irrelevant by static analysis, a subset of all variables that may be irrelevant in practice for a branch. Irrelevant variables not identified by static analysis, for example, include those that influence variables appearing in the predicates of infeasible branches. As the branch is infeasible, such variables cannot influence actual execution of the branch, and therefore are irrelevant as per Definition 2). However, as discussed in Section 3.1, automating the discovery of this type of irrelevancy is a generally undecidable problem, and is not considered further here.

4.3 Impact on Random Search

Random Search simply attempts to cover the target structure by constructing input vectors at random, and is not guided by a fitness function. Therefore, problem difficulty for Random Search centres around the *probability* of executing a target. The lower the probability of covering a target, a greater number of trials will be required to cover it, in expectation. The reverse is true for targets with a high probability of execution. However, irrelevant input variable removal has no impact on the probability of executing a target, and thus, in expectation, irrelevant input variable removal has no impact on Random Search.

Proposition 3. *Irrelevant input variable removal has no effect, on average, for Random Search.*

Proof. The probability of executing a target at random is simply another way of looking at a target's domain-to-execution ratio: for a domain-to-execution ratio of $card(D)$:$card(E)$ for a target, the chance of executing that target with one randomly chosen input is $\frac{card(E)}{card(D)}$.

However, as has already been shown by Proposition 1, irrelevant input variable removal has no impact on domain-to-execution ratio, and irrelevant input variable removal will not affect the probability of coverage of a target at random. Because excluded input variables cannot affect whether the target is covered, any of their values can be selected; removing them reduces $card(D)$ and $card(E)$ in equal proportion, and the probability of executing the target is unchanged.

□

The variation in the generation of random numbers and/or the order in which they are applied means that, in practice, differences will be observed at the level of individual random searches when irrelevant variables are removed from consideration. However, since the probability of generating test data is the same, the mean effect will be of no change in the performance of Random Search.

4.4 Impact on Hill Climbing

The Hill Climbing algorithm was described in Section 2.4. The algorithm begins with a randomly generated starting point, and makes moves from this initial position towards the nearest point of locally optimal fitness. If a locally optimal point does not represent test data that covers the target, the algorithm restarts with a freshly generated random point. Thus there are two distinct parts to the algorithm, *initialization* followed by the *climb* phase.

Initialization

As per Proposition 3, the randomly-generated starting point is no more likely to cover the target if irrelevant variables are present or not. Furthermore, it can be shown that the randomly generated starting point will be no closer to the required test data in the input domain with irrelevant input variable removal than it would have been under normal circumstances. That is, it will be no easier for a search to find test data starting with randomly-generated solutions whether irrelevant variables have been removed or not.

Proposition 4. *Initial, randomly generated solutions will be no closer to required test data in the input domain following the removal of irrelevant input variables from the search process.*

Proof. The distance from a starting solution in the fitness landscape to the required test data can be found by finding the Euclidean distance between the initial input vector $\mathbf{init} = (init_1, init_2, ...init_n)$ and the nearest qualifying, target-executing input vector $\mathbf{qual} = (qual_1, qual_2, ...qual_{len})$, *i.e.*:

$$d(\mathbf{init}, \mathbf{qual}) = \sqrt{\sum_{n=1}^{len} (init_n - qual_n)^2}$$

Let $\mathbf{init_{rel}} = (init_{rel_1}, init_{rel_2}, ..., init_{rel_p})$ and $\mathbf{init_{irrel}} = (init_{irrel_1}, init_{irrel_2}, ..., init_{irrel_q})$ be partial vectors of **init** that respectively contain either relevant or irrelevant variables only for a target t. Likewise, let $\mathbf{qual_{rel}} = (qual_{rel_1}, qual_{rel_2}, ..., qual_{rel_p})$ and $\mathbf{qual_{irrel}} = (qual_{irrel_1}, qual_{irrel_2}, ..., qual_{irrel_q})$ be partial vectors of **qual** that respectively contain relevant or irrelevant variables pertaining to t. The Euclidean distance function can now be represented as:

$$d(\mathbf{init}, \mathbf{qual}) = \sqrt{\sum_{j=1}^{p} (init_{rel_j} - qual_{rel_j})^2 + \sum_{k=1}^{q} (init_{irrel_k} - qual_{irrel_k})^2}$$

By definition, any value of an irrelevant input variable is suitable for executing the target. Thus each initial irrelevant input variable value is a qualifying value, and

$$\forall k \in \{1, \ldots, q\}, (init_{irrel_k} - qual_{irrel_k})^2 = 0$$

Therefore

$$\sum_{k=1}^{q} (init_{irrel_k} - qual_{irrel_k})^2 = 0$$

Consequently

$$\begin{aligned} d(\mathbf{init}, \mathbf{qual}) &= \sqrt{\sum_{j=1}^{p} (init_{rel_j} - qual_{rel_j})^2 + \sum_{k=1}^{q} (init_{irrel_k} - qual_{irrel_k})^2} \\ &= \sqrt{\sum_{j=1}^{p} (init_{rel_j} - qual_{rel_j})^2} \end{aligned}$$

That is, the euclidean distance to the required test data is dependent on relevant variables only, *i.e.* randomly-generated test data will be no closer to the required test data in the input domain whether the search process removes irrelevant variables or not.

□

Climb Phase

A performance increase is possible for Hill Climbing as the search commences from its initial position. As per Proposition 2, changing the values of irrelevant variables will never yield a difference in fitness. Thus, the exclusion of irrelevant variables will avoid incremental and decremental exploratory moves occurring in the climb phase. From this we can conclude that for non-trivial structures (*i.e.* those requiring the test data search to enter the climb phase of the Hill Climbing algorithm as described in Section 2.4) irrelevant input variable removal has the potential to improve the efficiency of Hill Climbing. That is, test data may be found using fewer fitness evaluations.

Proposition 5. *Irrelevant input variable removal can result in test data being found in fewer fitness evaluations for a target with Hill Climbing than if all variables are considered*

Proof. The climb phase of the Hill Climbing algorithm defined in Section 2.4 involves taking each input variable in turn and performing incremental and decremental 'exploratory' moves until a variable is found that has an impact on the fitness.

Let I_{rel} denote the set of relevant input variables for some target t, and I_{irrel} the set of irrelevant variables. One 'cycle' through the input vector will therefore involve at least two moves per input variable, *i.e.* $card(I) \times 2$, or $(card(I_{rel}) + card(I_{irrel})) \times 2$. If irrelevant variables are removed from the search process however, this is reduced to $card(I_{rel}) \times 2$, *i.e.* a 'saving' of $card(I_{irrel}) \times 2$ evaluations per cycle.

Thus, for non-trivial structures that are not easily covered at random in the initialization stage of Hill Climbing (and require traversal of the input vector in the following climb phase), the removal of irrelevant variables can result in the performance of fewer fitness evaluations. □

The theory predicts that improvements in efficiency are possible, but does not predict exactly what the improvement will be. This is because it is impossible to predict how many times the Hill Climbing algorithm will cycle through the input vector, and thus how many fitness evaluations will be saved. The number of cycles performed is dependent on the starting point and the shape of the fitness landscape, which is different for each program structure. This information can only be known by actually working through the process of the search on a case-by-case basis. As such, although the theoretical

analysis suggests that Hill Climbing will be subject to an improved performance as a result of irrelevant input variable removal, the actual improvement in performance in practice is better assessed through an empirical treatment. For two branches of similar difficulty (*i.e.* requiring a similar number of fitness evaluations before irrelevant input variable removal) the theory predicts a greater efficiency improvement for the branch that has more irrelevant input variables removed, as a greater number of fitness evaluations will be 'saved' per cycle of the input vector. The theory therefore suggests the existence of a relationship whereby the efficiency of Hill Climbing improves as a function of the number of irrelevant variables that are removed from the input domain.

4.5 Impact on Evolutionary Testing

The Evolutionary Testing algorithm was introduced and described in detail in Section 2.5. The effect of irrelevant input variable removal is analysed with respect to each stage of the algorithm.

Initialization

The initial phase of Evolutionary Testing involves the construction of 300 inputs at random for the first generation of the search. Thus for 'easy' branches (generally those with a probability of execution of $\frac{1}{300}$ or higher), there will be no change in performance for Evolutionary Testing. As proved in Proposition 4, these initial solutions will be no closer to the required solution in the input domain.

Selection

Selection is a function of fitness and does not take any other properties of candidate solutions into account. Thus, irrelevant input variable removal will not affect the operation of selection mechanisms.

Crossover

Evolutionary Testing performs discrete recombination. The choice of the two parents from which each individual element of the offspring's chromosome is taken is decided with a fixed probability of 0.5. Thus, irrelevant input variable removal will affect crossover in that it gives it slightly less work to do - without irrelevant input variable removal there are more variables to be inserted into child input vectors than are actually needed in practice. However, this overhead is small in terms of executing the overall algorithm and, more crucially, does not affect the number of fitness evaluations that will need to be performed.

Mutation

While irrelevant input variable removal has no impact on crossover in terms of fitness evaluations, it will affect the performance of the mutation operator.

Proposition 6. *Irrelevant input variable removal reduces the number of mutations that do not lead to a change in fitness of an input vector*

Proof. The mutation rate is set to $\frac{1}{card(I)}$, and thus, in expectation one variable will be mutated per input vector. If the variable mutated is an irrelevant variable, the mutation may as well not have been performed, as it cannot bring about a change in fitness.

When irrelevant input variables are present, the mutation operator is essentially working with a handicap; only $\frac{card(I_{rel})}{card(I_{rel})+card(I_{irrel})}$ of the mutations have the potential to affect fitness. When irrelevant variables are removed, mutation is concentrated on variables deemed to be relevant for covering the structure of interest. □

In the absence of a crossover operator, one would expect the increase in the number of effectual mutations to improve the performance of the search; fewer fitness evaluations would be performed on input vectors whose relevant variables are unchanged (*i.e.* there is no change in fitness value). Furthermore, a relationship is expected whereby performance should increase as a function of the number of irrelevant variables removed.

However, such a relationship needs to be confirmed by an empirical study. Although the mutation operator becomes more effective, the change of mutation rate on the relevant variables may not necessarily improve performance. For example, an increase may further disrupt useful recombinations brought about by the crossover operator.

Reinsertion, Competition and Migration

As with selection, the reinsertion, competition and migration mechanisms of Evolutionary Testing are functions of the fitness values of input vectors only, and as such no impact is expected of irrelevant input variable removal on these parts of the Evolutionary Testing algorithm.

4.6 Memetic Algorithm

Since the Memetic Algorithm is a combination of Evolutionary Testing and Hill Climbing, the same factors apply as discussed in the previous two sections. For trivial branches whose test data can be found on average in 20 trials or fewer (*i.e.* those usually found in the first generation by the Memetic Algorithm) there will be no impact. However, irrelevant input variable removal will increase the mutation rate as with Evolutionary Testing, and in the climb phase borrowed from the Hill Climbing algorithm. Irrelevant input variable removal will stop ineffectual moves being made involving variables that cannot affect whether the target will be executed or not.

4.7 Conclusions of the Theoretical Analysis

The theory shows that there will, on average, be no impact for Random Search. While the theory predicts a positive impact is possible for each of the meta-heuristic searches, it does not show exactly for all possible cases what the expected impact will be, for either specific cases or on 'average'. This is because search difficulty varies on a branch-by-branch basis, *i.e.* the domain-to-execution ratio for the branch and the shape of its fitness landscape. The true impact is more efficiently assessed by an empirical study, and this is the subject of the next section.

5 Empirical Study

The theoretical analysis raises several empirical questions which are answered in this section, using real world examples of industrial and open source test data generation problems.

5.1 Case Studies

The empirical study was performed on 636 branches, drawn from seven different case studies. Details of these case studies can be found in Table 1. Search-based test data generation requires the tester to specify information about the ranges of each input variable. In practice, this helps prevent the search from generating values for the unit that would not be possible in practice. In the empirical study, two different setups were used to produce two artificial domain sizes for each function, one larger and therefore 'harder' than the other, thus giving 1108 search problems available for study. As can be seen, approximate domain sizes ranged from 10^5 to 10^{712}, resulting in large search problems.

The search algorithms under study from the literature are capable of handling data of numerical type only (character, integer, double etc.). More recent work has demonstrated that this is not a limitation of the search-based approach [28]. However, in this study, dynamic data structures were fixed in size and shape where necessary for inputs to functions of the `space` case study. The size and shape of the data structures in question were fixed to the largest size required to prevent branches in the case study becoming artificially infeasible, as determined through manual tests..

The programs `f2` and `defroster` are industrial case studies provided by DaimlerChrysler, which are production code for engine and rear window defroster embedded control systems for which the source code was automatically generated from a design model.

To complement the industrial examples, five further case studies were selected. Different functions were selected from these case studies to allow for a broad range of styles including different types of input (for example primitive integers and non-trivial data structures involving pointers) and control structures (for example loops and the `switch` statement). The number of functions was chosen in order to keep the number of source lines of code studied large enough to allow for a rigorous empirical analysis, whilst keeping the study manageable to perform from a human perspective.

Of the five additional case studies, three were open source. Sixteen functions were selected from `gimp-2.2.4`, a graphics manipulation package. The program `space` comes from the European Space Agency and is available from the Software-artifact Infrastructure Repository [1, 14]. Nine functions were tested. The program `spice` is an open source general purpose analogue circuit simulator. Two

functions were tested, which were clipping routines for the graphical front-end. The library `tiff-3.8.2` manipulates images in the Tag Image File Format (TIFF). The functions tested comprised of routines for placing images on pages, and the building of 'overview' compressed sample images. Finally, the program `totinfo` was created by Siemens and, like the program `space`, is also drawn from the Software-artifact Infrastructure Repository. Three functions were tested.

5.2 Experimental setup and evaluation method

The test data generation experiments were performed sixty times using an identical list of sixty different random seeds for each combination of branch and search method; Random Search, Hill Climbing, Evolutionary Testing and the Memetic Algorithm, as described in Section 2; with and without irrelevant input variable removal. If test data was not found to cover a branch after 100,000 fitness evaluations, the search was terminated. However, in order to answer some research questions, further experiments were performed where searches in unreduced spaces were allowed to run for a certain amount of time past this limit. The precise configuration used in each case are discussed along with the results answering each research question in Section 5.4.

For the irrelevant input variable removal version of the experiments, values still have to be provided for the irrelevant inputs, even though they are not subject to the search process. These values were assigned randomly.

The success or failure of each search was recorded, along with the number of test data evaluations required to find the test data, if the search was successful. From this the *success rate* of each branch can be calculated - the percentage of the 60 runs for which test data to execute the branch was found. The success rate measure expresses the *effectiveness* of the search. Using the number of runs for which a particular search was successful in finding test data, effectiveness was contrasted for statistical significance with and without irrelevant input variable removal using Fisher's Exact test at a confidence level of 99.9%. All p-values were computed using the R statistical package [39]. For each successful run, the *average evaluations* statistic (the average number of test data evaluations) indicates how *efficient* the search has been in finding the required test data. The fewer test data evaluations required, the better and more efficient the search performed. Any increase in effectiveness or efficiency therefore contributes to an increase in the *performance* of a search algorithm.

Experiments were performed on a laptop running Windows 7 with 2GHz 64bit dual core processor, and 2GB of RAM.

5.3 Research Questions

The research questions to be addressed by the empirical study are as follows:

RQ1. Scope for irrelevant input variable removal

For the test subjects considered, what proportion of input variables are irrelevant for each branch? If each branch is dependent on most of the input variables, then the search may not become much more effective or efficient following irrelevant input variable removal.

RQ2. Effect on search performance

The theory predicts that irrelevant input variable removal will have no impact on Random Search, while the theory predicts that for non-trivial branches, irrelevant input variable removal will have a positive effect on the performance of Hill Climbing, and will increase the number of 'effectual' mutations for Evolutionary Testing. The Memetic Algorithm combines aspects of both Hill Climbing and Evolutionary Testing, and therefore there should also be improvements in performance for the Memetic Algorithm. Does the empirical study confirm that this is the case in practice? The theory suggests the existence of a relationship whereby the efficiency of the search increases with the number of irrelevant variables. Is this relationship observed in practice?

RQ3. Practical considerations

Does irrelevant input variable removal make for a more effective test data generation process overall when the time overhead of performing the static analysis is taken into account?

Table 1: Details of the case studies investigated

Case Study / Function	Physical Source Lines of Code	Number of of Branches	Number of of Input Variables	Domain Size (10^x) Setup 1	Setup 2
defroster					
Defroster_main		56	20	24	96
Total	250	56			
f2					
F2		24	17	54	81
Total	418	24			
gimp					
gimp_hsv_to_rgb		16	4	21	37
gimp_hsv_to_rgb_int		16	3	7	12
gimp_hsv_to_rgb4		16	3	16	27
gimp_hwb_to_rgb		18	3	17	27
gimp_rgb_to_hsl		14	4	20	37
gimp_rgb_to_hsl_int		14	3	7	12
gimp_rgb_to_hsv		10	4	20	37
gimp_rgb_to_hsv4		18	3	7	12
gimp_rgb_to_hsv_int		14	3	7	12
gradient_calc_bilinear_factor		6	6	34	51
gradient_calc_conical_sym_factor		8	6	31	49
gradient_calc_conical_asym_factor		6	6	31	49
gradient_calc_linear_factor		8	6	31	49
gradient_calc_radial_factor		6	4	21	33
gradient_calc_spiral_factor		8	7	37	58
gradient_calc_square_factor		6	4	21	33
Total	867	184			
space					
addscan		32	236	519	712
fixgramp		8	9	23	32
fixport		6	65	125	182
fixselem		8	65	125	182
fixsgrel		68	236	524	712
fixsgrid		22	44	101	120
gnodfind		4	24	70	89
seqrotrg		32	68	206	264
sgrpha2n		16	161	451	614
Total	2210	196			
spice					
cliparc		64	9	44	59
clip_to_circle		42	7	23	30
Total	269	106			
tiff					
TIFF_SetSample		14	3	10	13
PlaceImage		16	11	38	59
Total	182	30			
totinfo					
gser		6	2	10	14
InfoTbl		30	4	19	23
LGamma		4	1	5	7
Total	319	40			
Grand Total	4515	636			

RQ4. Relative impact

Which search is the all-round 'best' at test data generation following irrelevant input variable removal, and which search exhibits the largest improvement in effectiveness?

RQ5. Serendipitous coverage

During the search for test data for a particular branch, other branches may also be covered serendipitously. Does irrelevant input variable removal lead to the search covering a greater number of branches by chance?

5.4 Results

This section discusses the results and answers the research questions posed in Section 5.3.

RQ1. Scope for irrelevant input variable removal

Figure 6 plots the number of relevant and irrelevant variables for each branch of each case study listed in Table 1. The branches are sorted along the horizontal axis by their containing function's input vector size, and then by the number of irrelevant variables. The figure clearly shows a lot of scope for irrelevant input variable removal, particularly for the `space` case study, with functions made up of a large number of inputs, made up of linked lists of records grouped together using the `struct` keyword. Most of the variables contained within these records are not relevant for executing many of the branches contained within each individual function analyzed.

RQ2. Effect on search performance

In this research question, the performance of each search is considered with and without irrelevant input variable removal with the termination criterion set at 100,000 fitness evaluations if test data is not found.

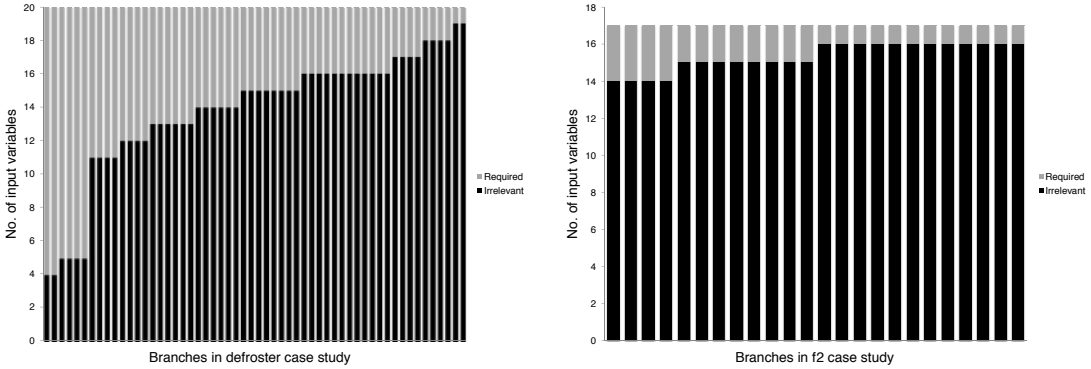
The results provide evidence to support the claim that Random Search is unaffected by the removal of irrelevant input variables from the search space. Although the results of the empirical study for Random Search exhibit variation before and after the application of search space reduction, no obvious relationship was found between the number of variables removed and the performance of Random Search. The variation is merely a result of a difference in the order in which random numbers are assigned to inputs from the same seed; with irrelevant input variable removal the relevant variables are assigned first, with the further irrelevant variables assigned next in order to form a complete input vector to execution the function containing the target. No branches were found for which there was a statistically significant difference in the number of successful test data searches before and after reduction. There is, therefore, no evidence to suggest that the effort required for Random Search is reduced by removing irrelevant variables.

The lack of improvement for Random Search contrasts strongly with those for Hill Climbing and Evolutionary Testing, as can be seen by comparing Figure 7 for Random Search with Figure 8 for Hill Climbing, Evolutionary Testing and the Memetic Algorithm. These scatter plots show the difference in success rate (the percentage of the sixty runs for which a particular branch was successfully covered) and average numbers of test data evaluations for each branch recorded before and after the removal of irrelevant variables from the search. Branches are ordered along the horizontal axis according to the number of variables removed from the input vector as a result of irrelevant variable analysis. Some points are plotted for the same horizontal axis value because some branches share the same number of irrelevant variables. The vertical axis value of each point plotted is found by subtracting the success rate or average evaluations value for searches performed *without* input domain reduction from the corresponding statistic for searches performed *with* irrelevant input variable removal. Therefore, a positive difference in success rate denotes an increase in success rate, and is indicated by a point on the positive vertical axis scale for the scatter plot of Figure 7a. Conversely, a poorer success rate is marked on the negative vertical axis scale. For the average number of evaluations, a positive difference means that the search required more test data evaluations to find a solution, and thus a point on the positive vertical axis corresponds to poorer efficiency as a result of irrelevant input variable removal, whereas a negative vertical axis mark is the result of fewer average evaluations and an improved efficiency. To enable a fair comparison, the average number of evaluations plot only includes branches that were covered with a 100% success rate before and after the removal of irrelevant variables. The plots of Figure 7 depict stochastic variation but reveal no trend to suggest an increased (or even decreased) performance of Random Search as the percentage of irrelevant variables increases and the search space becomes smaller.

The conclusion for Random Search, therefore, is the empirical lack of relationship between irrelevant input variable removal and its performance, validating the proof for Random Search in Section 5.3.

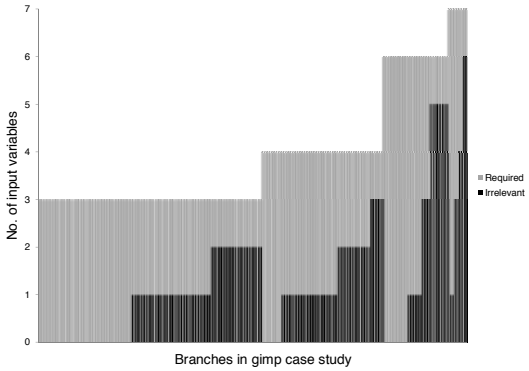
With respect to Hill Climbing, 41 branch test data searches were found for which Hill Climbing was significantly more effective with irrelevant input variable removal using Fisher's Exact Test when comparing the number of times the branch was successfully executed with and without reduction, and with the termination criterion set at 100,000 fitness evaluations for both types of search. No branch test data searches were found to be significantly worse. 9 of the 41 branches are not covered by Random Search with a 100% success rate, and are thus non-trivial. These branches are recorded in Table 2. 1 of the 9 branches is never covered without irrelevant input variable removal at the 100,000 evaluations termination criterion. This branch belongs to the `space` case study and appears in the `addscan` function.

Figure 8 (parts *a* and *b*) shows the performance difference with Hill Climbing against the number of variables removed for all branches considered in this experiment. Although there are a large number of

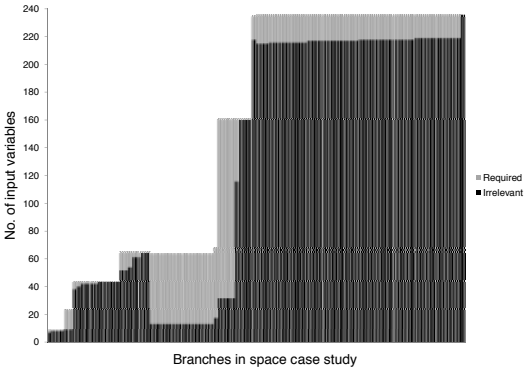


(a) `defroster` *case study*

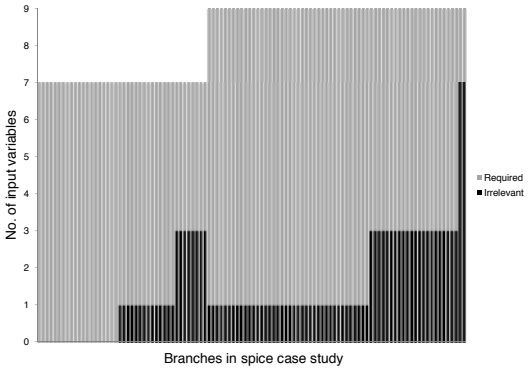
(b) `f2` *case study*



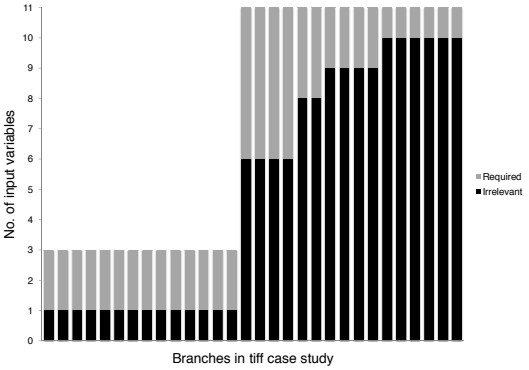
(c) `gimp` *case study*



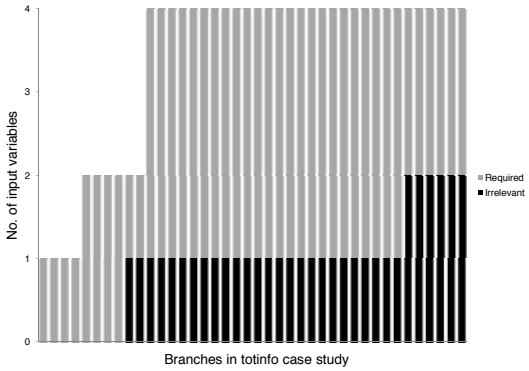
(d) `space` *case study*



(e) `spice` *case study*

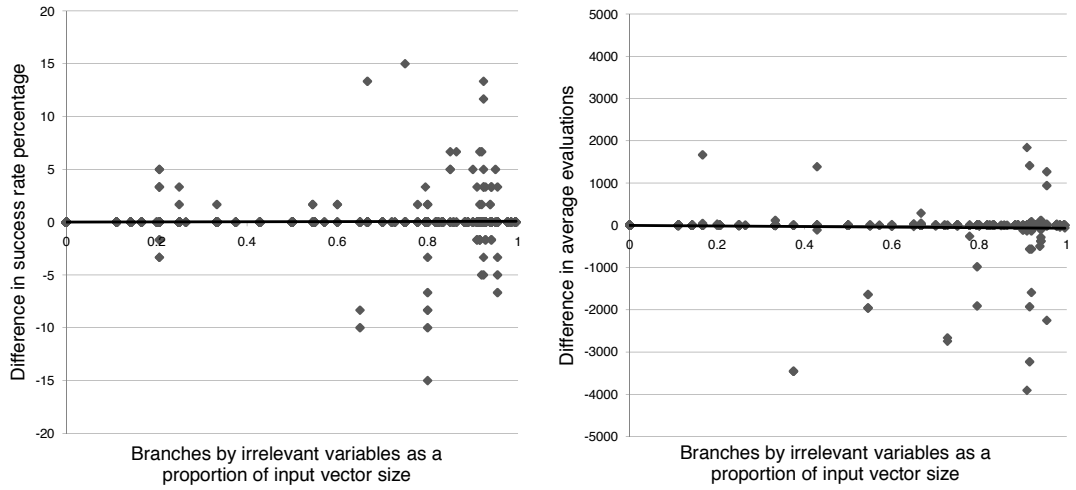


(f) `tiff` *case study*



(g) `totinfo` *case study*

Figure 6: A summary of required and irrelevant variables for each branch from each case study and function. The branches are sorted along the horizontal axis by the no. of input variables of their containing function and then by the number of irrelevant variables for the individual branch



(a) *Success rate* (b) *Average evaluations*

Figure 7: The effect of irrelevant input variable removal on Random Search. The scatter plots show changes in success rate and average test data evaluations as a result of removing irrelevant variables from the search process. No trend exists suggesting a change in performance of Random Search as more variables are removed and the search space becomes smaller

cases of improvement, in terms of success rate and average number of test data evaluations, some branches exhibited no improvement at all, simply because test data was found easily by Random Search. Search space reduction is superfluous in such instances. Due to stochastic variation, success rate decreased by 3% for one branch, and in a handful of cases, the average number of evaluations increased with irrelevant input variable removal. None of these branches were significantly worse with irrelevant input variable removal. The trend lines on the scatter plots strongly indicate the existence of a relationship whereby increasing the number of variables removed from the search improves the performance for Hill Climbing, in terms of higher success rate (improved effectiveness) and lower average number of test data evaluations (improved efficiency).

The conclusion for Hill Climbing is that the results provide evidence to support the hypothesis that removing irrelevant input variables from the search has a positive impact on the rate of success of test data generation.

For Evolutionary Testing, 26 branches were found for which irrelevant input variable removal from the search resulted in a statistically significant improvement in success of test data generation with the 100,000 fitness evaluations termination criterion. There were no branches for which the test data search became significantly less successful using irrelevant input variable removal. 25 of these branches were not covered by Random Search with a 100% success rate and appear in Table 2.

Figure 8 (parts *c* and *d*) shows the performance difference with Evolutionary Testing, against the number of irrelevant input variables removed for all branches considered in this experiment. The scatter plots show the existence of a trend: as the number of irrelevant input variables increases, and the search space is made smaller, the effectiveness and efficiency improves. This is observed through an increasing success rate and decreasing number of average test data evaluations required to cover the branches. For a handful of branches, the success rate became worse with irrelevant input variable removal, a result of stochastic variation and a difference that was not statistically significant. In further cases, as with Hill Climbing, the average number of evaluations increased with irrelevant input variable removal. Again, this appears to be due to simple random variation. None of the branches were significantly worse, with the difference being small or with only a very small number of variables actually removed from the search.

In conclusion for Evolutionary Testing, the results provide evidence to support the hypothesis that removing irrelevant variables from the search has a positive impact on Evolutionary Testing.

With respect to the Memetic Algorithm, statistical tests on the number of successful test data searches

revealed 29 branches became significantly more effective using irrelevant input variable removal with the 100,000 fitness evaluations termination criterion, with no branches becoming significantly worse. 9 of these branches were not covered by Random Search with a 100% success rate, and appear in Table 2.

The scatter plots for changes in success rate and average number of evaluations appear in Figure 8, parts *e* and *f*. As for Hill Climbing and Evolutionary Testing, there exists a trend (although less pronounced) whereby an increasing number of irrelevant variables improves efficiency and effectiveness.

Therefore, to conclude, the results again provide evidence that confirms the hypothesis that irrelevant input variable removal has a positive impact on the Memetic Algorithm also.

RQ3. Practical considerations

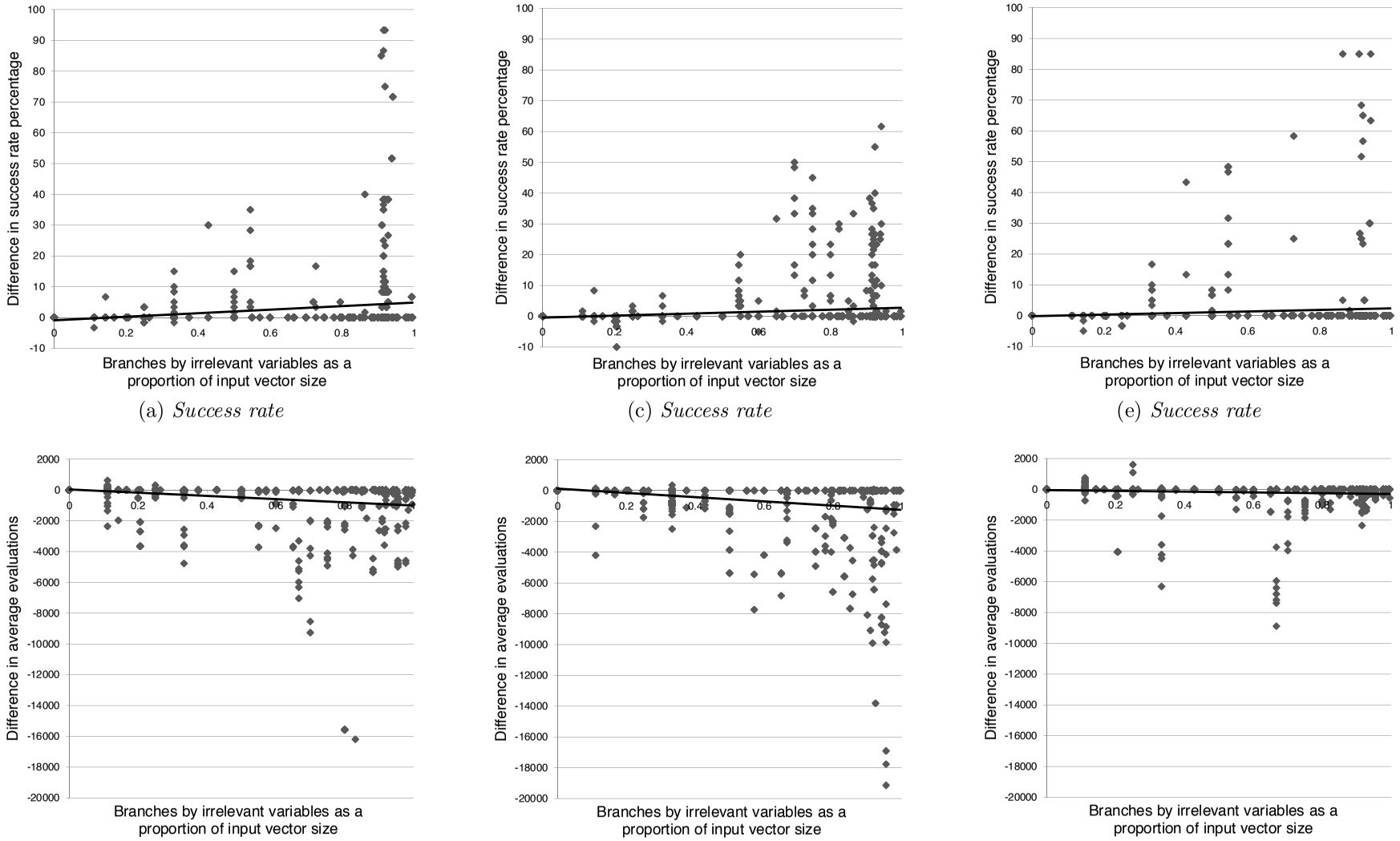
In the answer to RQ2, the analysis concentrated on a comparison with the performance of the search over 100,000 fitness evaluations for searches with and without irrelevant input variable removal. As detailed earlier in the paper, static analysis must be performed *a priori* on the source code of a test object in order to identify which variables are relevant and which are irrelevant. This research question aims to address the practical issue of whether it is worth doing the static analysis and searching for test data over the relevant variables only, or whether searches should just be run for more fitness evaluations in unreduced input domains, *i.e.* as many fitness evaluations as can be performed in the time taken by the static analysis for irrelevant input variable removal.

The times to perform static analysis are recorded in Table 3. The static analysis takes a C code file and determines the relevant and irrelevant variables for each function within it. The static analysis was performed 60 times in order to obtain an average wall clock time for each file, referred to as the 'file time'. In the case of `gimp`, the functions analyzed were split across two files, and as such the times for each file are recorded. The static analysis is performed once and used for each function and branch analyzed in the empirical study. Therefore, the table records two further times for each file analyzed. The 'function time' is the file time divided by the number of functions in it analyzed in the empirical study. The 'branch time' corresponds to the file time for a case study divided by the number of branches analyzed in that file.

Each search technique was re-performed for each branch without irrelevant input variable removal, except the process was not terminated after 100,000 fitness evaluations if test data was not found. Instead, the search was allowed to continue for the same amount of time that it took for the static analysis to be performed on the file containing that branch. This represents a worst case upper bound on the static analysis time required to perform irrelevant input variable removal, as it is equivalent to the static analysis being performed each time for each branch – in practice the information obtained would be re-used for other branches. However, even when the searches in the unreduced input domains were given the full amount of static analysis time, 41 cases were found where the search was found to be significantly more effective in finding test data when static analysis had been performed first. The significance test performed was Fisher's Exact Test on the number of successful searches, with and without irrelevant input variable removal. 23 of these cases are reported in Table 2; these cases correspond to branches not covered with a 100% success rate by Random Search up to the original 100,000 evaluations limit, and are thus deemed to be non-trivial. The cases include Hill Climbing, Evolutionary Testing and the Memetic Algorithm, and are drawn from the `defroster`, `f2` and `space` case studies. When the additional time allowed is tightened to the function time limit, a further 13 branches are found to be significant in favor of performing irrelevant input variable removal; and at the branch time level, a further 7, giving 43 cases in total.

Only 6 cases were found to be significant in favour of running the search longer and not performing irrelevant input variable removal. These cases all involved Evolutionary Testing with the `seqrotg` function of the `space` case study, and are also recorded in Table 2. For these branches, the conservative pointer analysis leads to more variables being included in the search than are strictly necessary, leading to an almost identical list of variables to search for test data even after static analysis has been performed. Since Evolutionary Testing fails to find test data within the 100,000 evaluations limit, it benefits significantly if the amount of time equivalent to static analysing the file in which they are contained is given over to extra fitness evaluations. However, it should be noted that these cases are only significant when the full file time is given as extra time for each branch. In practice, it would be performed at most once for the function and re-used for each branch. However the search is not significantly better when only extra function time or less is granted for additional fitness evaluations.

Of further interest are the 9 cases in which the search did not benefit at all from the additional 'file time' afforded to it. A further experiment was performed where the searches and branches in question



(a) *Success rate*

(b) *Average evaluations*

(c) *Success rate*

(d) *Average evaluations*

(e) *Success rate*

(f) *Average evaluations*

Hill Climbing **Evolutionary Testing** **Memetic Algorithm**

Figure 8: The effect of irrelevant input variable removal on meta-heuristic search. Success rate is the percentage of the 60 runs for which a branch was covered. A mark on the positive vertical axis denotes an improvement after irrelevant variables are removed. Average evaluations is the average number of fitness evaluations required to reach a solution. The fewer evaluations are required, the faster and therefore more efficient the search is. Therefore, a mark on the negative region of the vertical axis denotes an improvement. The number of evaluations required for each target branch is plotted as the average number of evaluations only when they were covered with a 100% success rate, both before and after the removal of irrelevant variables from the search. This is to ensure a fair comparison.

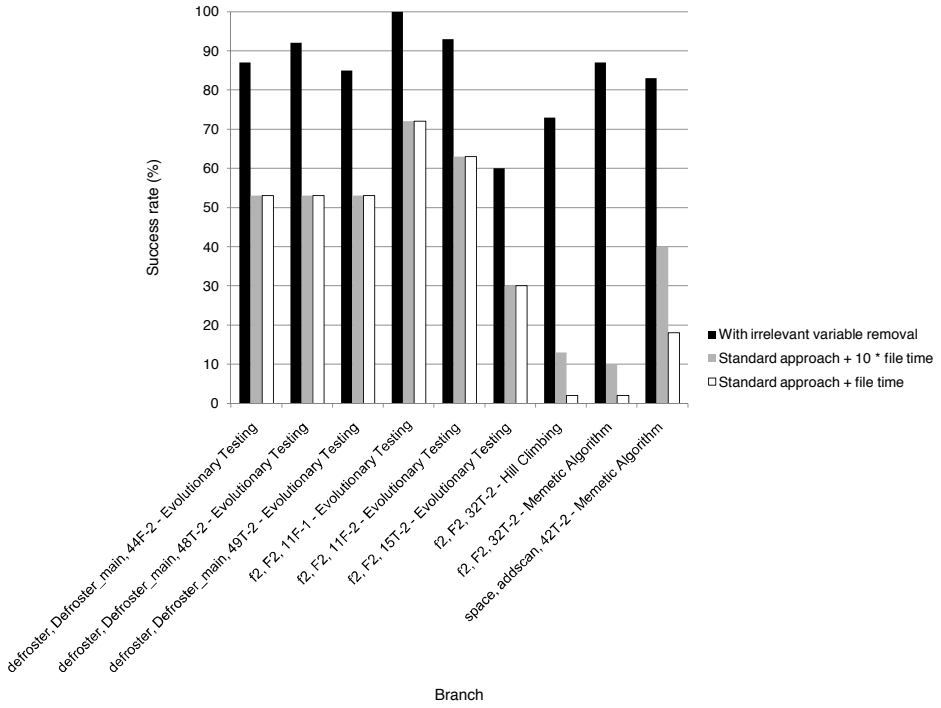


Figure 9: Success rate for cases of search technique and branch where the standard approach, without irrelevant input variable removal, is allowed to continue past the 100,000 fitness evaluations termination criterion for the amount of time equivalent to performing the static analysis on the file containing the branch 10 times

were run without irrelevant input variable removal for an additional amount of time equivalent to 10 times the 'file time'. The results can be seen in the bar chart of Figure 9. For Evolutionary Testing, there was no advance on the success rate without irrelevant input variable removal, even after the further amount of additional time, equivalent to 1.5 million fitness evaluations. In the failing search runs, the evolutionary process seems to converge on a local optimum that it cannot be shaken out of without the targeted mutations that occur as part of irrelevant input variable removal. On the other hand, the cases involving Hill Climbing and the Memetic Algorithm do result in a slight further improvement. In the case of Hill Climbing, this is only to be expected, as the irrelevant variables only contribute to a 'slowing down' in each cycle of the climb phase of the algorithm, given an identical set of starting points to a search where the irrelevant variables are removed. Still, the hill climb is more successful with irrelevant input variable removal than running the search without for an extremely long amount of time.

In conclusion, the evidence suggests that irrelevant input variable removal increases the reliability of test data generation even when the practical time costs of the static analysis are taken into account.

RQ4. Relative impact

Figure 10 depicts a bar chart of the success rate of each meta-heuristic search with and without irrelevant input variable removal over all branches in the empirical study. Success rate figures are also recorded for the standard approach, *i.e.* without irrelevant input variable removal, with the additional times introduced for practical consideration of the techniques using irrelevant input variable removal. The bar chart shows that the Memetic Algorithm is the most prolific technique at successfully finding test data with and without irrelevant input variable removal. Interestingly, the Memetic Algorithm without irrelevant input variable removal is approximately equivalent in performance terms to Evolutionary Testing with irrelevant input variable removal. All searches are more successful with irrelevant input variable removal, even when the standard approach for each branch is run for the 100,000 evaluations limit plus the time taken to statically analyze the entire file in which it is contained. Hill Climbing shows the biggest improvement when irrelevant input variable removal is applied compared to the standard approach terminated at 100,000 fitness evaluations. However when the search continues past this limit

Table 2: Branches for which test data searches are significantly more successful when irrelevant input variable removal is applied. Only *non-trivial* branches are reported in the table; branches that were not covered with a 100% success rate with Random Search. Success rate (SR) is reported for irrelevant input variable removal against the searches operating in unreduced search spaces with various termination criteria. Significant p-values at a confidence of 99.9% are given in bold for the alternative hypothesis comparing the number of successful standard approach searches against the number of successful searches for which irrelevant input variable removal had been applied as a pre-step. The values for 'file time', 'function time' and 'branch time' are reported for each case study in Table 3

Branch -setup	**Search**	**Irrelevant variable removal SR (%)**	**Unreduced search terminated at 100,000 evaluations + file time**		**Unreduced search terminated at 100,000 evaluations + function time**		**Unreduced search terminated at 100,000 evaluations + branch time**		**Unreduced search terminated at 100,000 evaluations**	
			SR (%)	**Sig. (less / greater)**	**SR (%)**	**Sig. (less / greater)**	**SR (%)**	**Sig. (less / greater)**	**SR (%)**	**Sig. (less / greater)**
defroster										
Defroster_main										
18F-2	Evolutionary Testing	78	45	**0.0002** / 1.0000	45	**0.0002** / 1.0000	43	**0.0000** / 1.0000	43	**0.0000** / 1.0000
22F-2	Evolutionary Testing	78	35	**0.0000** / 1.0000	35	**0.0000** / 1.0000	33	**0.0000** / 1.0000	33	**0.0000** / 1.0000
24F-2	Evolutionary Testing	57	13	**0.0000** / 1.0000	13	**0.0000** / 1.0000	8	**0.0000** / 1.0000	8	**0.0000** / 1.0000
24T-2	Evolutionary Testing	83	35	**0.0000** / 1.0000	35	**0.0000** / 1.0000	33	**0.0000** / 1.0000	33	**0.0000** / 1.0000
44F-2	Evolutionary Testing	87	53	**0.0000** / 1.0000	53	**0.0000** / 1.0000	53	**0.0000** / 1.0000	53	**0.0000** / 1.0000
48F-2	Evolutionary Testing	65	35	**0.0009** / 0.9998	35	**0.0009** / 0.9998	32	**0.0002** / 0.9999	32	**0.0002** / 0.9999
48T-2	Evolutionary Testing	92	53	**0.0000** / 1.0000	53	**0.0000** / 1.0000	53	**0.0000** / 1.0000	53	**0.0000** / 1.0000
49F-2	Evolutionary Testing	82	52	**0.0004** / 0.9999	52	**0.0004** / 0.9999	50	**0.0002** / 1.0000	50	**0.0002** / 1.0000
49T-2	Evolutionary Testing	85	53	**0.0002** / 1.0000	53	**0.0002** / 1.0000	53	**0.0002** / 1.0000	53	**0.0002** / 1.0000
60F-2	Evolutionary Testing	87	62	0.0016 / 0.9997	62	0.0016 / 0.9997	58	**0.0005** / 0.9999	58	**0.0005** / 0.9999
f2										
F2										
32T-2	Hill Climbing	73	2	**0.0000** / 1.0000	2	**0.0000** / 1.0000	2	**0.0000** / 1.0000	2	**0.0000** / 1.0000
32T-2	Memetic Algorithm	87	2	**0.0000** / 1.0000	2	**0.0000** / 1.0000	2	**0.0000** / 1.0000	2	**0.0000** / 1.0000
11F-1	Evolutionary Testing	100	72	**0.0000** / 1.0000	72	**0.0000** / 1.0000	72	**0.0000** / 1.0000	72	**0.0000** / 1.0000
11F-2	Evolutionary Testing	93	63	**0.0000** / 1.0000	63	**0.0000** / 1.0000	63	**0.0000** / 1.0000	63	**0.0000** / 1.0000
15T-2	Evolutionary Testing	60	30	**0.0008** / 0.9998	30	**0.0008** / 0.9998	30	**0.0008** / 0.9998	30	**0.0008** / 0.9998
space										
addscan										
42T-2	Hill Climbing	83	13	**0.0000** / 1.0000	8	**0.0000** / 1.0000	8	**0.0000** / 1.0000	8	**0.0000** / 1.0000
62F-2	Hill Climbing	35	2	**0.0000** / 1.0000	0	**0.0000** / 1.0000	0	**0.0000** / 1.0000	0	**0.0000** / 1.0000
42T-2	Memetic Algorithm	83	18	**0.0000** / 1.0000	18	**0.0000** / 1.0000	18	**0.0000** / 1.0000	18	**0.0000** / 1.0000
62F-2	Memetic Algorithm	28	5	**0.0005** / 0.9999	3	**0.0001** / 1.0000	3	**0.0001** / 1.0000	3	**0.0001** / 1.0000
26T-2	Evolutionary Testing	100	98	0.5000 / 1.0000	80	**0.0001** / 1.0000	72	**0.0000** / 1.0000	72	**0.0000** / 1.0000
33T-2	Evolutionary Testing	100	73	**0.0000** / 1.0000	65	**0.0000** / 1.0000	63	**0.0000** / 1.0000	63	**0.0000** / 1.0000
42T-2	Evolutionary Testing	47	22	0.0033 / 0.9991	13	**0.0000** / 1.0000	12	**0.0000** / 1.0000	12	**0.0000** / 1.0000
fixselem										
30T-2	Hill Climbing	100	100	1.0000 / 1.0000	75	**0.0000** / 1.0000	48	**0.0000** / 1.0000	48	**0.0000** / 1.0000
32F-2	Hill Climbing	100	100	1.0000 / 1.0000	73	**0.0000** / 1.0000	50	**0.0000** / 1.0000	48	**0.0000** / 1.0000
30T-2	Memetic Algorithm	100	100	1.0000 / 1.0000	83	**0.0006** / 1.0000	70	**0.0000** / 1.0000	70	**0.0000** / 1.0000
32F-2	Memetic Algorithm	100	100	1.0000 / 1.0000	83	**0.0006** / 1.0000	70	**0.0000** / 1.0000	70	**0.0000** / 1.0000
30T-2	Evolutionary Testing	97	77	0.0011 / 0.9999	73	**0.0003** / 1.0000	70	**0.0000** / 1.0000	70	**0.0000** / 1.0000
32F-2	Evolutionary Testing	97	77	0.0011 / 0.9999	73	**0.0003** / 1.0000	70	**0.0000** / 1.0000	70	**0.0000** / 1.0000
fixsgrel										
13F-1	Evolutionary Testing	100	98	0.5000 / 1.0000	87	0.0030 / 1.0000	83	**0.0006** / 1.0000	83	**0.0006** / 1.0000
13T-1	Evolutionary Testing	98	60	**0.0000** / 1.0000	52	**0.0000** / 1.0000	45	**0.0000** / 1.0000	43	**0.0000** / 1.0000
27F-1	Evolutionary Testing	100	100	1.0000 / 1.0000	85	0.0014 / 1.0000	73	**0.0000** / 1.0000	73	**0.0000** / 1.0000
27T-1	Evolutionary Testing	97	72	**0.0001** / 1.0000	63	**0.0000** / 1.0000	57	**0.0000** / 1.0000	57	**0.0000** / 1.0000
96F-1	Evolutionary Testing	100	92	0.0287 / 1.0000	78	**0.0000** / 1.0000	75	**0.0000** / 1.0000	75	**0.0000** / 1.0000
fixsgrid										
59F-2	Hill Climbing	88	17	**0.0000** / 1.0000	7	**0.0000** / 1.0000	3	**0.0000** / 1.0000	3	**0.0000** / 1.0000
71F-2	Hill Climbing	43	13	**0.0002** / 1.0000	7	**0.0000** / 1.0000	3	**0.0000** / 1.0000	3	**0.0000** / 1.0000
59F-2	Memetic Algorithm	100	93	0.0594 / 1.0000	30	**0.0000** / 1.0000	17	**0.0000** / 1.0000	15	**0.0000** / 1.0000
71F-2	Memetic Algorithm	100	87	0.0030 / 1.0000	23	**0.0000** / 1.0000	17	**0.0000** / 1.0000	15	**0.0000** / 1.0000
59F-2	Evolutionary Testing	72	43	0.0015 / 0.9996	33	**0.0000** / 1.0000	33	**0.0000** / 1.0000	33	**0.0000** / 1.0000
71F-2	Evolutionary Testing	67	42	0.0050 / 0.9984	33	**0.0002** / 0.9999	33	**0.0002** / 0.9999	33	**0.0002** / 0.9999
seqrotrg										
17T-1	Evolutionary Testing	52	95	1.0000 / **0.0000**	63	0.9304 / 0.1339	55	0.7084 / 0.4275	53	0.6426 / 0.5000
17T-2	Evolutionary Testing	52	93	1.0000 / **0.0000**	63	0.9304 / 0.1339	55	0.7084 / 0.4275	53	0.6426 / 0.5000
22T-1	Evolutionary Testing	52	93	1.0000 / **0.0000**	63	0.9304 / 0.1339	55	0.7084 / 0.4275	53	0.6426 / 0.5000
22T-2	Evolutionary Testing	52	93	1.0000 / **0.0000**	62	0.9015 / 0.1785	55	0.7084 / 0.4275	53	0.6426 / 0.5000
27F-1	Evolutionary Testing	77	100	1.0000 / **0.0000**	90	0.9872 / 0.0423	87	0.9514 / 0.1189	87	0.9514 / 0.1189
27F-2	Evolutionary Testing	77	100	1.0000 / **0.0000**	90	0.9872 / 0.0423	87	0.9514 / 0.1189	87	0.9514 / 0.1189
tiff										
PlaceImage										
16T-1	Hill Climbing	78	68	0.1510 / 0.9261	55	0.0057 / 0.9983	45	**0.0002** / 1.0000	43	**0.0000** / 1.0000
20T-2	Hill Climbing	40	28	0.1240 / 0.9384	20	0.0138 / 0.9955	12	**0.0003** / 0.9999	12	**0.0003** / 0.9999
16T-1	Memetic Algorithm	87	85	0.5000 / 0.6992	77	0.1189 / 0.9514	42	**0.0000** / 1.0000	40	**0.0000** / 1.0000
16T-2	Memetic Algorithm	53	55	0.6429 / 0.5000	45	0.2327 / 0.8634	23	**0.0006** / 0.9998	22	**0.0003** / 0.9999

Table 3: Static analysis times for each of the case studies. The static analysis works on a file-by-file basis, and as the functions tested with the `gimp` case study appear in two different files, timing figures are provided for each file. 'File time' is the time taken to analyze a file, 'function time' is the file time divided by the number of functions analyzed in that file, while branch time is the file time divided by the number of branches tested in the file

Case Study	File Time (ms)	Function time (ms) (number of functions)		Branch time (ms) (number of branches)	
defroster	4482	4482	(1)	80	(56)
f2	4741	4741	(1)	198	(24)
gimp - gimpcolorspace.c	14617	1624	(9)	107	(136)
gimp - gimpdrawableblend.c	11995	1714	(7)	250	(48)
space	42475	4719	(9)	217	(196)
spice	6458	3229	(2)	61	(106)
tiff	7361	3681	(2)	245	(30)
totinfo	7680	2560	(3)	192	(40)

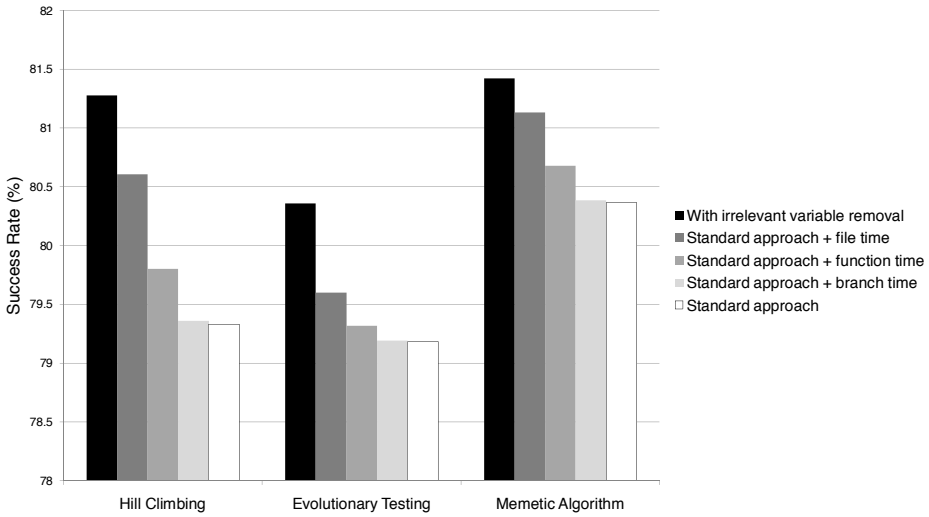


Figure 10: Overall success rate of each search technique, with and without irrelevant input variable removal, for all of the branches in all of the case studies considered

for the file time, Evolutionary Testing shows the largest improvement. The Memetic Algorithm exhibits the least improvement at all levels.

The difference in success rate levels is shown to be relatively moderate, varying between a few percentage points with each percentage corresponding to approximately 750 searches. However any technique that improves the reliability of test data generation can only be a good thing in practical terms, since the costs of machine generating test data are much less than if a human performed the job manually, further supporting the view that irrelevant input variable removal is a practically useful step to perform before test data generation using a meta-heuristic search technique.

RQ5. Serendipitous coverage

Of further interest in comparing search techniques with and without irrelevant input variable removal, is the level of 'serendipitous' coverage - the number of branches covered in the course of searching for test data for another specific branch. Although irrelevant variables are removed from the search process, the test object remains unchanged, and as such random values are assigned for the purposes of fitness evaluation. In a normal search process, including all variables, only certain variable values are changed at any one time. Thus, irrelevant input variable removal has the potential to cover more branches through unconstrained random search at the same time as concentrating on the search for test data for some other specific branch.

Serendipitous coverage was monitored in the experiments. Using the Wilcoxon Rank-Sum test, num-

bers of serendipitous branches covered where compared for significance with and without irrelevant input variable removal, for each branch with each meta-heuristic search technique. The results are depicted in the bar charts of Figure 11. For all case studies, more branches were covered serendipitously with irrelevant input variable removal than without it. In many cases, higher serendipitous coverage levels were recorded for a branch without irrelevant input variable removal merely because it had more fitness evaluations in which to do it (*i.e.* the search for the actual targeted branch was more inefficient). Thus the bar chart also records the number of branches for which the search covered a significantly higher number of other branches significantly, but did not also take a significantly larger number of fitness evaluations in which to do it, comparing numbers of fitness evaluations with and without irrelevant input variable removal using the Wilcoxon Rank-Sum test. Even after such branches are removed from consideration, there are still cases where the search without irrelevant input variable removal still covers significantly more other branches. In such cases it appears that constraining irrelevant variables may help serendipitous coverage of closely related branches executed with similar but not identical test data.

In terms of search techniques, it appears that Hill Climbing has the most to gain in terms of serendipitous coverage with respect to irrelevant input variable removal. This is because Hill Climbing tends to be poorer than Random Search at covering trivially easy branches with a high domain-to-execution ratio. Unlike Evolutionary Testing and the Memetic Algorithm, which begin with a certain quantity of randomly-generated input vectors, only the first input vector generated for Hill Climbing is random following initialisation and each restart, with each individual input variable manipulated in turn in a constrained manner. Allowing the irrelevant variables to take on random values for each fitness evaluation allows Hill Climbing to cover more other branches at random which it would not have done otherwise.

5.5 Threats to validity

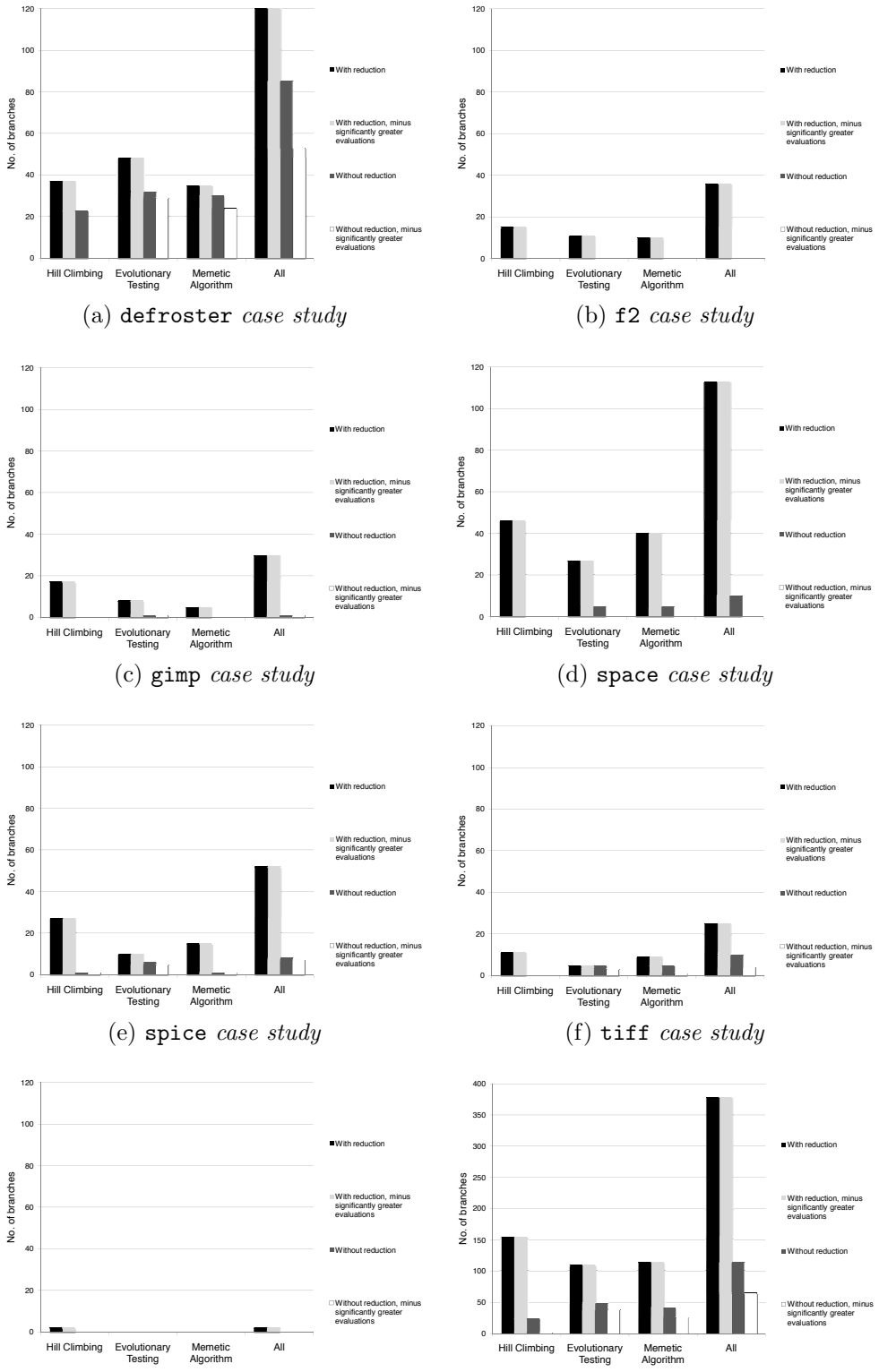
This section discusses the potential threats to validity with respect to the empirical study, and details how they were addressed. The hypotheses studied in this paper concerned relationships between irrelevant input variable removal and the performance of search algorithms employed for branch coverage. Therefore, one issue to address is the so-called *internal validity*, that is, to check whether there has been a bias in the experimental design that could affect the causal relationship under study.

In order to determine the appropriate size of the search space by eliminating contributions due to irrelevant variables, the CodeSurfer tool was scripted using Scheme functions. As these scripts represent research prototypes, a manual check was performed on results obtained to ensure that the tool had correctly identified the variables that could potentially affect the predicates of interest.

Another potential source of bias comes from the inherent stochastic behaviour of the meta-heuristic search algorithms under study. A common and reliable technique for overcoming this source of variability is to perform tests for statistical significance on a sufficiently large sample of result data. Such a test is required whenever one wishes to make the claim that one technique produces superior results to another. A set of results are obtained from a set of runs (essentially sampling from the population of random seeds). To show that one technique is more effective than another, Fisher's Exact Test for categorical data was used to compare the number of successful searches. The Wilcoxon rank-sum test was used to compare numbers of serendipitous branches covered. Both tests were applied with the confidence level set at 99.9%, to see if there is a statistical significant difference in the means of each set of results. Both tests are non-parametric, which was a deliberate choice in order to avoid making assumptions or having to perform additional analysis showing that the conditions for a parametric test have been met (*i.e.* normality of the sample means). Such additional analysis could introduce further possible sources of error into the study. The R package [39] was used to run the statistical tests.

Another source of bias comes from the selection of the programs to be studied. This impacts upon the *external validity* of the empirical study. That is, the extent to which it is possible to generalize from the results obtained. Naturally, it is impossible to sample a sufficiently large set of programs such that the full diversity of all possible programs could be captured. The rich and diverse nature of programs makes this an unrealistic goal. However, where possible, a variety of programming styles and sources have been used. The study draws upon code from real world programs, both from industrial production code and from open source. Furthermore, it should be noted that the number of different branches considered is 636, providing a relatively large pool of subjects from which to make observations.

Nonetheless, caution is required before making any claims as to whether these results would be observed on other programs, possibly from different sources and in different programming languages. The experiments studied search techniques commonly used in Search-Based Software Testing, complete with their usual parameter settings; but likewise, care should be taken in generalizing the results to their



(a) `defroster` *case study*

(b) `f2` *case study*

(c) `gimp` *case study*

(d) `space` *case study*

(e) `spice` *case study*

(f) `tiff` *case study*

(g) `totinfo` *case study*

(h) *All case studies*

Figure 11: Number of branches which were significant when comparing, with and without irrelevant input variable removal, the number of additional branches covered serendipitously in the same program while searching for test data for the target branch

respective families (local search, Genetic Algorithms and Memetic Algorithms) as a whole.

As with all such experimental software engineering, further experiments are required in order to replicate the results contained here. However, the results show that cases do indeed exist where there is a statistically significant relationship between search space reduction and improved performance of search algorithms for test data generation.

6 Related work

Although the search space reduction question has been asked and answered for other search problems [12, 43] there has been little work addressing the issue for Search-Based Test Data Generation. The question is a highly timely one, as the last ten years have experienced an explosion in work in the area, resulting in a survey by McMinn in 2004 [33]. The field began in 1976 with the work of Miller and Spooner [36], who applied numerical maximization techniques to generate floating point test data for paths. Korel was the first to apply the technique referred to as Hill Climbing in this paper [26], whilst Xanthakis *et al.* were the first to apply GAs [52]. Harman and McMinn [23] applied a theoretical analysis and empirical comparison of Random Search, Hill Climbing, Evolutionary Testing and Memetic Algorithms for branch coverage. The present paper is the first to undertake a detailed theoretical analysis of input domain reduction via irrelevant input variable removal for Search-Based Software Testing and conduct a large empirical study assessing its impact on Random Search, Hill Climbing, Evolutionary Testing and Memetic Algorithms on a series of wide-ranging test subjects.

In previous work, Harman *et al.* [19] were the first to consider search space reduction for Search-Based Test Data Generation. The authors investigated the impact of irrelevant input variable removal on Random Search, Hill Climbing and Evolutionary Testing. In the present paper, the empirical study is broadened to include further test subjects, while also investigating the impact of irrelevant input variable removal on Memetic Algorithms. The present paper also considers the impact of serendipitous coverage of non-targeted branches and investigates the impact of domain reduction on the trade off between test effectiveness and test effort.

This paper also performs an in-depth theoretical analysis of the impact of search space reduction on Search-Based Test Data Generation. Aspects of problem difficulty are investigated for search-based techniques and how these factors are impacted by removing irrelevant input variables from the search space. This analysis assumes that each branch is attempted by the search in serial, although in practice, other branches may be executed serendipitously. Arcuri *et al.* [7] have recently undertaken work performing a theoretical analysis of Random Testing, in which several branches are sought by the search at once. Arcuri [5] has also undertaken a theoretical analysis of the use of local search in software test data generation, proving that it is better than Random Testing for certain classes of program. Lammermann *et al.* [29] were the first to investigate problem difficulty for Evolutionary Testing. An empirical study was performed by testing the correlation between traditional software metrics and branches with which the evolutionary search struggled to find test data. No correlation was found, pointing to the need for specific metrics to be developed for search-based testability.

Irrelevant variable removal is one strategy for reducing the size of the input domain for search-based test data generation. Ribeiro *et al.* [41] have investigated input domain reduction strategies for testing object-oriented software, which typically involves generating sequences of method calls to objects to change their state so that test goals can be attained. The reduction strategy used involves removing methods from consideration by the search if they only query the state of an object, rather than mutating it, and thus being irrelevant for constructing a test sequence that changes the state of an object so that a test target can be executed. Harman *et al.* [20] investigated test data generation and search space reduction for aspect-oriented programs, reporting a decrease in search effort with reduction and an increase in the number of branches covered. As with the results presented in this paper, they also find the number of non-targeted branches covered also increased with search space reduction.

Korel proposed a method [26] closely related to the search space reduction technique proposed here, but for path coverage. In his approach, each input variable in the search underwent a risk analysis using an influences graph, constructed using dynamic data-flow information. The value of an input variable would remain fixed if it was highly likely that changes would impact current segments of the path that were currently being traversed correctly, or, if the input variable did not affect the path at all. Unfortunately, however, the method was not empirically evaluated.

Sagarna and Lozano [42] use a new search technique for branch coverage that works in a reverse fashion to input domain reduction. The size of the search space is iteratively increased, from an initially

small domain, until the optimal objective value is reached. The technique is based on the Estimation of Distribution Algorithm (EDA) [30]. EDAs differ from the search algorithms considered in this paper; they are a type of meta-heuristic search algorithm, but differ from a Genetic Algorithm with respect to the procedure of breeding new individuals.

The tool used in our experiments to reduce the search space produces dependence information as a by-product of static program slicing [50]. Source code analysis techniques such as symbolic execution [25] or abstract interpretation [13] could further support search space reduction by finding constraints or defining ranges on the input variables relevant to the branch under investigation. However, this remains a topic for future work.

7 Conclusions and future work

Search-Based Software Testing is a dramatically growing research field in which optimizing search techniques are used to automate aspects of the testing process, one such aspect being test data generation. Despite the large volume of work in this area, there has been little work investigating tactics for reducing the potentially very large input domain of a test object, which the optimization technique must search for test data.

This paper presented the irrelevant input variable removal strategy as means of reducing input domain dimensionality for search-based structural test data generation techniques. Irrelevant input variables are input variables that do not influence whether a target structure will be executed or not, and thus can be safely removed from the search process without making the target infeasible by doing so. The paper theoretically assessed and empirically evaluated the impact of input domain reduction for search-based structural test data generation by removing irrelevant input variables; variables that do not influence whether a structure can be executed or not. By studying the principles of problem difficulty for search-based test data generation techniques, the theory predicted that irrelevant input variable removal would have no impact on Random Search, but had the potential to enhance the performance of so-called *meta-heuristic* search techniques, which are guided by a fitness function, including Hill Climbing, Evolutionary Testing and a Memetic Algorithm. These theoretical findings were validated by a broad empirical study, which analysed 636 branches in seven pieces of software, including two embedded controller systems supplied by DaimlerChrysler. The empirical study revealed several statistically significant cases in which irrelevant input variable removal improved the performance of the meta-heuristic search.

Future work intends to further reduce the input domain by concentrating on relevant variables as well. Here, overall input domain size may be decreased by reducing the set of values from the domain to be searched, through the use of further static analysis.

This paper has only considered the effects of input domain reduction on the generation of test data and not its evaluation. The techniques may also impact the cost of assessing whether the corresponding outputs were correct for the inputs generated. This is referred to as the oracle cost [21, 35]. The removal of irrelevant variables may help simplify test cases and reduce oracle cost. However, the values of these irrelevant variables may be important if each test case should cover as many branches as possible, *i.e.* causing a reduction in the number of test cases to be evaluation. This potential trade-off is an issue for further work.

Of further interest is the issue of serendipitous coverage, *i.e.* branches that were executed in the course of a specific target branch. The paper found that serendipitous coverage tended to increase with irrelevant input variable removal. Further work should investigate and characterise the situations when irrelevant input variable removal is likely to result in higher serendipitous collateral coverage, and where it is not.

Finally, the search techniques appearing in this paper have been replicated from the work of other researchers [26, 47, 48]. An issue for future work, therefore, is whether the parameters of these algorithms (for example population size, mutation rate etc.) may be tuned in order to obtain better results following the application of input domain reduction. In addition, this paper only considered the effects of irrelevant input variable removal on branch coverage; future work will also seek to study its effects for other types of coverage.

Acknowledgements

The authors would like to thank the anonymous referees for their comments on earlier versions of this paper. The authors are also grateful to Lorna Anderson, Hamilton Gross and Becky McMinn for proof

reading.

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). Mark Harman is supported, in part, by EPSRC grants EP/G060525 (CREST Platform grant), EP/F059442 (SLIM : SLIcing state based Models) and EP/D050863 (SEBASE: Software Engineering By Automated SEarch) and by a development grant from DaimlerChrysler AG Berlin. Youssef Hassoun and Kiran Lakhotia are supported by the EPSRC CREST Platform Grant.

References

[1] The Software-artifact Infrastructure Repository (http://sir.unl.edu/portal/index.html).

[2] W. Afzal, R. Torkar, and R. Feldt. A systematic review of search-based testing for non-functional system properties. *Information and Software Technology*, 51(6):957–976, 2009.

[3] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test-case generation. *IEEE Transactions on Software Engineering*, 2010. To appear.

[4] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[5] A. Arcuri. Theoretical analysis of local search in software testing. In *Proceedings of Symposium on Stochastic Algorithms, Foundations and Applications (SAGA 2009)*, pages 156–168. Lecture Notes in Computer Science, Volume 5792, Springer Verlag, 2009.

[6] A. Arcuri. It does matter how you normalise the branch distance in search based software testing. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST 2010)*, pages 205–214. IEEE, 2010.

[7] A. Arcuri, M. Z. Iqbal, and L. Briand. Formal analysis of the effectiveness and predictability of random testing. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2004)*, pages 219–230. ACM, 2010.

[8] J. E. Baker. Reducing bias and inefficiency in the selection algorithm. In *Proceedings of the 2nd International Conference on Genetic Algorithms and their Application*, Hillsdale, New Jersey, USA, 1987. Lawrence Erlbaum Associates.

[9] D. W. Binkley, N. Gold, and M. Harman. An empirical study of static program slice size. *ACM Transactions on Software Engineering and Methodology*, 16(2):1–32, 2007.

[10] L. C. Briand, J. Feng, and Y. Labiche. Using genetic algorithms and coupling measures to devise optimal integration test orders. In *14th IEEE Software Engineering and Knowledge Engineering (SEKE 2002)*, pages 43–50, Ischia, Italy, 2002.

[11] O. Buehler and J. Wegener. Evolutionary functional testing of an automated parking system. In *International Conference on Computer, Communication and Control Technologies and The 9th International Conference on Information Systems Analysis and Synthesis (CCCT 2003 and ISAS 2003)*, Orlando, Florida, USA, 2003.

[12] S. Chen and S. Smith. Improving genetic algorithms by search space reductions. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 1999)*, pages 135–140. Morgan Kaufmann, 1999.

[13] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

[14] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405 – 435, 2005.

[15] J. W. Duran and S. C. Ntafos. An evaluation of random testing. *IEEE Transactions on Software Engineering*, 10(4):438–444, 1980.

[16] Grammatech Inc. The Codesurfer Slicing System, 2002.

[17] M. Harman. The current state and future of search based software engineering. In L. Briand and A. Wolf, editors, *Future of Software Engineering 2007*, pages 342–357, Los Alamitos, California, USA, 2007. IEEE Computer Society Press.

[18] M. Harman, C. Fox, R. M. Hierons, L. Hu, S. Danicic, and J. Wegener. Vada: A transformation-based system for variable dependence analysis. In *IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2002)*, pages 55–64, Los Alamitos, California, USA, Oct. 2002. IEEE Computer Society Press.

[19] M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, and J. Wegener. The impact of input domain reduction on search-based test data generation. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE 2007)*, pages 155–164, Cavat, near Dubrovnik, Croatia, 2007. ACM Press.

[20] M. Harman, F. Islam, T. Xie, and S. Wappler. Automated test data generation for aspect-oriented programs. In *Proceedings of the 8th International Conference on Aspect-Oriented Software Development (AOSD 2009)*, pages 185–196. ACM, 2009.

[21] M. Harman, S. G. Kim, K. Lakhotia, P. McMinn, and S. Yoo. Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem. In *Proceedings of the 3rd International Workshop on Search-Based Testing*. IEEE digital library, 2010.

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

[23] M. Harman and P. McMinn. A theoretical and empirical study of search-based testing: Local, global and hybrid search. *IEEE Transactions on Software Engineering*, 36:226–247, 2010.

[24] S. Horwitz, T. Reps, and D. W. Binkley. Interprocedural slicing using dependence graphs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 1988)*, pages 25–46, Atlanta, Georgia, June 1988. Proceedings in *SIGPLAN Notices*, 23(7), pp.35–46, 1988.

[25] J. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[26] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, 1990.

[27] J. Krinke and G. Snelting. Validation of measurement software as an application of slicing and constraint solving. *Information and Software Technology Special Issue on Program Slicing*, 40(11 and 12):661–675, 1998.

[28] K. Lakhotia, P. McMinn, and M. Harman. An empirical investigation into branch coverage for C programs using CUTE and AUSTIN. *Journal of Systems and Software*, 83:2379–2391, 2010.

[29] F. Lammermann, A. Baresel, and J. Wegener. Evaluating evolutionary testability with software-measurements. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2004), Lecture Notes in Computer Science vol. 3103*, pages 1350–1362, Seattle, USA, 2004. Springer-Verlag.

[30] P. Larrañaga and J. A. Lozano. *Estimation of Distribution Algorithms*. Kluwer, 2002.

[31] N. Mansour and M. Salame. Data generation for path testing. *Software Quality Journal*, 12(2):121–134, 2004.

[32] G. McGraw, C. Michael, and M. Schatz. Generating software test data by evolution. *IEEE Transactions on Software Engineering*, 27(12):1085–1110, 2001.

[33] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, 2004.

[34] P. McMinn, M. Harman, D. Binkley, and P. Tonella. The species per path approach to search-based test data generation. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2006)*, pages 13–24, Portland, Maine, USA, 2006. ACM.

[35] P. McMinn, M. Stevenson, and M. Harman. Reducing qualitative human oracle costs associated with automatically generated test data. In *Proceedings of the 1st International Workshop on Software Test Output Validation (STOV 2010)*, pages 1–4, Trento, Italy, 2010. ACM.

[36] W. Miller and D. Spooner. Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering*, 2(3):223–226, 1976.

[37] H. Mühlenbein and D. Schlierkamp-Voosen. Predictive models for the breeder genetic algorithm: I. continuous parameter optimization. *Evolutionary Computation*, 1(1):25–49, 1993.

[38] R. Pargas, M. Harrold, and R. Peck. Test-data generation using genetic algorithms. *Software Testing, Verification and Reliability*, 9(4):263–282, 1999.

[39] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2009. ISBN 3-900051-07-0.

[40] O. Räihä. A survey on search based software design. Technical Report D-2009-1, Department of Computer Sciences, University of Tampere, 2009.

[41] J. Ribeiro, M. Zenha-Rela, and F. de Vega. Test case evaluation and input domain reduction strategies for the evolutionary testing of object-oriented software. *Information and Software Technology*, 51:1534–1548, 2009.

[42] R. Sagarna and J. Lozano. Dynamic search space transformations for software test data generation. *Computational Intelligence*, 24(1):23–61, 2008.

[43] F. Schmiedle, R. Drechsler, and B. Becker. Exact routing with search space reduction. *IEEE Transactions on Computers*, 52(6):815–825, 2003.

[44] B. Steensgaard. Points-to analysis in almost linear time. In *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*, pages 32–41, St. Petersburg, Florida, Jan. 1996. ACM Press.

[45] N. Tracey, J. Clark, K. Mander, and J. McDermid. An automated framework for structural test-data generation. In *Proceedings of the International Conference on Automated Software Engineering (ASE 1998)*, pages 285–288, Hawaii, USA, 1998. IEEE Computer Society Press.

[46] J. Voas and K. Miller. Software testability: The new verification. *IEEE Software*, 12(3):17–28, May 1995.

[47] H.-C. Wang and B. Jeng. Structural testing using memetic algorithm. In *Proceedings of the Second Taiwan Conference on Software Engineering*, Taipei, Taiwan, 2006.

[48] J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841–854, 2001.

[49] J. Wegener and M. Grochtmann. Verifying timing constraints of real-time systems by means of evolutionary testing. *Real-Time Systems*, 15(3):275–298, 1998.

[50] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[51] D. Whitley. The GENITOR algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. In J. D. Schaffer, editor, *Proceedings of the International Conference on Genetic Algorithms (ICGA 1989)*, pages 116–121, San Mateo, California, USA, 1989. Morgan Kaufmann.

[52] S. Xanthakis, C. Ellis, C. Skourlas, A. Le Gall, S. Katsikas, and K. Karapoulios. Application of genetic algorithms to software testing (Application des algorithmes génétiques au test des logiciels). In *5th International Conference on Software Engineering and its Applications*, pages 625–636, Toulouse, France, 1992.