A Theoretical and Empirical Study of Search Based Testing: Local, Global and Hybrid Search

Mark Harman
King's College London,
CREST centre,
Strand,
London
WC2R 2LS, UK.
mark.harman@kcl.ac.uk

Phil McMinn*
University of Sheffield,
Department of Computer Science,
Regent Court, 211 Portobello,
Sheffield,
S1 4DP, UK.
p.mcminn@sheffield.ac.uk

Abstract

Search based optimization techniques have been applied to structural software test data generation since 1992, with a recent upsurge in interest and activity within this area. However, despite the large number of recent studies on the applicability of different search based optimization approaches, there has been very little theoretical analysis of the types of testing problem for which these techniques are well-suited. There are also few empirical studies that present results for larger programs.

This paper presents a theoretical exploration of the most widely studied approach, the global search technique embodied by Genetic Algorithms. It also presents results from a large empirical study that compare the behaviour of both global and local search based optimization on real world programs. The results of this study reveal that cases exist of test data generation problem that suit each algorithm, thereby suggesting that a hybrid global-local search (a Memetic Algorithm) may be appropriate. The paper presents a Memetic Algorithm along with further empirical results studying its performance.

*corresponding author

Keywords: Automated test data generation, search based testing, search based software engineering, evolutionary testing, genetic algorithms, hill climbing, schema theory, Royal Road

Categories and Subject Descriptors. D.2.5 [Software Engineering]: Testing and Debugging – *Testing Tools*; I.2.8 [Artificial Intelligence]: Problem Solving, Control Methods, and Search – *Heuristic Methods*

General Terms. Algorithms, Experimentation, Measurement, Performance, Theory

1 Introduction

There is strong empirical evidence [36, 20] that deficient testing of both functional and non-functional properties is one of the major sources of software and system errors. In 2002, NIST estimated the cost of software failure to the US economy at $\$6 \times 10^{10}$; which was 0.6% of GDP at the time [49].

The same report found that more than one third of these costs of software failure could be eliminated by an improved testing infrastructure.

Automation of testing is a crucial concern [8]. Through automation, large-scale thorough testing can become practical and scalable. However, the automated generation of test cases presents challenges. The general problem involves finding a (partial) solution to the path sensitisation problem. That is, the problem of finding an input to drive the software down a chosen path. Of course, the underlying problem of path sensitisation is known to be undecidable, so research has focused on techniques that seek to identify near optimal test sets in reasonable time.

One class of techniques that has received much recent attention consists of applying search based optimization to the problem of software test data

generation, an approach that has come to be known as Search Based Testing [40], because it denotes an exemplar of the class of applications referred to as Search Based Software Engineering [22, 25].

Search Based Testing is the process of automatically generating test data according to a test adequacy criterion (encoded as a fitness function) using search based optimization algorithms, which are guided by a fitness function. The role of the fitness function is to capture a test objective that, when achieved, makes a contribution to the desired test adequacy criterion. Using the fitness function as a guide, the search seeks test inputs that maximize the achievement of this test objective.

The search based approach is very generic, because different fitness functions can be defined to capture different test objectives, allowing the same overall search based optimization strategy to be applied to very different test data generation scenarios. The approach has been successfully applied to structural testing [7, 9, 33, 38, 39, 43, 51, 56, 57, 60, 61], temporal testing [58], stress testing [12], mutation testing [30], finite state machine testing [15] and exception testing [55]. Of these test adequacy criteria, the most widely studied has been structural testing, with a particular focus on branch coverage. This has been motivated by the importance of branch coverage and its variants in testing standards [13, 52] and the cost of generating branch adequate test inputs by hand.

Evolutionary Testing is a sub-field of Search Based Testing in which Evolutionary Algorithms are used to guide the search. These algorithms use a global search (most commonly, but not exclusively implemented as a Genetic Algorithm). Evolutionary Testing has been widely studied in the literature, where it has been applied to many test data generation scenarios including temporal testing [58], stress testing [12] and exception testing [55]. Since Evolutionary Testing uses a global search approach to find structural test data, it is natural to

compare the efficiency and effectiveness of Evolutionary Testing with a widely studied local search technique. The most widely studied local search technique for search based test data generation is Korel's Alternating Variable Method [33, 34, 18]; an approach which is a form of Hill Climbing.

Despite the considerable level of interest in Search Based Testing, to date there has been no theoretical analysis that characterizes the types of test data generation scenario for which different types of search (global or local) are predicted to be effective. As a result, there is a serious lack of firm, scientific underpinning for what has become a widely researched approach to test data generation. Furthermore, the empirical results for Search Based Testing tend to consider small, artificial 'laboratory programs' rather than real world programs with large and complex search spaces. This leaves the literature with an important open question

> "Which type of search, global (Evolutionary Testing) or local (Korel's Alternating Variable Method), is best for which type of structural test data generation problem?"

Global search techniques aim to overcome the problem of local optimum in the search space and can, thereby, find more globally optimal solutions. Local search may become trapped in local optima within the solution space, but can be far more efficient for simpler search problems. In software engineering terms, this establishes an apparent instance of the classic trade off between efficiency and effectiveness; one might expect global search to achieve better branch coverage than local search, but at the cost of greater computational effort. However, as the results in this paper will show, the situation is a lot more subtle than the assertion that 'global search is more effective but less efficient'. Perhaps surprisingly, the findings reveal strong performance for local search.

This paper addresses the question theoretically as well as empirically. The

complex behaviour of global search makes it harder to reason about its theoretical performance. Global search involves a collection (a 'population') of candidate solutions that evolve over time, allowing for a wide sampling of the search space. By contrast, local search uses the fitness function to evaluate possible moves within the search space from a single current solution point until a local optimal is reached.

To address this, the paper presents a theoretical development of Holland's well known schema theory [53]. The schema theory was later developed by Mitchell *et al.* [47] in a study of the so-called 'Royal Road' functions, which account for the effect of the all-important crossover operator, unique to Genetic Algorithms. The crossover operator attempts to build fitter (*i.e.* better) candidate solutions from good solutions present in the current population, by recombining the elements that make up the 'chromsome' of each solution. Both the schema theory and the Royal Road theory were developed purely for chromosomes represented as bit strings, and have not been previously adapted for the more complex chromosomes required by Evolutionary Testing.

According to schema theory [53], Genetic Algorithms should be effective at generating test data for problems where the test inputs contain building blocks, *i.e.* sets of potentially productive chromosomal elements ('schemata'). Such building blocks imbue all chromosomes with a higher fitness than they would have had if their productive schema were replaced with some other less productive schema. Furthermore, in order for crossover to work as an effective evolutionary search operator, there is a necessary 'Royal Road' property, which means that productive schemata contain subsets which are also productive. Though not all subsets need to be productive, some must be. If not, then crossover cannot combine two partially fit individuals to produce fitter offspring. The paper introduces a generalization of both theories that caters for Evolutionary

Testing, showing how the generalized theory predicts the kinds of 'Royal Road' search problem for which Evolutionary Testing will be well suited.

The paper then presents the results of a large scale empirical study that plays two roles: it validates the predictions of the theory and it answers the questions concerning the relative performance of global and local search. Random Testing is also included to provide a baseline, indicating which branches denote non-trivial optimization problems. The results show that local search can be very effective and efficient, but there remain Search Based Testing optimization problems for which global search is the only technique that can successfully achieve coverage. These were found to display the 'Royal Road' property predicted as ideal for Genetic Algorithms, together with plateaux features that render local search ineffective.

The strong performance of local search, coupled with the necessity to retain global search for optimal effectiveness, naturally points to the consideration of hybrid techniques. The paper presents a further empirical study designed to address the question of whether the best overall results can be achieved by combining Evolutionary Testing and Hill Climbing into a hybrid Memetic Algorithm approach. The results confirm that this hybrid approach is capable of the best overall performance.

More specifically, the primary contributions of the paper are:

1. The introduction of a schema theory and Royal Road theory for Evolutionary Testing that predict the structural test data generation problems to which Evolutionary Testing will be well suited.

2. An empirical validation of the predictions of the theory that provides evidence to support the claim that Evolutionary Testing does indeed perform well for Royal Road functions and that this is due to the effect of the crossover operation.

3. An empirical assessment of the performance of Evolutionary Testing compared to Hill Climbing and Random Testing. This empirical study has several findings, some of which are surprising:

   (a) The results support the view that Random Testing can find test data for many cases, but leaves some hard–to–cover branches for which more intelligent search is required.

   (b) Where test data generation scenarios do not have a Royal Road property, Hill Climbing performs far better than Evolutionary Testing. This is surprising, given the emphasis on Evolutionary Testing in the literature; perhaps there has been an over-emphasis on Evolutionary Testing at the expense of other more simple search techniques.

   (c) Though Hill Climbing outperforms Evolutionary Testing in non-Royal Road scenarios, Royal Road scenarios do exist in which Evolutionary Testing is successful and Hill Climbing and Random Testing fail.

4. An empirical assessment of a hybrid Memetic Algorithm approach which incorporates Hill Climbing into Evolutionary Testing. Findings from this empirical study were as follows:

   (a) The Memetic Algorithm can cover branches with Royal Road properties, but with lower success rate than Evolutionary Testing. That is, though branches are covered after several runs, the likelihood of success on each individual run is reduced, indicating that the price of maintaining success with the Memetic approach is greater computational cost. Fortunately, test input generation is a task for which a practicing software engineer may be prepared to wait for optimal coverage.

(b) The Memetic Algorithm is successful at covering all but one of the non-Royal Road branches that Hill Climbing and Evolutionary Testing are capable of covering. For non-Royal Road branches the efficiency of the approach is consistent with that of Hill Climbing. More detailed analysis reveals that the single uncovered branch is a special 'pathological' case.

(c) In terms of coverage, the hybrid approach is capable of the best overall performance. The results with Royal Road branches, however, indicate that fine-tuning of the balance between intensification and diversification of the search process is required for a more optimal performance.

The rest of the paper is organised as follows. Section 2 provides a detailed description of the Evolutionary Testing, Hill Climbing and the Memetic Algorithm used in the paper to facilitate replication. Section 3 introduces the Schema and Royal Road Theory for Evolutionary Testing, while Section 4 presents the results of the empirical study that both validates the theory and addresses performance questions, with Section 5 discussing potential threats to validity. Section 6 presents related work, while Section 7 concludes.

2 Search Based Testing

Search based test data generation searches a test object's input domain to automatically find test data, guided by a fitness function. This paper concentrates on structural test data generation, which is the most widely studied of all the applications of search based techniques to the test data generation problem. The paper considers branch coverage, a widely used structural test adequacy criterion. However, the results can be extended to apply to other forms of structural

```
...
for (checksum = 0, k = 0, new_ISBN = YES; n = 1; current_value[n+1]; ++n)
{
  ...
  switch (current_value[n])
  {
    case ' ': case '-':
      /* ignore space and hyphen */
      break;

    case '0': case '1': case '2': case '3': case '4': case '5':
    case '6': case '7': case '8': case '9': case 'X': case 'x':
      /* valid ISBN digit */
      k++;
      if (k < 10) /* Node 23 (originating node of branches '23T' and '23F') */
      {
        ISBN[k] = current_value[n];
        checksum += ISBN_DIGIT_VALUE(ISBN[k]) * k;
        break;
      }
      else if (k == 10) /* Node 27 (originating node of branches '27T' and '27F') */
      {
        ISBN[k] = current_value[n];

         /* Node 29 (originating node of branches '29T' and '29F') */
        if ((checksum % 11) != ISBN_DIGIT_VALUE(ISBN[k]))
          bad_ISBN(ISBN);
        new_ISBN = YES;
        break;
      }

    default:
      /* ignore all other characters */
      if (k > 0) /* then only got partial ISBN */
      {
        bad_ISBN(ISBN);
        new_ISBN = YES; /* start new checksum */
      }
      break;
  } /* end switch (current_value[n]) */
} /* end for (loop over current_value[]) */
...
```

Figure 1: Code snippet of the `check_ISBN` function. The snippet shows the main loop of the function which iterates over a series of characters. Valid digits are collated and used to accumulate a checksum. When a certain number of valid characters are entered, the checksum is used to validate the complete ISBN. The comments in italics are added to identify certain branches of interest to the empirical study in Section 4

test data generation.

The `check_ISBN` function in Figure 1 is used to demonstrate some of the concepts in this section. The function, which is part of the open source `bibclean` program, shows a snippet of a function used to validate ISBNs. The snippet depicts the central part of the function, which is a loop that iterates over an array of characters, `current_value`. The array represents an ISBN to be validated. Within the loop body, the function checks if the individual array characters are valid ISBN characters, whilst maintaining a checksum. Once ten characters have been entered the checksum is evaluated to see if the complete ISBN is valid.

The `check_ISBN` function is used as part of the empirical study in this paper to evaluate different approaches to search based structural test data generation. For branch coverage, a separate search process is undertaken in order to find test data that executes each uncovered branch. A fitness function scores inputs with respect to how close they were to covering the target branch, and is to be minimized by the search. Therefore, lower numerical fitness values represent fitter input vectors.

The fitness function combines a measure known as the *approach level* with the *branch distance* [57]. The approach level is a count of how many of the branch's control dependent nodes were *not* encountered in the path executed by the input. An input which executes more control dependent nodes is 'closer' to reaching the target in terms of the control flow graph, and thus is rewarded with a lower approach level. Suppose, for example, the target of the search is to find test data to execute the true branch from control flow graph node 27 of the `check_ISBN` function ('27T'). If the predicate `k == 10` is reached, all control dependent nodes are executed, and the approach level is 0. If `k < 10`, however, node 27 is not encountered, and the approach level is 1. If none of the case

statements are true, or the loop is not even entered, the approach level takes on higher integer values.

The branch distance is computed using the values of variables at the predicate appearing in the conditional where control flow went 'wrong' - *i.e.*, where the path diverged away from the target branch. It reflects how close the predicate came to switching outcome, causing control to pass down the desired alternative branch. For example, if the false branch were taken from node 27 of the `check_ISBN` function the branch distance is computed using the formula `|k - 10|`. The closer the value of `k` is to `10` the 'closer' the conditional is deemed to being true. If the conditional is encountered several times in the body of the loop, the smallest branch distance is used.

A full list of branch distance formulas for different predicate types can be found in McMinn's survey [40]. The complete fitness value is traditionally computed by normalizing the branch distance and adding it to the approach level [57] according to the following equation:

$$fitness = approach\_level + normalize(branch\_distance)$$

The branch distance is normalized using the formula:

$$normalize(branch\_distance) = 1 - 1.001^{-branch\_distance}$$

The above normalization formula does not require the maximum or minimum value of the branch distance to be known, which would require complex analysis. In principle, any normalization formula can be used, so long as it is not so coarse as to be useless in distinguishing input vectors which produce lower branch distances from those which result in higher values. For the algorithms featured in this paper, candidate solutions are compared and ranked according to their

relative fitness values, thus the absolute values produced are of less importance.

The next two subsections describe in detail the implementation of Evolutionary Testing, Hill Climbing and the hybrid Memetic Algorithm approach used in this paper so as to facilitate accurate replication.

2.1 Genetic Algorithms and Evolutionary Testing

Genetic Algorithms belong to the family of Evolutionary Algorithms, which work to evolve superior candidate solutions using mechanisms inspired by the processes of natural Darwinian evolution. The search simultaneously evolves several individuals in a population, creating a global search. This section begins with a description of a basic Genetic Algorithm, and then explains how the Genetic Algorithm used by Evolutionary Testing differs.

Genetic Algorithms

Figure 2 outlines the main steps of a Genetic Algorithm. The first stage is the initialization of a population of n candidate solutions, known as 'individuals', at random. The chromosomes representing each individual are encoded as bit strings for manipulation by the algorithm. After initialization the Genetic Algorithm enters a loop, comprising of distinct stages of *evaluation*, *selection*, *crossover*, *mutation* and *reinsertion*.

The *evaluation* phase is simply where each individual is assessed for fitness using the fitness function. *Selection* is the process of choosing 'parent' candidate solutions which will be 'bred' in the *crossover* phase to produce offspring solutions and then subject to a stage of *mutation*. Crossover loosely models the exchange of genetic information that takes place during reproduction in the natural world. There are many choices of crossover operator. Simple one-point crossover involves selecting a crossover point where two parent chromosomes

are to be spliced in order to form the composite chromosomes of two children. The example of Figure 3 shows the crossover of two bit strings at position 3 using one-point crossover. *Mutation* involves random modification of offspring to introduce diversity into the search. Traditionally this involves flipping bits in each individual's chromosome at a probability of p_m, where p_m is typically $1/len$ where len is the length of the chromosomal bit string.

As with all evolutionary computation, the hope is that crossover will combine the best features of both parents to create super-fit children from fit parents, and that mutation will also help discover fitter individuals. Where this fails to take place, bias involved in the selection phase ensures that less-fit individuals have less chance of being selected to reproduce in the next iteration of the algorithm, and thus 'die out'. One type of selection process is *fitness-proportionate selection*, where individuals are selected at a probability that is proportionate to their fitness value compared to other individuals in the population. However, over-selection of the best candidate solutions may result in premature convergence if the populations become dominated by a few super-fit individuals. Thus, ranking selection methods are often preferred. Individuals are ranked in fitness order, and are then chosen randomly at a probability proportionate to their rank.

Reinsertion involves forming a new generation of individuals from the current population and the generated offspring. One approach is to replace all of the current population with offspring, another is an *elitist* strategy, where the best individuals are retained, taking the place of some of the weakest newly-generated offspring, which are discarded.

Finally, the new individuals of the population are evaluated for fitness. At each evaluation stage, a test is performed to see if the goal of the search has been met *i.e.* the global optimum has been found; or if the search has failed, and

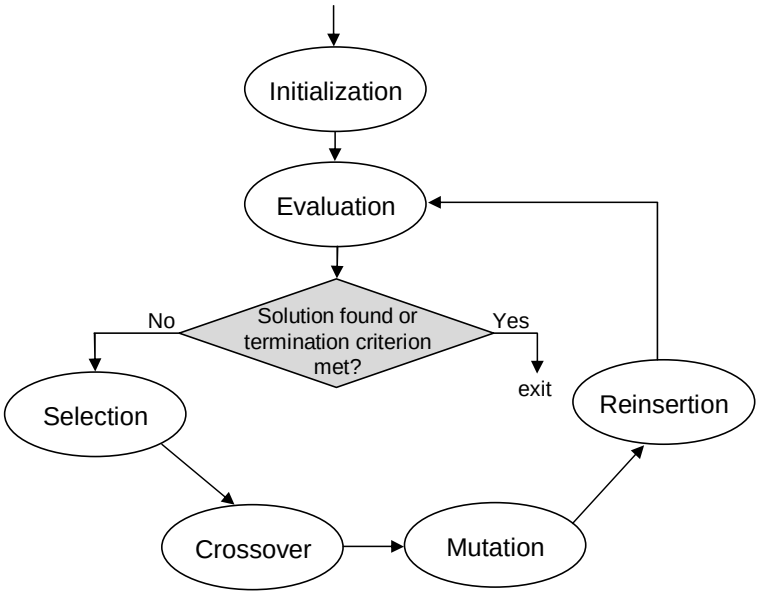


Figure 2: Overview of the main steps of a Genetic Algorithm

```
000000      000111
111111  ↣   111000
```

Figure 3: One-point crossover of two bit strings at position 3

should be terminated. Termination conditions tend to test if a certain number of trials (fitness evaluations) have been performed, or a certain number of loops of the algorithm ('generations') have been performed.

Evolutionary Testing

It is the application of Genetic Algorithms to Search Based Testing that has become known as Evolutionary Testing. The rest of this section presents the details of the algorithm implemented for Evolutionary Testing used in this paper. The approach is based on a careful replication of the DaimlerChrysler system for Evolutionary Testing, which has been widely studied in the literature [6, 7, 57]. The DaimlerChrysler system has been developed and improved over a

period of over a decade and so it can be argued to be the 'state of the art' in Evolutionary Testing. The aim of using this Evolutionary Testing approach is to ensure that the results for Evolutionary Testing do, indeed, represent the state of the art. This lends additional weight to any findings that reveal superior performance by the comparatively straightforward Hill Climbing approach, to which Evolutionary Testing is compared. As will be seen (in Section 4), the empirical study does indeed yield such results.

For Evolutionary Testing the chromosome making up each individual is a direct representation of the input vector to the program concerned. The 'genes' of the chromosome represent the input values with which the program will be executed, because test data generation requires chromosomes that must respect typing information embodied in any valid input type [57]. As will be seen in the next section, the richer chromosome types required by Evolutionary Testing entail a generalization of Holland's schema theory of Genetic Algorithms in order to render it applicable to Evolutionary Testing.

Evolutionary Testing uses a population of 300 individuals, and in contrast to the basic Genetic Algorithm described earlier, is further subdivided into 6 subpopulations, initially of 50 individuals each. As will be explained in detail later, the subpopulations compete for a share of individuals as the search progresses. Individuals can also migrate from one subpopulation to another at various pre-defined points.

The selection strategy first applies a *linear ranking* [59] of individuals, in order to promote diversity and prevent super-fit individuals dominating the selection process. Each individual is assigned a special rank value, which depends on the individual's position in the overall population when sorted by fitness. The linear ranking mechanism assigns rank values such that the best individual receives a value Z, the median individual receives a value of 1.0, and the worst

individual receives a value of $2 - Z$, where Z is a parameter in the range [1.0, 2.0]. The value of Z used is 1.7. Stochastic universal sampling [5] is then used on the basis of the ranked values, with the probability of an individual being selected for reproduction proportionate to its fitness.

Once a pool of individuals have been selected, parents are taken two at a time, starting at the beginning of the list. Discrete recombination [48] is used as the crossover operator. Discrete recombination is similar to uniform crossover; every position in the chromosome is a potential crossover point. However, unlike uniform crossover, a gene can be copied into one or both children with an even probability.

The traditional mutation operator described in the last section, which flips bits in a bit string, cannot be usefully applied to real values. Evolutionary Testing employs the mutation operation of the Breeder Genetic Algorithm [48]. This operator mutates genes through the addition and subtraction of values of varying magnitude, and is designed to work with the 6 subpopulations. Each subpopulation can invoke a different magnitude of mutation; subpopulation #1 can make large mutations, whilst subpopulation #6 can only make relatively small mutations. As subpopulations compete for a share of the number of individuals that they can evolve, the search can allocate resources according to where the most progress is being made. For example, the input vectors randomly generated at the beginning of the search could be far away from the required test data. Large mutation steps result in the largest increases in fitness, and the subpopulations responsible are rewarded with individuals transferred from 'weaker', less successful subpopulations. As the search progresses, the general area of the input domain containing the test data may have been located, with input vectors found that are closer to executing the required branch. At this point, fine-tuning is required. The large mutation step subpopulations fail to

make further ground, eventually losing resources to the small mutation size subpopulations, which grow in strength.

As with a basic Genetic Algorithm, an input variable x_i is mutated at a probability of $p_m = 1/len$, where len this time represents the size of the input vector rather than a bit string. Each subpopulation p ($1 \leq p \leq 6$) has a different mutation step size, $step_p = 10^{-p}$, which is used in combination with the variable's domain size $domain_i$ to define the mutation range, $range_i$:

$$range_i = domain_i \cdot step_p$$

The new value z_i is computed using the following formula:

$$z_i = x_i \pm range_i \cdot \delta$$

Addition or subtraction is decided with an even probability. The value of δ is defined to be $\sum_{x=0}^{15} \alpha_x \cdot 2^{-x}$, where each α_x is 1 with a probability of 1/16 else 0. On average, therefore, one α_x will have a value of 1. If the mutated value falls outside the bounds of the variable, its value is reset to its lower or upper limit.

The next generation is then constructed using an elitist reinsertion strategy. The best 10% of the current generation is retained, with the remaining places filled with the best 90% of the new offspring.

After the reinsertion phase, Evolutionary Testing transfers individuals from one subpopulation to another according to its competition and migration strategies. Competition ensures that more resources (*i.e.* individuals) are devoted to subpopulations that are performing well. Migration attempts to avoid subpopulations becoming 'stale' and stagnating, due to a lack of diversity amongst individuals. Individuals are injected from different subpopulations that may

contain new genetic material. Every 20 generations, subpopulations exchange a random 10% of their individuals with one another.

The competition algorithm is careful to ensure that the transfer of resources between subpopulations is not subject to rapid fluctuation. A progress value is computed for each subpopulation at the end of each generation. The average fitness is then found for each subpopulation, using linearly ranked fitness values for each individual. The subpopulations are then themselves linearly ranked (again using $Z = 1.7$). The progress value, $progress_g$, of a subpopulation at generation g is computed using the formula:

$$0.9 \cdot progress_{g-1} + 0.1 \cdot rank$$

Every four generations, a slice of individuals is computed for each subpopulation in proportion to its progress value. Subpopulations with a decreased share lose individuals to subpopulations with an increased allocation. However, subpopulations are not allowed to lose their last five individuals, ensuring individual subpopulations cannot disappear completely.

2.2 Hill Climbing

Hill Climbing is a comparatively simple local search algorithm that works to improve a single candidate solution, starting from a randomly selected starting point. From the current position, the neighbouring search space is evaluated. If a fitter candidate solution is found, the search moves to that point. If no better solution is found in the neighbourhood, the algorithm terminates. The method has been called 'Hill Climbing', because the process is likened to the climbing of hills on the surface of the fitness function (referred to as the 'fitness landscape'). Since the fitness is to be minimized in this case, the equivalent term 'gradient descent' is potentially less confusing.

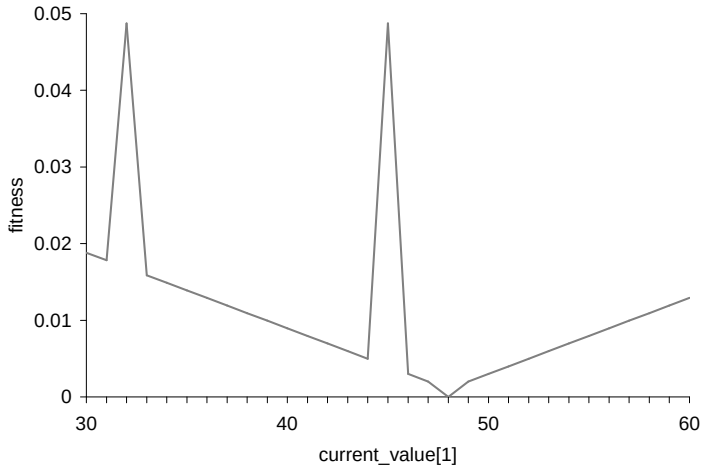


Figure 4: Fitness landscape for the coverage of the true branch from the `case '0'` statement of `check_ISBN` for one array value. The required value is found at the base of the valley, *i.e.* 48, the ASCII value of '0'. The peaks are caused by the space and hyphen characters which cause the function to drop out of the switch block before the branch distance can be evaluated for this particular case

For example, the coverage of the true branch from the '`case '0'`' statement of the `check_ISBN` function is represented in the fitness landscape visualisation of Figure 4 with the valley touching zero on the x axis of the fitness function at 48; the ASCII value for the character '0'.

As with Evolutionary Testing, there are many choices in the formulation of a Hill Climbing algorithm. The approach used in this paper is the 'Alternating Variable Method', which was used by Korel in early papers in the search based test data generation literature [33, 34, 18], hereinafter referred to simply as 'Hill Climbing'. The method takes each input variable in turn and adjusts its value in isolation from the rest of the vector. If altering the variable does not result in better fitness, the next input variable is selected, and so on, until no modification of input values results in an improved fitness.

The alternating variable method proposed by Korel expects each individual input variable to be of an ordinal type. In this paper, the method is extended to

handle floating-point types by requiring that the tester set the accuracy of each floating-point variable. Character types, such as those used in the `check_ISBN` function, are treated as integer values. The method is explained in terms of the `check_ISBN` function, whose input is an array of characters.

Initially, an input vector is generated at random. The first input value is selected, and its neighbourhood probed. Suppose the target of the search is the execution of the third case of the `switch` statement of the `check_ISBN` function, *i.e.* where a '0' character is required, and the input vector is <'b', '5', 'x', ... >. The first value of the `current_value` array is probed through 'exploratory' moves. Neighbouring ordinal values are 'a' and 'c'. Recall how the fitness calculation works; the branch distance is the smallest distance encountered in the loop. The second element of the array, '5', is the closest to '0'. Therefore modifying the first value of the array has no effect on fitness. However exploratory moves around the second element, '5', do have an effect - the value '4' reduces the branch distance by 1, as it is one character closer to the target '0'.

Once a better fitness has been found, further 'pattern' moves are made in the direction of improvement. In this paper, the value of the ith move m_i made in the direction of improvement, $dir \in \{-1, 1\}$ is computed using $m_i = 2^i \cdot 10^{-acc_v} \cdot dir$. acc_v is the accuracy set for each floating-point variable in decimal places, and is zero for integer and character types.

Successive pattern move values for `current_value[1]`, moving through the respective integer (ASCII) values, are therefore 52 ('4'- the initial move), 50 ('2'), 46 ('.'). At the consideration of '.', accelerated pattern moves lead the search to miss the base of the valley. This is recognised through a non-improvement of fitness. Therefore the search stops and re-establishes a new direction through further exploratory moves, proceeding with new pattern moves from this point.

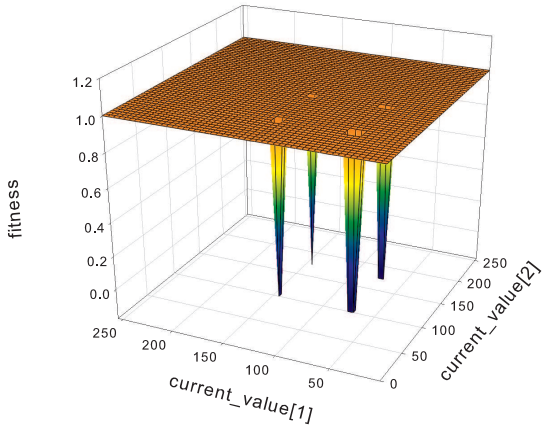


Figure 5: Fitness landscape for the coverage of the true branch from node 27 of `check_ISBN`

Eventually the character value '0' (ASCII value 48) is found for `current_value[1]`.

A well-known problem with local search methods like Hill Climbing is their tendency to become trapped in a local minima. For example, consider again coverage of the true branch from node 27 of the `check_ISBN` function. The relationship between the values of the input `current_value` and values of the variable `k` used in the nodes s predicate is not as simple and direct as that for the case statements previously considered. For the most part, exploratory moves for values of `current_value` have no effect on `k`, and result in the same fitness value. The fitness landscape for this branch is depicted in Figure 5. It shows one large plateaux. The test data are found where the surface of the fitness landscape touches zero on the z axis, *i.e.* the four pits. However, the predominantly flat surface does not lead the search to their location. On encountering such fitness-invariant plateaux, Hill Climbing terminates without finding the required test data. In order to address this problem, Hill Climbing is restarted at a new randomly chosen start point many times, until a budget of fitness evaluations has been exhausted.

2.3 Hybrid Memetic Algorithm Approach

Memetic Algorithms are Evolutionary Algorithms which employ a stage of local search to improve each individual at the end of each generation.

The Memetic Algorithm used in this paper combines the Evolutionary Testing and Hill Climbing methods described in the previous section. However, some important modifications are made in order to balance the new hybrid algorithm's abilities to (a) *intensify* the search *i.e.* to concentrate on an explicit sub-region of the search space, and (b) *diversify* the search, *i.e.* explore new and unseen areas.

Firstly, the Hill Climbing phase terminates for each individual upon reaching a local optima, and does not restart. Secondly, a smaller population size of 20 is employed, without the use of subpopulations. In the hybrid algorithm, Hill Climbing is used to intensify the search on particular areas of the search space, effectively fulfilling the role of the subpopulations with different mutation step sizes used with Evolutionary Testing. Since diversification does not happen until the end of each generation, in the form of crossover and large mutation steps, a reduced population size is also necessary to prevent the search spending the majority of its time merely intensifying around the space of its current set of individuals.

Finally, the Breeder Genetic Algorithm mutation operator is replaced with uniform mutation, which encourages greater diversification, balancing the high intensification of the Hill Climbing phase. Uniform mutation simply involves overwriting an input variable value with a new value from its domain, chosen uniformly at random. The same probability of mutation is used, *i.e.* $p_m = 1/len$.

3 Theoretical Foundations

This section presents an overview of the schema and Royal Road theories and introduces a generalization of both that caters for Evolutionary Testing.

3.1 The Schema Theory of Genetic Algorithms

In a binary Genetic Algorithm, a schema is a sequence consisting of three possible values, drawn from the set $\{0, 1, *\}$. The asterisk, $*$, is a wildcard, indicating that either a zero or a one could occur at this position. Thus, for a chromosome of length four, a schema $10*1$ denotes the two chromosomes 1011 and 1001, while the schema $****$ denotes all possible chromosomes. A schema can be thought of as a template chromosome that stands for a whole set of individual chromosomes, each of which share some common fixed values. An instantiation of a schema is any chromosome that matches the template, when $*$ values are replaced by the corresponding fixed values of the instantiation. If a chromosome x is an instantiation of a schema $\mathbf{h}$, this is denoted $x \in \mathbf{h}$.

The number of fixed positions in a schema is called the order of the schema. The schema $1**1$ has order two, while $10*1$ has order three. For a schema, $\mathbf{h}$, the order will be denoted $o(\mathbf{h})$. Suppose that the length of a chromosome is denoted by λ. A schema, $\mathbf{h}$, denotes $2^{\lambda - o(\mathbf{h})}$ chromosome instantiations, all of which have their own individual fitness values.

It is useful to think of the chromosomes denoted by a schema $\mathbf{h}$ as defining the fitness for the schema itself. That is, the schema has a fitness that is defined to be the average fitness of the instantiations it denotes. More formally, the fitness of a schema $\mathbf{h}$, is defined as follows:

$$\overline{f}(\mathbf{h}) = \frac{1}{\mid \mathbf{h} \mid} \sum_{x \in \mathbf{h}} f(x)$$

In a particular generation of a Genetic Algorithm, g, the population will be a set of chromosomes, denoted $P(g)$. The Genetic Algorithm will not be able to determine the true fitness of a schema $\mathbf{h}$, because it will not necessarily contain all possible instantiations. However, the schema processing at each generation g, will be able to approximate the fitness of $\mathbf{h}$, based on the instantiations of $\mathbf{h}$ present in the population $P(g)$. For this reason it is useful to define a measure of approximate fitness, $\overline{f}(\mathbf{h}, K)$ for a set of chromosomes K.

$$\overline{f}(\mathbf{h}, K) = \frac{1}{\mid \{x \mid x \in \mathbf{h} \wedge x \in K\} \mid} \sum_{x \in \mathbf{h} \wedge x \in K} f(x)$$

For a generation g, $\overline{f}(\mathbf{h}, P(g))$ is the approximate value of the fitness of $\mathbf{h}$ based upon the current members of the population at generation g.

The distance between the outermost fixed positions in a schema is known as the defining length.

This is measured as one less than the length from the leftmost outermost fixed position to the rightmost outermost fixed position. Thus, the schema, $1 * *0$ has defining length 3, the schema $*101$ has defining length 2 and the schema $*11*$ has defining length 1. The defining length of a schema $\mathbf{h}$, will be denoted $\delta(\mathbf{h})$. As the Genetic Algorithm executes, it evaluates the fitness of the chromosomes in each generation.

Every time a chromosome is evaluated, the evaluation contributes to the estimation of the fitness of 2^{λ} schemata, so the Genetic Algorithm processes a large number of schemata, far larger than the number of individual chromosomes that it evaluates.

The principle that underlies the schema theory is that those schemata with a better than average fitness will receive proportionally more fitness evaluations as the computation of the Genetic Algorithm progresses. This observation can be made more formal, by considering the number of instances of a schema that

pertain at each generation of the Genetic Algorithm.

Without the presence of mutation and crossover, but merely with selection, the number of occurrences of a schema $\mathbf{h}$ at generation $g+1$ can be bounded below in terms of the number of occurrences of $\mathbf{h}$ at generation g. Let the number of occurrences of $\mathbf{h}$ at generation g be denoted $N(\mathbf{h}, g)$. The schema theory (without mutation and crossover), for a population of size M is:

$$N(\mathbf{h}, g+1) \geq N(\mathbf{h}, g) \frac{\overline{f}(\mathbf{h}, P(g))}{\frac{1}{M} \sum_{x \in P(g)} f(x)}$$

That is, the term

$$N(\mathbf{h}, g) \frac{\overline{f}(\mathbf{h}, P(g))}{\frac{1}{M} \sum_{x \in P(g)} f(x)}$$

places a lower bound on the expected number of occurrences of schema $\mathbf{h}$ in generation $g+1$. It is the product of the number of occurrences at generation g and the ratio of the schemata approximate fitness at generation g and the average fitness of the entire population at generation g.

The idea that underlies the bound is that the term

$$\frac{\overline{f}(\mathbf{h}, P(g))}{\frac{1}{M} \sum_{x \in P(g)} f(x)}$$

denotes a value that is proportional to the probability that the schema $\mathbf{h}$ will be selected. For instance, if tournament selection is used, then the chance that an instance of $\mathbf{h}$ will prevail in a tournament with an arbitrary choice of opponent, $\mathbf{o}$ is clearly proportional to the ratio of the fitness of $\mathbf{h}$ relative to the fitness of the entire population, because $\mathbf{h}$ prevails if and only if it has a higher fitness than $\mathbf{o}$.

The fitter $\mathbf{h}$ is, relative to the overall population, the better its chances

to prevail in a tournament. The number of occurrences of a schema **h** that survive from one generation to the next is therefore proportional to the number of occurrences of **h** in the current population, multiplied by the probability that these schemas will prevail and therefore pass on to the next generation. This is treated as a lower bound since chance may also produce more of a schema through the effect of genetic operators. However, of course, operators governed by elements of chance, such as mutation and crossover also have a chance to disrupt a schema, as will be seen below.

Perhaps the 'schema theory' would be more accurately termed the 'schema hypothesis', since this equation is not proved in the literature. Rather, the equation captures the belief that above average fitness schemata will tend to receive exponentially more fitness evaluations than below average schemata as the Algorithm progresses.

Both mutation and crossover disrupt schemata in a population. The mutation operation can replace a bit of a productive (high fitness) schema with a bit that mutates the overall schema to a less fit schema. The crossover operator may replace a whole section of the schema with a less fit string of bits. To take account of mutation and crossover, the schema theory is extended to take account of the mutation probability (p_m, the probability that an individual bit is mutated) and the crossover probability (p_c).

$$N(\mathbf{h}, g+1) \geq N(\mathbf{h}, g) \frac{\overline{f}(\mathbf{h}, P(g))}{\frac{1}{M} \sum_{x \in P(g)} f(x)} (1 - p_c \frac{\delta(\mathbf{h})}{\lambda - 1} - p_m \, o(\mathbf{h}))$$

It is this equation that is known as the schema 'theorem' of Genetic Algorithms, due to Holland [29]. It makes the implicit assumption that crossover is a single point crossover operation and that mutation is achieved by flipping a single, randomly chosen, bit of the chromosome. Holland's schema theory is a 'worst case' formulation, because it places a lower bound on the number

of schemata present at each generation of the evolution of the Genetic Algorithm. It also indicates that crossover can disrupt the schemata. This may seem counter-intuitive, because it is from the crossover operator that Genetic Algorithms are intended to derive much of their capability [47]. This issue is addressed by the Royal Road theory of Genetic Algorithms, which clarifies the important role of crossover.

3.2 Schema Theory for Test Data Generation by Genetic Algorithms

Evolutionary Testing typically does not use binary Genetic Algorithms, so the schema theory is not directly applicable. A new form of schema theory for Evolutionary Testing therefore has to be constructed. Fortunately, the input vectors used in Evolutionary Testing can be captured by a generalization of Holland's schema theory. These Evolutionary Testing schemata arise from the constraints on the input that a branch-covering solution must satisfy. The constraints can be defined naturally in terms of the computation of fitness for the approach level and branch distance computation. For example, suppose a program has three inputs, x, y and z and that in order to execute the branch under test B, the program must first follow a branch B_1, for which the condition $x > y$ must hold, and must then follow a branch B_2, for which the condition $y = z$ must hold.

In this example, a chromosome is a triple; three genes, one each for the input values of x, y and z. A schema is a constraint, denoting all instantiations of input vectors that satisfy the constraint. For example, two possible schemata $\{(x, y, z) \mid x > y\}$ and $\{(x, y, z) \mid x = y\}$. Of these two schemata, the first has higher fitness than the second for branch B, because all instantiations of the first have a higher fitness than all instantiations of the second due to their superior

fitness for the 'approach level'. Observe that this formulation of schemata is simply a generalization of the traditional schemata, because traditional schema can always be denoted by a corresponding constraint-schema. For example, Holland's traditional schema $1*01$ can be denoted by $\{(a,b,c,d) \mid a = 1 \wedge c = 0 \wedge d = 1\}$.

For Evolutionary Testing, a schema is thus denoted by a constraint $\mathbf{c}$, whose fitness is the average fitness of all instantiations that satisfy the constraint. To distinguish traditional schemata from those defined by a constraint, the latter shall be referred to as constraint-schemata. The fitness of a constraint-schema $\mathbf{c}$ is:

$$\overline{f}(\mathbf{c}) = \frac{1}{\mid \{y \mid \mathbf{c}(y)\} \mid} \sum_{x \in \{y \mid \mathbf{c}(y)\}} f(x)$$

and the approximate fitness of a constraint-schema $\overline{f}(\mathbf{c}, K)$ for a set of chromosomes K, is:

$$\overline{f}(\mathbf{c}, K) = \frac{1}{\mid \{x \mid \mathbf{c}(x) \wedge x \in K\} \mid} \sum_{\mathbf{c}(x) \wedge x \in K} f(x)$$

The basic form of the schema theory (without mutation and crossover), for Evolutionary Testing with respect to a constraint-schema $\mathbf{c}$ of a population of size M can now be defined in the same way as that for traditional schemata. That is:

$$N(\mathbf{c}, g+1) \geq N(\mathbf{c}, g) \frac{\overline{f}(\mathbf{c}, P(g))}{\frac{1}{M} \sum_{x \in P(g)} f(x)}$$

The full form of the schema theory, taking account of crossover and mutation, can also be formulated by defining the order of a constraint-schema, $o(\mathbf{c})$ to be the number of input variables that participate in the definition of the constraint.

For example, the order of $\{(x, y, z) \mid x > y\}$ is 2, while the order of $\{(x, y, z) \mid x = y = z\}$ is 3 and the order of $\{(x, y, z) \mid x > 17\}$ is 1.

Evolutionary Testing uses discrete recombination, in which each gene of each parent has an equal chance of being copied to the offspring. An upper bound on the probability of discrete recombination disrupting a constraint schema is thus the product of the probability of crossover occurring (p_c) and the ratio of genes in the constraint schema to total genes.

In Evolutionary Testing, mutation is typically applied to a single gene through the addition of randomly chosen values.

An upper bound on the probability that this form of mutation will disrupt a constraint-schema is simply the product of the probability of a gene mutation and the order of the constraint schema. With these two observations, it is possible to formally define the schema theory for constraint-schemata as follows:

$$N(\mathbf{c}, g+1) \geq N(\mathbf{c}, g) \frac{\overline{f}(\mathbf{c}, P(g))}{\frac{1}{M} \sum_{x \in P(g)} f(x)} (1 - p_c \frac{o(\mathbf{c})}{\lambda} - p_m \; o(\mathbf{h}))$$

However, as with the traditional schema theory, this schema theory of Evolutionary Testing also indicates that mutation and crossover disrupt constraint schemata and so it is necessary to consider the Royal Road theory, which explains the form of search problems for which the crossover will be most likely to succeed.

3.3 The Genetic Algorithm Royal Road

Mitchell, Forrest and Holland [47] introduced the theoretical study of Royal Road landscapes in order to capture formally the intuition underlying the 'folk theorem' that Genetic Algorithm will outperform a local search such as Hill Climbing, because of the way in which Genetic Algorithm uses the crossover operation to combine building blocks. Building blocks are fit schemata that can

be combined together to make even fitter schemata.

It is widely believed that the combination of building blocks through crossover (recombination of genetic material) is the primary underlying mechanism through which evolutionary progress is achieved. This applies both in the world of evolution by natural selection and through the Genetic Algorithm's mimicry of this natural process. The Royal Road theory of Genetic Algorithm aims to explain how this process works. In so doing, it captures a set of fitness functions (the so–called Royal Road functions) for which a Genetic Algorithm is well suited and for which it is theoretically predicted to equal or outperform other search techniques, such as local search.

The Royal Road theory addresses the perplexing aspect of the schema theory; the way in which it indicates that crossover could be viewed as a harmful operation that disrupts fit schema.

Mitchell *et al.* defined an example fitness function in terms of a set of schemata $\{s_1, \ldots, s_{15}\}$ as follows:

$$F(x) = \sum_{s \in S} c_s \sigma_s(x)$$

Where $F(x)$ is the fitness of a bit string x, c_s is the order of the schema s, and $\sigma_s(x)$ is one if x is an instantiation of s, and zero otherwise. The schemata $\{s_1, \ldots, s_{15}\}$ are defined in Figure 6. Notice how the Royal Road example is constructed so that lower fitness schemata can be combined to yield higher fitness schemata. Such a landscape is 'tailor made' to suit a Genetic Algorithm; crossover allows the Genetic Algorithm to follow a tree of schemata that lead directly to the global optimum. This tree of ever fitter schemata form the 'Royal Road'.

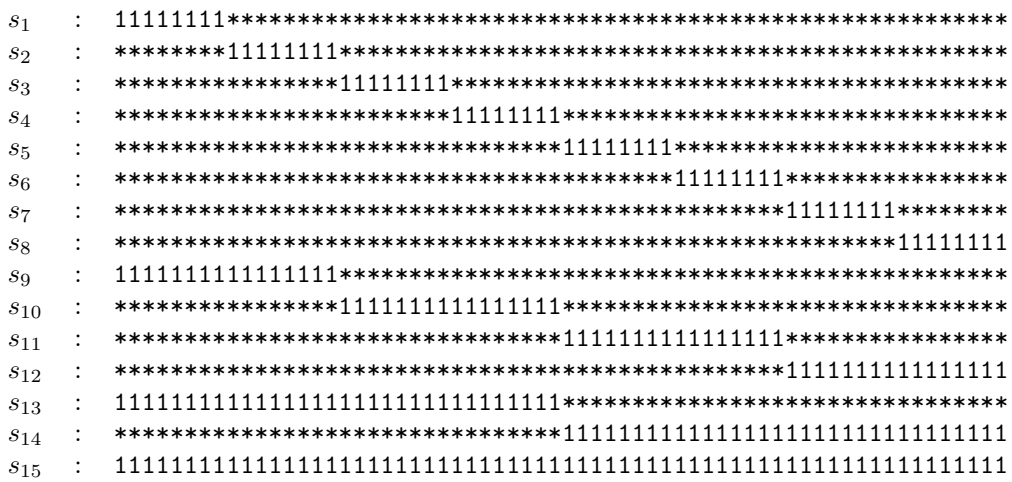


Figure 6: Royal Road Function of Mitchell *et al.* [47]. As i increases, all instances of the schema S_i have a higher fitness, until ultimately, the global optimum for this optimization problem is S_{15}, which has maximum possible fitness. The genetic crossover operation is therefore highly likely to combine chromosomes containing two partially fit schemata to create a chromosome with higher fitness. In this instance, crossover tends to combine lower fitness schemata to create higher fitness schemata, rather than disrupting the lower fitness schemata; the crossover operator has a 'Royal Road' to guide it from lower to higher fitness. This Royal Road property of the schemata of a set of instances was used as an archetype of the Royal Road by Mitchell *et al.*

3.4 Royal Road for Evolutionary Test Data Generation

For a schema concerned with constraints, the order of the schema is the number of variables that are mentioned in the constraint. In order for lower order schemata to be combined with higher order schemata, as with the binary Genetic Algorithm model, the higher order schemata must contain the union of the genes of the lower order schemata. Also, to avoid destroying the properties of a lower order schema, there must be no intersection of genes in the lower order schemata, otherwise the genes of one would overwrite those of the other when combined. This is also the case with the Royal Road theory of Mitchell *et al.*

However, since the genes in the chromosomes for Evolutionary Testing are input variables and the schemata denote constraints on these variables, there is an additional property that can be seen to hold for Evolutionary Testing Royal Road functions. The higher order and fitter schema will respect both the constraints respected by the lower order schemata (since it will contain the same values for genes of each of the lower order schemata). Therefore, the constraint of the higher order schemata must respect a conjunction of the constraints of the lower order schemata. That is, if two constraint schema c_1 and c_2 are combined to produce a fitter schema C then $C \Rightarrow c_1 \wedge c_2$.

This observation indicates that there must exist a tree of logical implications along any Royal Road of constraint schemata for Evolutionary Testing. Since the constraint schema theory is merely a generalization of the standard Holland schema theory, it can also be shown that (trivially) such a tree of constraints also holds for the Royal Road of Mitchell *et al.* For example, the constraint that denotes the Mitchell schema s_1 is $\forall i.1 \leq i \leq 8.s_1(i) = 1$, while the constraint denoted by Mitchell's s_2 is $\forall i.9 \leq i \leq 16.s_1(i) = 1$. Clearly the conjunction of these two constraints yields the constraint for Mitchell's s_9, namely, $\forall i.1 \leq i \leq 16.s_1(i) = 1$.

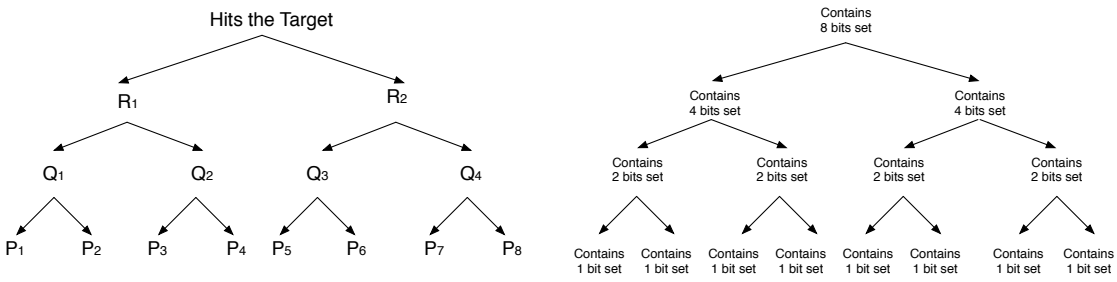
This can be extended to any implication, creating a relationship between subschemas and logical implication. For instance, $\{(x, y, z) \mid x = 5 \wedge y = 4 \wedge z = 3\}$ is a subschema of $\{(x, y, z) \mid x > y > z\}$. That is, all instances of $\{(x, y, z) \mid x = 5 \wedge y = 4 \wedge z = 3\}$ are also instances of $\{(x, y, z) \mid x > y > z\}$ (but not vice versa). In general, if P and Q are predicates over an alphabet X and P implies Q then $\{X|P\}$ is a subschema of $\{X|Q\}$.

These constraint-schemata have subschemata in the same way that traditional schemata also have subschemata. For instance, the traditional schema $1 * * * 111$ is a subschema of $1 * * * * * *$; all instances of $1 * * * 111$ are also instances of $1 * * * * * *$.

The implication is that for an Evolutionary Testing approach to exhibit a Royal Road property, the more fit schemata must be expressed as conjunctions of lower order schemata (possibly involving disjoint sets of input variables). Where this property holds, the Evolutionary Testing Royal Road theory predicts that Evolutionary Testing will perform well and that it will do so because of the presence of the crossover operation and the way in which fitter schemata are given exponentially more trials than less fit schema.

Consider Figure 7, which shows this principle as general property (Figure 7(a)) and also gives a concrete example (Figure 7(b)). The code in Figure 8 contains a target branch for which the optimization problem has the Royal Road illustrated by Figure 7(b).

The problem is the simple one of determining whether all of the bits of a bitset of 8 bits are set. This is achieved with a count. Clearly, if two or more bits are set this implies that one or more bits are set and if 4 or more bits are set then this implies that two or more are set. This problem is a transliteration of the Royal Road function of Mitchell *et al.* into code for optimization in Evolutionary Testing.



(a) General Case (b) Specific Case

Figure 7: How implication on constraint schema creates Royal Roads in Evolutionary Testing. In the left hand figure, (a), the predicates have the implication relations indicated by the arrows, for instance, $Q_i \Rightarrow P_i \wedge P_{i-1}, 1 \leq i \leq 4$. In the right hand figure, (b), the problem is to determine how many of the 8 bits of a byte are set. Once again, the arrows indicate the direction of logical implication.

```
...
for (count=0, i=0; i<8; i++)
{
  if (bitset[i]==1)
  {
      count++;
  }
}
if (count==8)
{
    /* target */
}
...
```

Figure 8: Code snippet for the number of set bits test problem, used to illustrate Royal Roads in Figure 7(a).

Observe that crossover has a fairly good chance of making a fitter schemata from less fit schemata. That is, solutions with a single bit set are fitter than those with no bits set and so they are more likely to be selected. Furthermore, combining two chromosomes, each with a single bit set, has a good chance of producing offspring with two or more bits set. Offspring with two or more bits set are fitter than their parents with only one bit set. Combining chromosomes with two bits set is more likely to produce offspring with more than two bits set than, for example, combining those with fewer than two bits set, and so on.

The archetype of this Royal Road property for testing is a predicate, the outcome of which is determined by a set of values S. Such a predicate will have a Royal Road fitness function if it tests for the presence of properties exhibited by non–intersecting subsets of S. This situation arises, for example, in string processing, where substrings are tested for the presence of certain properties of interest and in numeric array processing, where the program aims to establish whether subsets of the array are related in certain ways. It remains an open question as to how often this situation arises in practice. An extremely wide-ranging study (or perhaps set of studies) would be required to answer this question. However, the authors were able to find only a handful of example branches in those studied here which exhibited this Royal Road property.

4 Empirical Study

An empirical study was performed using 9 different programs, two of which were provided by DaimlerChrylser, whilst the other seven are open source. From these programs, 38 functions were studied, containing 760 branches (Table 1).

The minimum and maximum values of the input variables, along with their precision in the case of floating point variables, were specified for each function for the search process.

From this information, the input domain size (i.e. the search space size) can be computed. Input domain sizes range from 10^5 to 10^{524}. Each separate branch denotes a separate search problem, for which the size of the search space is one of the more important factors in determining the difficulty of the problem. Furthermore, the code analyzed was by no means trivial, containing many examples of complex, unstructured control flow, unbounded loops and computed storage locations in the form of pointers and array access.

Input domain sizes differ in some cases from those appearing for the same test objects used in [27], because of differences in minimum and maximum variable value information supplied to the search process. This had no effect on the conclusions of the study.

`bibclean-2.08` is an open source program used to syntax check and pretty-print BibTeX bibliography files. The two functions tested are validity checks for ISBN (as already seen in Figure 1) and ISSN codes used to identify publications. The programs `f2` and `defroster` are industrial case studies provided by DaimlerChrysler. An S–Class Mercedes car has over 80 such embedded controllers, which, taken together represent approximately 0.5GB of object code. The two systems used in this study are production code for engine and rear window defroster control systems. The code is machine generated from a design model of the desired behaviour. `eurocheck-0.1.0` is an open source program. It contains a single function used to validate serial numbers on European bank notes. `gimp-2.2.4` is the open source GNU image manipulation program. Several library functions were tested, including routines for conversion of different colour representations (for example RGB to HSV) and the manipulation of drawable objects. `space` is a program from the European Space Agency and is available from the Software-artifact Infrastructure Repository [1, 16]. Nine functions were tested. `spice` is an open source general purpose analogue circuit simula-

tor. Two functions were tested, which were clipping routines for the graphical front-end. `tiff-3.8.2` is a library for manipulating images in the Tag Image File Format (TIFF). Functions tested include image placing routines and functions for building 'overview' compressed sample images. `totinfo` is a program created by Siemens, which like `space` is also drawn from the Software-artifact Infrastructure Repository.

Where the type signature of the function was straightforward, the numerical vectors generated by the search could be used directly as input vectors. In other cases, the input values had to be mapped into structure types. Linked lists and arrays, where used, were fixed in length. This is not a limitation of search based test data generation in general, as approaches have been developed to generate variable-length dynamic data structures [35].

The `addscan` function of `space` is responsible for allocating memory, but not deallocating it, leading to potential memory leaks in the testing process. Therefore, the `malloc` function had to be overridden to keep track of the pointers allocated so that the test execution process could release the memory afterwards. These modifications affect neither the size of the search space nor the distribution of fitness values so they have no impact upon the research questions.

The empirical study was conducted in two parts. The first part focuses on global and local search only, *i.e.* Evolutionary Testing and Hill Climbing, using Random Testing to identify the non-trivial branches that only these searches can generate test data for. Having studied global and local search, and having found non-trivial search problems, the second part of the study conducts experiments using the hybrid Memetic Algorithm approach, comparing the results with those of Evolutionary Testing and Hill Climbing.

Table 1: Test object details

Test Object / Function	Lines of code	Number of branches	Approximate domain size imposed (10^x)
bibclean-2.08			
check_ISBN		42	72
check_ISSN		42	72
Total	178	84	
defroster			
Defroster_main		56	24
Total	250	56	
f2			
F2		24	54
Total	418	24	
eurocheck-0.1.0			
euchk		22	31
Total	70	22	
gimp-2.2.4			
gimp_hsv_to_rgb		16	21
gimp_hsv_to_rgb_int		16	7
gimp_hsv_to_rgb4		16	16
gimp_hwb_to_rgb		18	17
gimp_rgb_to_hsl		14	20
gimp_rgb_to_hsl_int		14	7
gimp_rgb_to_hsv		10	20
gimp_rgb_to_hsv4		18	7
gimp_rgb_to_hsv_int		14	7
gradient_calc_bilinear_factor		6	34
gradient_calc_conical_sym_factor		8	31
gradient_calc_conical_asym_factor		6	31
gradient_calc_linear_factor		8	31
gradient_calc_radial_factor		6	21
gradient_calc_spiral_factor		8	37
gradient_calc_square_factor		6	21
Total	867	184	
space			
addscan		32	519
fixgramp		8	23
fixport		6	125
fixselem		8	125
fixsgrel		68	524
fixsgrid		22	101
gnodfind		4	70
seqrotrg		32	206
sgrpha2n		16	451
Total	2210	230	
spice			
cliparc		64	44
clip_to_circle		42	23
Total	269	106	
tiff-3.8.2			
TIFF_SetSample		14	10
TIFF_GetSourceSamples		18	15
PlaceImage		16	38
Total	182	48	
totinfo			
gser		6	10
InfoTbl		30	19
LGamma		4	5
Total	319	40	
Grand Total	4763	760	

4.1 Empirical Study Part 1: Global and Local Search

Test data generation experiments for branch coverage were performed using Evolutionary Testing, Hill Climbing and Random Testing. The Evolutionary Testing and Hill Climbing algorithms were described in the previous section. The Random Testing algorithm simply constructs valid random inputs until the required test data is found, or the stopping criterion is met.

For each algorithm, the test data search for each branch was terminated after the evaluation of 100,000 inputs if the required test data had not been found. This was repeated 60 times using a fixed list of different seeds for random number generation. Two metrics were then computed from the results. The 'success rate' (SR) for each branch and search method is the percentage of the 60 runs that the branch was successfully covered. The 'average number of evaluations' (AE) is the average number of test object executions (i.e. fitness evaluations) that were performed in order to find the test data for each successful run.

Figure 9 summarises the branches covered by the different search techniques. For each search, a branch is counted as 'covered' if test data was found on at least one of the sixty trials. Of the 760 branches, 634 branches (83%) were covered by all search techniques, including Random Testing. Of these 634 branches, every branch was covered more than once, *i.e.* not merely by 'pure luck', with 575 covered with a 100% success rate, *i.e.* on every trial. This high degree of coverage for simple-minded Random Testing tends to support the view that it can be effective for easy-to-cover branches, leaving relatively few hard-to-cover branches for which more intelligent search may be required.

37 branches were covered by either Evolutionary Testing and Hill Climbing. Evolutionary Testing exclusively covered 9 branches whilst Hill Climbing exclusively covered 8. Perhaps surprisingly, Random Testing covered 4 branches that the other methods were unable to. These branches were covered with a low suc-

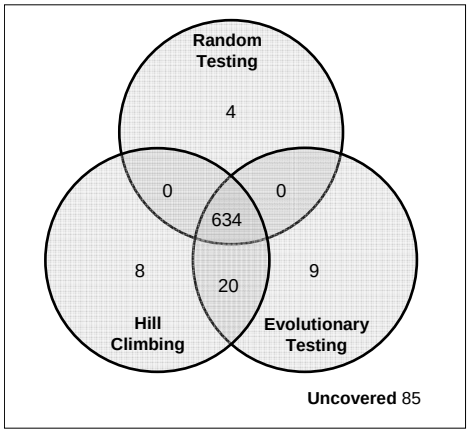


Figure 9: Venn diagram showing the numbers of branches covered by each different search method

cess rate (18% or lower). Closer examination of the fitness landscapes revealed local optima in which metaheuristic approaches became stuck. Because random search is not guided by a fitness function, this did not present a problem, and on a small number of occasions was able to find the required test data by chance. The final 85 branches were infeasible or simply not covered by any of the search techniques. In many instances, the fitness function surface is flat, affording the search no guidance to the required test data. Also, because the target-covering test inputs occupy a tiny portion of the input domain, Random Testing also fails. Such 'difficult' fitness landscapes which are flat or contain local optima are studied further elsewhere in the literature [24, 43, 44, 42]. Clearly there is no point in attempting to use an 'intelligent' (and therefore more computationally expensive) metaheuristic search when Random Testing will do. Thus in answering the research questions, only branches not covered by Random Testing are considered.

The first part of the empirical study addresses three research questions, described below:

Research Question 1: Validation of Evolutionary Testing theory

For a predicate, the fitness function of which denotes a Royal Road function, the theory predicts that the Genetic Algorithm should perform well. Does it perform well and how does it compare to a Hill Climbing algorithm?

Research Question 2: Validation of crossover hypothesis

According to the theory, the reason for Evolutionary Testing's good performance on Royal Road functions should be due to the effect of the crossover operator. Therefore, there is a second 'validation of theory' question: How does Evolutionary Testing perform on Royal Road functions when the effects of the crossover operator are removed?

Research Question 3: Performance for non-Royal Road branches

For predicates that do not have a Royal Road fitness function, Evolutionary Testing may perform no better, and possibly worse than Hill Climbing. The theory is concerned with effectiveness not efficiency and so it cannot make predictions about how Evolutionary Testing will perform relative to Hill Climbing, nor how badly its performance would be affected by the absence of Royal Roads. However, this remains an important question and one that can be addressed empirically.

4.2 Answers to Research Questions

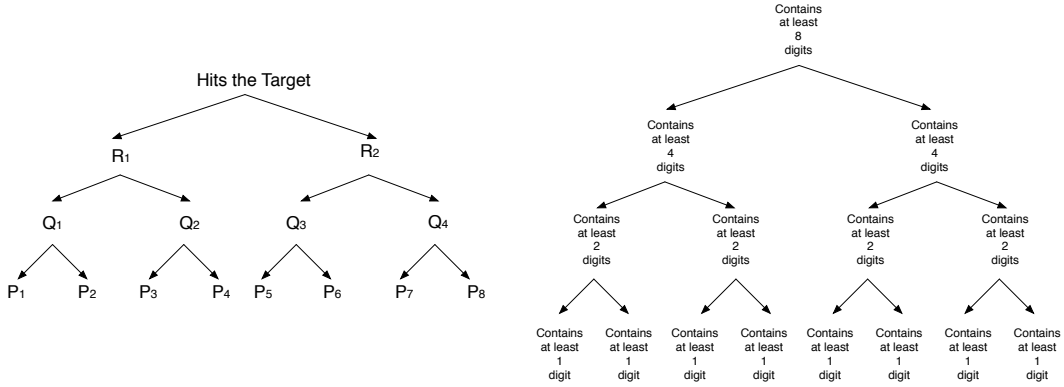
Research Question 1: Validation of Evolutionary Testing theory

The identification of Royal Road functions was a test, necessarily performed by hand for each predicate, because the decision as to whether a particular predicate denotes a Royal Road is one determined by a deep understanding of the semantics of the predicate in question.

The Royal Road property was found in branches of the `bibclean` test object. The `check_ISBN` (Figure 1) and `check_ISSN` functions both read a string of 30 characters. The function sequentially searches through the characters in order to find those valid for an ISBN or ISSN number. When such a character is found, a counter variable is incremented. When this counter is equal to 10 (`check_ISBN`) or 8 (`check_ISSN`), validation can take place. The constraint schemata for this program form a Royal Road, where for example, the constraint 'contains at least 3 valid characters' subsumes the constraints: 'contains at least 2 valid characters' and 'contains at least 1 valid character'.

There are 256 different character values, of which 12 are valid, giving a 12/256 chance that a character will be valid. Therefore, a string of 30 characters is likely to contain at least one valid character. A template string with some valid characters denotes a schema; the more valid characters, the fitter the schema. According to the schema theory, these schemata will receive ever more evaluations as the algorithm progresses and, according to the Royal Road theory, their recombination through crossover is likely to yield super-fit offspring. Thus, the theory developed in Section 3 predicts that Evolutionary Testing will perform well for this example. This situation is too large to depict effectively in a diagram like those used to illustrate Royal Roads in Figure 7. However, it is possible to depict the same problem, simply using a slightly smaller scale, thereby illustrating the way in which the `check_ISBN` contains a predicate that denotes a Royal Road for Evolutionary Testing. Consider the scaled down version of the optimization problem denoted by branch 23F of the `check_ISBN` function depicted in Figure 10.

The problem is to determine whether a string contains at least 8 digits. If the input string satisfies this property then branch 23F will be followed. If a string contains at least two digits then it certainly contains at least one digit.



(a) General Case (b) Example from the `bibclean` test object

Figure 10: How implication on constraint schema creates Royal Road. In the left hand figure (a) is the abstract general case from Figure 7(a). In the right hand figure (b), the problem depicted is a scaled down version of the problem from branch 23F of the `check_ISBN` function of the `bibclean` test object, the code of which is contained in Figure 1.

If it contains at least four then it certainly contains at least 2 and so on. This explains the direction of the implication arrows in Figure 10. Observe that combining two strings that contain at least one digit is more likely to produce a string that contains at least two digits than combining string that contain no digits (or one that contains none and one that contains a single digit).

Therefore, since single-digit containing strings are fitter than no-digit containing strings and these are more likely to combine to make two-digit containing strings, which are, in turn, more likely to combine to make strings with even more digits, the optimization problem has the Royal Road property. This property strongly favours the crossover operation of the Genetic Algorithm approach.

The empirical results (presented in Table 2) support this prediction. Evolutionary Testing almost always succeeds in finding test data, whilst Hill Climbing always fails. By contrast, Hill Climbing gets stuck along plateaux appearing in the fitness landscape for branches depending on code validation. If an invalid character, c is generated, exploratory moves are unlikely to result in an improve-

Table 2: Evolutionary Testing and Hill Climbing for branches with Royal Road properties. SR is the success rate. AE is the average number of fitness evaluations for successful trials (*i.e.* where test data was found to execute the branch). SD is the standard deviation of the fitness evaluations for successful trials.

Test Object (Branch ID)	**Evolutionary Testing**			**Hill Climbing**		
	SR	AE	SD	SR	AE	SD
bibclean-2.08						
check_ISBN (23F)	95%	7,986	4,338	0%	*n/a*	*n/a*
check_ISBN (27T)	95%	7,986	4,338	0%	*n/a*	*n/a*
check_ISBN (29T)	95%	8,001	4,337	0%	*n/a*	*n/a*
check_ISBN (29F)	95%	9,103	4,497	0%	*n/a*	*n/a*
check_ISSN (23F)	98%	5,273	1,672	0%	*n/a*	*n/a*
check_ISSN (27T)	98%	5,273	1,672	0%	*n/a*	*n/a*
check_ISSN (29T)	98%	5,324	1,666	0%	*n/a*	*n/a*
check_ISSN (29F)	98%	6,380	1,926	0%	*n/a*	*n/a*

ment in fitness, unless c happens to be adjacent to a block of valid characters.

Research Question 2: Validation of crossover hypothesis

The recombination operator was disabled, and the Evolutionary Testing experiment re-run for the `bibclean` test object. Test data generation failed in almost every instance (Table 3). In order to rule out the possibility that recombination was not just adding further 'mutation', and that true crossover was not really having an effect, a third experiment was carried out. This time, parents were recombined with a randomly generated individual, rather than with another parent drawn from the current population. This is a form of the so-called 'Headless Chicken Test' [31], classically used to identify whether crossover is actively contributing in finding a solution to a search problem.

The outcome was as follows. For four branches, test data could not be generated using the randomly generated second parent, as seen in Table 4.

For `check_ISSN` branch 29F, test data was generated on only a few occurrences, and for branches of the `check_ISBN` function, only during one trial. Test data was generated for the remaining branches with reasonable consistency, but with almost ten times as many fitness evaluations. Statistical significance was

Table 3: Comparing Evolutionary Testing with and without crossover for branches with Royal Road Properties. SR is the success rate. AE is the average number of fitness evaluations for successful trials (*i.e.* where test data was found to execute the branch). SD is the standard deviation of the fitness evaluations for successful trials.

Test Object	**With Crossover**			**Without Crossover**		
(Branch ID)	SR	AE	SD	SR	AE	SD
bibclean-2.08						
check_ISBN (23F)	95%	7,986	4,338	0%	*n/a*	*n/a*
check_ISBN (27T)	95%	7,986	4,338	0%	*n/a*	*n/a*
check_ISBN (29T)	95%	8,001	4,337	0%	*n/a*	*n/a*
check_ISBN (29F)	95%	9,103	4,497	0%	*n/a*	*n/a*
check_ISSN (23F)	98%	5,273	1,672	2%	45,219	0
check_ISSN (27T)	98%	5,273	1,672	2%	45,219	0
check_ISSN (29T)	98%	5,324	1,666	2%	45,219	0
check_ISSN (29F)	98%	6,380	1,926	2%	56,007	0

tested using a one-tailed Wilcoxon rank sum test in favour of the set of average fitness evaluations using normal parents, with the confidence level set at 0.99. In all cases where the comparable sample sizes were of sufficient size, the test revealed that the results were statistically significant. The p-values appear in Table 4.

It can be concluded therefore, that crossover is having an effect, and is not merely a source of further mutational-like effects on the search.

Research Question 3: Performance for non-Royal Road branches

Figure 11 and Table 5 record data with respect to the 29 branches covered by Evolutionary Testing or Hill Climbing and which do not exhibit Royal Road properties. The finding is that Hill Climbing significantly outperforms Evolutionary Testing in many of these cases. This is a surprising finding given the high degree of attention paid to Evolutionary Testing, compared to the simpler Hill Climbing approach.

Figure 11 shows non-Royal Road branches which were not covered 100% of the time by both techniques. Out of these 17 branches, Hill Climbing achieves the highest success rate score on 15 occasions, although 7 are only covered on a

Table 4: Evolutionary Testing using crossover with normal parents versus crossover using a randomly generated second parent that is not a member of the current population. SR is the success rate. AE is the average number of fitness evaluations for successful trials (*i.e.* where test data was found to execute the branch). SD is the standard deviation of the fitness evaluations for successful trials. The Observed Significance Level is the p-value produced by a Wilcoxon rank sum test with confidence level set at 0.99. The test is one-sided to discover whether using normal parents requires significantly fewer fitness evaluations to find test data

Test Object (Branch ID)	**Normal Parents**			**Random 2nd Parent**			**Observed Significance Level**
	SR	AE	SD	SR	AE	SD	
bibclean-2.08							
check_ISBN (23F)	95%	7,986	4,338	2%	41,701	0	*n/a*
check_ISBN (27T)	95%	7,986	4,338	2%	41,701	0	*n/a*
check_ISBN (29T)	95%	8,001	4,337	0%	*n/a*	*n/a*	*n/a*
check_ISBN (29F)	95%	9,103	4,497	2%	41,701	0	*n/a*
check_ISSN (23F)	98%	5,273	1,672	78%	58,998	19,036	0.00
check_ISSN (27T)	98%	5,273	1,672	78%	58,998	19,036	0.00
check_ISSN (29T)	98%	5,324	1,666	77%	59,944	18,650	0.00
check_ISSN (29F)	98%	6,380	1,926	20%	71,108	19,373	*n/a*

few trials, with a success rate below 15%. Evolutionary Testing scores a higher success rate for branch 20T of the function `PlaceImage`, but the success rates for both techniques are low (20% or lower).

There is only one branch for which Evolutionary Testing is successful and for which Hill Climbing fails on each trial. This is branch 12T of the function `gimp_hwb_to_rgb`, which has a fitness landscape containing a series of plateaux, and is thus hard for Hill Climbing to navigate. One specific value is required for one input variable, which happens to be the top value of its range. It is only covered by Evolutionary Testing due to an artifact in the way the mutation operator works. If a large value is added to the variable, such that it goes out of range, the value is reset to its maximal value. Thus the branch is covered.

The greater expense of the Evolutionary Testing approach is also clearly revealed by the remaining cases (for which both Evolutionary Testing and Hill Climbing achieve coverage of the target branch with a 100% success rate). These branches are recorded in Table 5. For these 12 branches, Hill Climbing is more

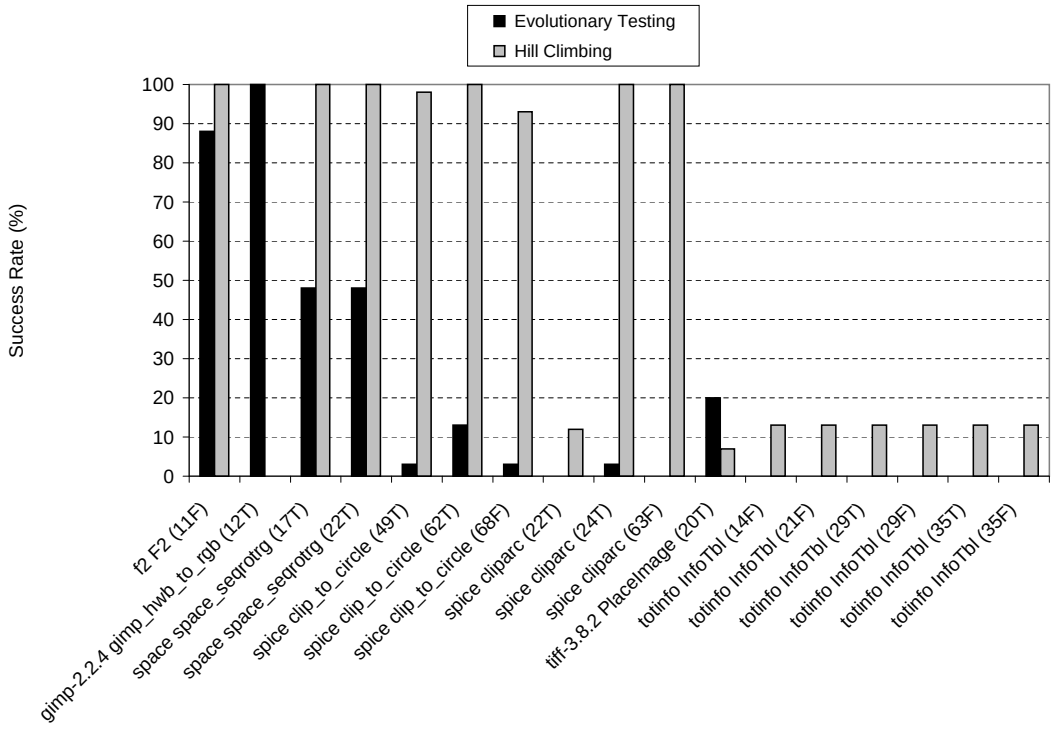


Figure 11: Non Royal Road branches not covered with a 100% success rate by both Evolutionary Testing and Hill Climbing

efficient by an order of a magnitude. Statistical tests were again performed using a one-tailed Wilcoxon rank sum test with a confidence level of 0.99, this time in favour of the sample of average fitness evaluations using Hill Climbing. The results for Hill Climbing were significant in each case. The p-values are recorded in the table.

4.3 Empirical Study Part 2: Hybrid Search

In the second part of the empirical study, the Memetic Algorithm described in Section 2.3 was run 60 times for each branch for each of the test objects used in part one. Three further research questions are addressed, as described below.

Table 5: Branches not exhibiting Royal Road properties covered with a 100% success rate by Evolutionary Testing and Hill Climbing. AE is the average number of fitness evaluations for successful trials (*i.e.* where test data was found to execute the branch). SD is the standard deviation of the fitness evaluations for successful trials. The Observed Significance Level is the p-value produced by a Wilcoxon rank sum test with confidence level set at 0.99. The test is one-sided to discover whether the use of Hill Climbing requires significantly fewer fitness evaluations to find test data

Test Object (Branch ID)	**Evolutionary Testing**		**Hill Climbing**		**Observed Significance Level**
	AE	SD	AE	SD	
gimp-2.2.4					
gimp_rgb_to_hsl (4T)	11,183	2,181	120	20	0.00
gimp_rgb_to_hsv (5F)	7,679	1,710	106	20	0.00
gimp_rgb_to_hsv4 (11F)	4,911	1,607	244	135	0.00
gimp_rgb_to_hsv_int (10T)	4,911	1,607	244	135	0.00
gradient_calc_bilinear_factor (8T)	13,970	3,370	210	41	0.00
gradient_calc_conical_asym_factor (3F)	20,629	4,560	208	32	0.00
gradient_calc_conical_sym_factor (3F)	20,629	4,560	208	32	0.00
gradient_calc_spiral_factor (3F)	21,599	4,751	224	34	0.00
spice					
cliparc (13F)	10,421	3,035	430	527	0.00
cliparc (15T)	11,172	3,413	996	1,388	0.00
cliparc (15F)	11,022	3,202	1,048	1,098	0.00
tiff-3.8.2					
TIFF_SetSample (5T)	8,482	2,730	87	51	0.00

Research Question 4: Subsumption of Evolutionary Testing and Hill Climbing by Hybrid Memetic Algorithm Approach

The Memetic Algorithm combines Evolutionary Testing and Hill Climbing. Therefore, can it cover all the branches that Evolutionary Testing and Hill Climbing can? In terms of coverage, therefore, one would expect the Memetic Algorithm to be the best overall performer. Is this the case?

Research Question 5: Performance for Royal Road branches

The Memetic Algorithm, being derived from the Evolutionary Algorithm, incorporates a population (albeit a smaller one) and the use of the same crossover operator as Evolutionary Testing. Is the performance of the Memetic Algorithm similar with Royal Road branches as for Evolutionary Testing?

Research Question 6: Performance for non-Royal Road branches

The Memetic Algorithm incorporates a phase of Hill Climbing. Therefore it should improve on Evolutionary Testing, offering similar levels of efficiency on these branches as with Hill Climbing. Does it?

4.4 Answers to Research Questions

Research Question 4: Subsumption of Evolutionary Testing and Hill Climbing by Hybrid Memetic Algorithm Approach

Figure 12 shows the breakdown of these branches covered by the Memetic Algorithmm, Evolutionary Testing and Hill Climbing, but not covered by Random Testing. The Venn diagram shows that the Memetic Algorithm failed to cover any new branches, *i.e.* the branches depicted are the same 37 branches covered by either Evolutionary Testing or Hill Climbing discussed in the last section. The diagram shows that the Memetic Algorithm covers all except one of these 37 branches. This is branch 12T of the function `gimp_hwb_to_rgb`. This branch was discussed in the previous section and is covered only due to an artifact in the way the breeder algorithm mutation operator works, as used by Evolutionary Testing. Since the Memetic Algorithm uses uniform mutation instead, this branch was not covered.

The conclusion that can be drawn from the diagram is, however, that in terms of coverage, the hybrid Memetic Algorithm does offer the best overall performance.

Research Question 5: Performance for Royal Road branches

Table 6 compares the Memetic Algorithm against Evolutionary Testing for the Royal Road branches. The table shows that the Memetic Algorithm has a much poorer success rate with Royal Road branches when compared to Evolutionary

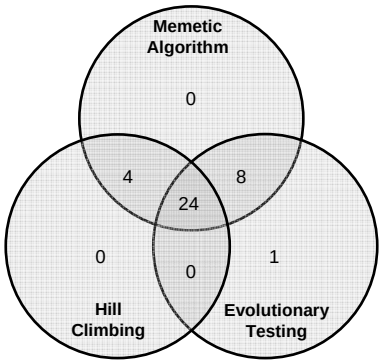


Figure 12: Branches covered by Evolutionary Testing, Hill Climbing or the Memetic Algorithm. The Memetic Algorithm was successful for all branches covered by Evolutionary Testing and Hill Climbing except for one pathological case. Branches not appearing in the Venn diagram were either covered by Random Testing or not covered at all

Testing. The Memetic Algorithm uses a much smaller population size of 20 compared to a size of 300 for Evolutionary Testing. Therefore population sizes of 50 and 300 were used with the Memetic Algorithm to see if this was the contributory factor to a poorer performance. However, the table shows that as the population size increases, the success rate of the Memetic Algorithm gets worse.

The reason for the poorer performance of the Memetic Algorithm is due to the fact that much of the 100,000 fitness evaluation budget is wasted in the Hill Climbing phase. The algorithm performs one hill climb per individual, performing fitness evaluations until a plateaux is reached. However, since the Royal Road branches are largely made up of fitness invariant plateaux, Hill Climbing does not help, and the effort is wasted. Increasing the population size will make matters worse because more fitness evaluations will be used on performing more hill climbs. The evidence for this reason comes from the table. A higher number of average fitness evaluations is required by the Memetic Algorithm to cover each branch when it is successful. This number increases as the population size

Table 6: Performance of the Memetic Algorithm compared to Evolutionary Testing for branches with Royal Road properties. SR is the success rate. AE is the average number of fitness evaluations for successful trials (*i.e.* where test data was found to execute the branch). SD is the standard deviation of the fitness evaluations for successful trials. The Observed Significance Level is the p-value produced by a Wilcoxon rank sum test with confidence level set at 0.99. The test is one-sided to discover whether the use of Evolutionary Testing requires significantly fewer fitness evaluations to find test data

Test Object (Branch ID)	**Evolutionary Testing**			**Memetic Algorithm**				**Observed Significance Level**
	SR	AE	SD	Pop. Size	SR	AE	SD	
bibclean-2.08								
check_ISBN (23F)	95%	7,986	4,338	20	18%	79,711	15,620	*n/a*
				50	7%	82,072	1,918	*n/a*
				300	0%	*n/a*	*n/a*	*n/a*
check_ISBN (27T)	95%	7,986	4,338	20	18%	79,711	15,620	*n/a*
				50	7%	82,072	1,918	*n/a*
				300	0%	*n/a*	*n/a*	*n/a*
check_ISBN (29T)	95%	8,001	4,337	20	18%	79,711	15,621	*n/a*
				50	7%	82,072	1,918	*n/a*
				300	0%	*n/a*	*n/a*	*n/a*
check_ISBN (29F)	95%	9,103	4,497	20	18%	79,866	15,707	*n/a*
				50	7%	82,139	1,932	*n/a*
				300	0%	*n/a*	*n/a*	*n/a*
check_ISSN (23F)	98%	5,273	1,672	20	58%	68,055	22,971	0.00
				50	42%	79,697	16,169	0.00
				300	0%	*n/a*	*n/a*	*n/a*
check_ISSN (27T)	98%	5,273	1,672	20	58%	68,055	22,971	0.00
				50	42%	79,697	16,169	0.00
				300	0%	*n/a*	*n/a*	*n/a*
check_ISSN (29T)	98%	5,324	1,666	20	58%	68,056	22,971	0.00
				50	42%	79,699	16,171	0.00
				300	0%	*n/a*	*n/a*	*n/a*
check_ISSN (29F)	98%	6,380	1,926	20	55%	66,747	22,299	0.00
				50	33%	75,605	15,480	0.00
				300	0%	*n/a*	*n/a*	*n/a*

increases. The difference with Evolutionary Testing is an order of magnitude. A one-sided Wilcoxon rank sum statistical test confirms that Evolutionary Testing is significantly more efficient at covering these branches in all cases. The p-values are listed in the table.

Research Question 6: Performance for non-Royal Road branches

Figure 13 and Table 7 record the performance of the Memetic Algorithm against Evolutionary Testing and Hill Climbing for non-Royal Road branches. In each case, the Memetic Algorithm is compared against the data for the previous best

performer for the branch in question. Figure 13 records branches which were not covered with a 100% success rate by both the Memetic Algorithm and the previous best performer. The figure shows that apart from branch 12T of the function `gimp_hwb_to_rgb`, the Memetic Algorithm has the same or a modestly improved success rate over the previous best performer.

Table 7 records branches which were covered with a 100% success rate by the Memetic Algorithm and the previous best performer, which was always Hill Climbing for the branches in question. The table shows that the Memetic Algorithm has a similar level of efficiency for these branches. This was confirmed by a Wilcoxon rank sum test, which was performed to see if there was any significant difference in the sets of fitness evaluations needed to find test data for each branch in each trial. The test recorded a significant difference in only 5 of the 18 cases, *i.e.* where the observed significance level was less than 0.01. For these 5 cases the Memetic Algorithm has a worse average on each occasion. However, the difference is relatively low; and only 12 evaluations on one occasion. The reason the test is significant is as follows. The branches in question are very easy to cover by Hill Climbing (and the Hill Climbing phase of the Memetic Algorithm) because the fitness landscape is extremely smooth, and thus only a small number of fitness evaluations are required. However the Memetic Algorithm must always set up the initial population first, which will on average require more fitness evaluations than Hill Climbing. This effect however is only noticeable for easy-to-cover branches, as branches with more complex landscapes will require restarts for Hill Climbing or the consideration of different individuals for the Memetic Algorithm, and thus the extra effort initially required of the Memetic Algorithm becomes insignificant.

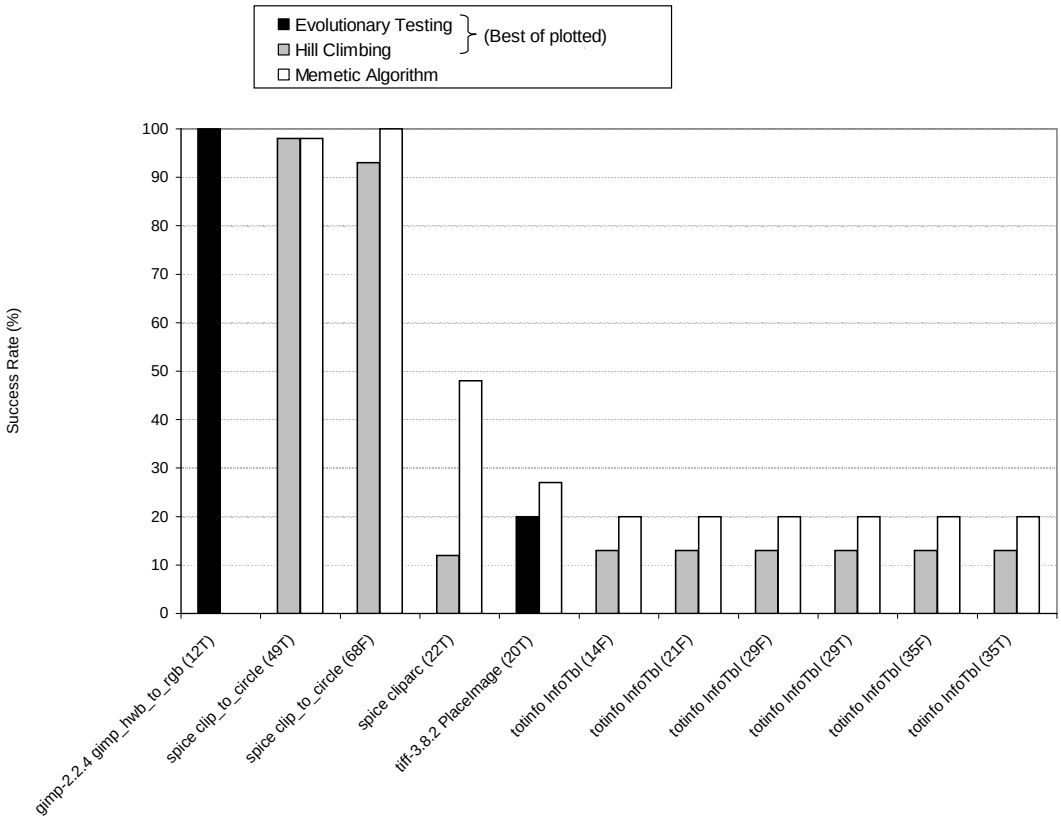


Figure 13: Non royal road branches not covered with a 100% success rate by all methods

Table 7: Memetic Algorithm compared with the best performer out of Evolutionary Testing and Hill Climbing for non Royal Road branches. Branches only appear if the success rate was 100% for the Memetic Algorithm and the best performer. AE is the average number of fitness evaluations and SD is the standard deviation. The Observed Significance Level is the p-value produced by a Wilcoxon rank sum test with confidence level set at 0.99. The test is two-tailed to discover if there is significant difference in the number of fitness evaluations required by Memetic Algorithm and the best performer to find test data for the particular branch in question

Test Object (Branch ID)	**Best of Evolutionary Testing / Hill Climbing**			**Memetic Algorithm**		**Observed Significance Level**
	Best	AE	SD	AE	SD	
f2						
F2 (11F)	Hill Climbing	3,491	3,846	1,728	1,399	0.03
gimp-2.2.4						
gimp_rgb_to_hsl (4T)	Hill Climbing	120	20	133	16	0.00
gimp_rgb_to_hsv (5F)	Hill Climbing	106	20	118	15	0.00
gimp_rgb_to_hsv4 (11F)	Hill Climbing	244	135	559	444	0.00
gimp_rgb_to_hsv_int (10T)	Hill Climbing	244	135	559	444	0.00
gradient_calc_bilinear_factor (8T)	Hill Climbing	210	41	198	37	0.17
gradient_calc_conical_asym_factor (3F)	Hill Climbing	208	32	207	34	1.00
gradient_calc_conical_sym_factor (3F)	Hill Climbing	208	32	207	34	1.00
gradient_calc_spiral_factor (3F)	Hill Climbing	224	34	226	36	0.80
space						
space_seqrotrg (17T)	Hill Climbing	14,684	8,173	11,630	6,356	0.09
space_seqrotrg (22T)	Hill Climbing	14,684	8,173	11,630	6,356	0.09
spice						
clip_to_circle (62T)	Hill Climbing	18,435	16,191	21,233	19,494	0.59
cliparc (13F)	Hill Climbing	430	527	290	285	0.08
cliparc (15T)	Hill Climbing	996	1,388	589	530	0.38
cliparc (15F)	Hill Climbing	1,048	1,098	715	680	0.10
cliparc (24T)	Hill Climbing	890	663	1,353	1,122	0.04
cliparc (63F)	Hill Climbing	16,558	16,235	10,942	12,320	0.04
tiff-3.8.2						
TIFF_SetSample (5T)	Hill Climbing	87	51	118	46	0.00

5 Threats to Validity

This section looks at the possible threats to the validity of the empirical study performed in this paper. The first issue to address is that of the *internal validity* of the experiments, *i.e.*, whether there has been a bias in the experimental design that could affect the causal relationship under study. The hypotheses studied in this paper concerned relationships between search based test data generation techniques and branches with Royal Road and non-Royal Road fitness functions. No automated decision procedure exists for finding Royal Roads. If there were, then automated test data generation would not be as hard as it is. Thus the identification of Royal Road properties was necessarily performed by hand for each predicate, taking into account the semantics of the program and predicate in question.

Another potential source of bias comes from the inherent stochastic behaviour of the metaheuristic search algorithms under study. The most reliable (and widely used) technique for overcoming this source of variability is to perform tests using a sufficiently large sample of result data. In order to ensure a large sample size, experiments were repeated 60 times. Furthermore, in order to establish superiority of one search algorithm over another, statistical tests were applied. To perform these tests, a set of results, *i.e.* the number of fitness evaluations required to cover a branch, are obtained from a set of runs. To show that one technique is superior to another, the Wilcoxon rank sum test is performed to see if there is a statistical significant difference in the means of each set of results. A non-parametric test was chosen 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.

A further source of bias includes the selection of the programs used in the empirical study, which could potentially affect its *external validity*; i.e. the extent to which it is possible to generalise from the results obtained. The rich and diverse nature of programs makes it impossible to sample a sufficiently large set of programs such that all the characteristics of all possible programs could be captured. 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. It should also be noted that the empirical study drew on 794 branches, making it one of the largest search based structural test data generation study to date, and providing a relatively large pool of results from which to make observations.

The results of the empirical study show that Evolutionary Testing is consistently the best performer on branches with Royal Road properties when compared to all the other search techniques. The results provide evidence indicating that simpler techniques outperform Evolutionary Testing on non-Royal Road branches, whilst also supporting the claim that in terms of coverage, the Memetic Algorithm is capable of the best overall performance. Nevertheless, 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. As with all such experimental software engineering, further experiments are required in order to replicate the results here.

6 Related Work

Miller and Spooner [45] were the first authors to dynamically generate test data, defining an objective function to be optimized using numerical maximization techniques. Korel [33] was the first to use the strategy described as Hill Climbing in this paper. Xanthakis *et al.* [60] were the first to apply evolu-

tionary computation to test data generation for the execution of paths. This work has been extended by various authors [39, 51, 55, 57] for branch coverage. Wang and Jeng [56] were the first to propose the use of Memetic Algorithms for Search Based Testing. In 2004 there was a sufficiently large body of work in Search Based Testing to warrant a detailed survey of the field [40]. However, despite this large volume of work, this is the first paper to provide a theoretical explanation of why and where evolutionary approaches work.

Many empirical studies in the literature compare Evolutionary Testing with Random Testing alone [39, 51, 57], finding that Evolutionary Testing achieves the highest levels of coverage, and more efficiently. Although this finding is important to validate the use of metaheuristic search for test data generation, it is something of a 'sanity check'; in any optimization problem worthy of study, the chosen technique should be able to convincingly outperform random search.

Other studies which do compare Evolutionary Testing with other metaheuristic search methods tend to report results on small numbers of programs, each with limited complexity. For instance, Wang and Jeng [56] compare Evolutionary Testing with Hill Climbing and Memetic Algorithms for branch coverage on six examples. Memetic Algorithms are found to outperform Hill Climbing, which in turn outperforms Evolutionary Testing. However, only a small number of branches are investigated, and none of these contain Royal Road properties. Mansour and Salame [38] compare Evolutionary Testing, Hill Climbing and Simulated Annealing for test data generation for path coverage, finding that Hill Climbing discovers test data faster than Evolutionary Testing and Simulated Annealing, but that Evolutionary Testing and Simulated Annealing can cover more paths. They also report that Simulated Annealing performs better than Evolutionary Testing. However, Hill Climbing is only applied to programs with integer inputs and the study is performed on eight functions of fewer than

86 lines of code. Finally Xiao *et al.* [61] compare Evolutionary Testing with Simulated Annealing for condition-decision coverage, finding that Evolutionary Testing is consistently the best performer. However, once again, the study is small scale, featuring test objects of limited complexity.

Previously Harman *et al.* compared Evolutionary Testing, Hill Climbing and Random Testing on a subset of the test objects used in this paper [27]. The present paper widens this empirical study by considering further subjects and by comparing Memetic Algorithms also. A further paper by Harman *et al.* [23] compared Evolutionary Testing, Hill Climbing and Random Testing, but here the focus was the impact of input domain reduction by removing input variables from the search which had no impact on the current branch of interest.

Search Based Testing is an example of the application of search based optimization algorithms to the identification of optimal or near optimal solutions to problems in Software Engineering. Search based optimization has proved to be a valuable tool for supporting Software Engineering activities right across the software engineering lifecycle, including requirements engineering [4, 19, 21], project planning and cost estimation [2, 3, 17] maintenance [10, 37, 46], refactoring [28, 50, 54], service oriented software engineering [14], and quality assessment [11, 32].

The present paper is the first to combine theoretical analysis grounded in theory (generalized from the literature of evolutionary computation [53]) with a large scale empirical study that both validates the predictions of the theory and provides an empirical assessment of the performance implications for choice of search based test data generation technique. This paper contains the largest empirical study conducted to date comparing the performance of Search Based Testing when applied to real-world code, comparing the performance local and global search and of the hybrid 'memetic' algorithm.

7 Conclusions and Future Work

This paper provides the large body of existing work on Search Based Testing with a theoretical underpinning, constructed as a generalization of the theories of schemata and Royal Roads from the literature of evolutionary computation. The theory is used to predict the situations in which Evolutionary Testing will perform well and to explain why. These predictions are validated by empirical observation. The empirical study then goes on to explore the impact of the choice of search technique providing some important and perhaps counter-intuitive findings. The findings of the study are surprising because they indicate that sophisticated search techniques, such as Evolutionary Testing can often be outperformed by far simpler search techniques. However, as the theory indicates, the findings also show that there do exist test data generation scenarios for which the evolutionary approach is ideally suited. Currently there is no theory for the local search approach employed, and this remains an item for future work. Where local search outperformed evolutionary search in the empirical study, it appeared to be because the fitness landscapes concerned appeared to contain only a few mimima, a type of landscape in which local search can perform very efficiently.

A further empirical study in the paper shows that in order to maximize coverage, Evolutionary Testing should be hybridized with the Hill Climbing approach. The findings of this empirical study suggest, however, that if the presence or absence of a Royal Road property can be ascertained, pure global or local search is more efficient. However, automatic identification of Royal Road properties is a hard problem, and was performed by hand in this paper. Automatic identification of Royal Road properties for structural test data generation is therefore an issue for future work. An alternative approach would be to extend the hybrid search so that it dynamically adapts to progress made by

either the global or local search component, by allocating more resources to the operators concerned and improving its efficiency and potentially its effectiveness also.

All previous work on search based branch coverage has focused on covering as many branches as possible. While this is one possible formulation of the coverage optimization problem, it fails to take account of the oracle cost. That is, the cost of determining whether a program under test produces the correct output for a test input. More work is required in order to consider multi objective formulations of the coverage optimization problem [26], that can take account of oracle cost, seeking to both maximize coverage *and* to minimize oracle cost.

An alternative approach to minimizing oracle cost is to generate pseudo-oracles using program transformations (also known in this context as testability transformations), as proposed by McMinn [41]. Search based testing is then used in conjunction with the pseudo-oracle to find faults relating to certain properties of the program under test, *e.g.* numerical precision. Here, more work is required to develop pseudo-oracle transformations for a wide range of programs and properties.

Acknowledgements

The authors would like to thank Joachim Wegener and DaimlerChrylser for providing the two industrial examples used in the empirical study, and Simon Poulding (University of York) for statistical advice.

Mark Harman is supported by EPSRC grants EP/F059442 (SLIM: SLIcing state based Models), EP/F010443 (A-CluB: Automated Cluster Breaking) and EP/D050863 (SEBASE: Software Engineering By Automated SEarch).

Phil McMinn is supported in part by EPSRC grants EP/G009600/1 (Automated Discovery of Emergent Misbehaviour) and EP/F065825/1 (REGI: Re-

verse Engineering State Machine Hierarchies by Grammar Inference).

References

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

[2] J. Aguilar-Ruiz, I. Ramos, J. C. Riquelme, and M. Toro. An evolutionary approach to estimating software development projects. *Information and Software Technology*, 43(14):875–882, 2001.

[3] G. Antoniol, M. D. Penta, and M. Harman. Search-based techniques applied to optimization of project planning for a massive maintenance project. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2005)*, pages 240–249. IEEE Computer Sociey, 2005.

[4] A. J. Bagnall, V. J. Rayward-Smith, and I. M. Whittley. The next release problem. *Information and Software Technology*, 43(14):883–890, 2001.

[5] 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.

[6] A. Baresel and H. Sthamer. Evolutionary testing of flag conditions. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2003), Lecture Notes in Computer Science vol. 2724*, pages 2442 – 2454, Chicago, USA, 2003. Springer-Verlag.

[7] A. Baresel, H. Sthamer, and M. Schmidt. Fitness function design to improve evolutionary structural testing. In *Proceedings of the Genetic and*

Evolutionary Computation Conference (GECCO 2002), pages 1329–1336, New York, USA, 2002. Morgan Kaufmann.

[8] A. Bertolino. Software testing research: Achievements, challenges, dreams. In *Future of Software Engineering 2007 (FOSE 2007)*, pages 85–103. IEEE Computer Society, 2007.

[9] L. Bottaci. Instrumenting programs with flag variables for test data search by genetic algorithm. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002)*, pages 1337 – 1342, New York, USA, 2002. Morgan Kaufmann.

[10] S. Bouktif, G. Antoniol, E. Merlo, and M. Neteler. A novel approach to optimize clone refactoring activity. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2006)*, pages 1885–1892. ACM Press, 2006.

[11] S. Bouktif, H. Sahraoui, and G. Antoniol. Simulated annealing for improving software quality prediction. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2006)*, pages 1893–1900. ACM Press, 2006.

[12] L. C. Briand, Y. Labiche, and M. Shousha. Stress testing real-time systems with genetic algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2005)*, pages 1021–1028, Washington DC, USA, 2005. ACM Press.

[13] British Standards Institute. BS 7925-1 vocabulary of terms in software testing, 1998.

[14] G. Canfora, M. Di Penta, R. Esposito, and M. L. Villani. An approach for QoS–aware service composition based on genetic algorithms. In *Proceedings*

of the Genetic and Evolutionary Computation Conference (GECCO 2005), pages 1069–1075. ACM Press, 2005.

[15] K. Derderian, R. Hierons, M. Harman, and Q. Guo. Automated unique input output sequence generation for conformance testing of FSMs. *The Computer Journal*, 39:331–344, 2006.

[16] 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.

[17] J. J. Dolado. A validation of the component-based method for software size estimation. *IEEE Transactions on Software Engineering*, 26(10):1006–1021, 2000.

[18] R. Ferguson and B. Korel. The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology*, 5(1):63–86, 1996.

[19] A. Finkelstein, M. Harman, A. Mansouri, J. Ren, and Y. Zhang. Fairness analysis in requirements assignments. In *Proceedings of the IEEE International Requirements Engineering Conference*. IEEE Computer Society, 2008. To Appear.

[20] R. L. Glass. *Facts and Fallacies of Software Engineering*. Addison Wesley, 2002.

[21] D. Greer and R. G. Software release planning: an evolutionary and iterative approach. *Information and Software Technology*, 46(4):243–253, 2004.

[22] M. Harman. The current state and future of search based software engineering. In *Future of Software Engineering 2007 (FOSE 2007)*, pages 342–357. IEEE Computer Society, 2007.

[23] 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.

[24] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper. Testability transformation. *IEEE Transactions on Software Engineering*, 30(1):3–16, 2004.

[25] M. Harman and B. Jones. Search-based software engineering. *Information and Software Technology*, 43(14):833–839, 2001.

[26] M. Harman, K. Lakhotia, and P. McMinn. A multi-objective approach to search-based test data generation. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2007)*, pages 1098–1105, London, UK, 2007. ACM Press.

[27] M. Harman and P. McMinn. A theoretical & empirical analysis of evolutionary testing and hill climbing for structural test data generation. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2007)*, pages 73–83, London, UK, 2007. ACM Press.

[28] M. Harman and L. Tratt. Pareto optimal search-based refactoring at the design level. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2007)*, pages 1106 – 1113. ACM Press, 2007.

[29] J. H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, 1975.

[30] Y. Jia and M. Harman. Constructing subtle faults using higher order mutation testing. In *8th International Working Conference on Source Code*

Analysis and Manipulation (SCAM 2008, Beijing, China, 2008. IEEE Computer Society. To appear.

[31] T. Jones. *Evolutionary Algorithms, Fitness Landscapes and Search*. PhD thesis, University of New Mexico, Albuquerque, 1995.

[32] T. M. Khoshgoftaar, L. Yi, and N. Seliya. A multi-objective module-order model for software quality enhancement. *IEEE Transactions on Evolutionary Computation*, 8(6):593–608, December 2004.

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

[34] B. Korel. Dynamic method for software test data generation. *Software Testing, Verification and Reliability*, 2(4):203–213, 1992.

[35] K. Lakhotia, M. Harman, and P. McMinn. Handling dynamic data structures in search-based testing. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2008)*, pages 1759–1766, Atlanta, Georgia, USA, 2008. ACM Press.

[36] D. Leffingwell and D. Widrig. *Managing Software Requirements: A Use Case Approach*. Addison Wesley, 2003.

[37] K. Mahdavi, M. Harman, and R. Hierons. A multiple hill climbing approach to software module clustering. In *IEEE International Conference on Software Maintenance (ICSM 2003)*, pages 315–324. IEEE Computer Sociey, 2003.

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

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

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

[41] P. McMinn. Search-based failure discovery using testability transformations to generate pseudo-oracles. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2009)*, Montreal, Canada, to appear, 2009. ACM Press.

[42] P. McMinn, D. Binkley, and M. Harman. Empirical evaluation of a nesting testability transformation for evolutionary testing. *ACM Transactions on Software Engineering Methodology*, 18(3), 2009.

[43] 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.

[44] P. McMinn and M. Holcombe. Evolutionary testing using an extended chaining approach. *Evolutionary Computation*, 14:41–64, 2006.

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

[46] B. S. Mitchell and S. Mancoridis. On the automatic modularization of software systems using the bunch tool. *IEEE Transactions on Software Engineering*, 32(3):193–208, 2006.

[47] M. Mitchell, S. Forrest, and J. H. Holland. The royal road for genetic algorithms: Fitness landscapes and GA performance. In *Proceedings of the*

First European Conference on Artificial Life, pages 245–254. MIT Press, 1992.

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

[49] National Institute of Standards and Technology. The economic impacts of inadequate infrastructure for software testing, May 2002. Planning Report 02-3.

[50] M. O'Keeffe and M. O'Cinneide. Search-based software maintenance. In *Conference on Software Maintenance and Reengineering (CSMR'06)*, pages 249–260, 2006.

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

[52] Radio Technical Commission for Aeronautics. RTCA DO178-B Software considerations in airborne systems and equipment certification, 1992.

[53] C. R. Reeves and J. E. Rowe. *Genetic Algorithms - Principles and Perspectives: A Guide to GA Theory*. Springer, 2002.

[54] O. Seng, J. Stammel, and D. Burkhart. Search-based determination of refactorings for improving the class structure of object-oriented systems. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2006)*, pages 1909–1916. ACM Press, 2006.

[55] N. Tracey, J. Clark, K. Mander, and J. McDermid. Automated test data generation for exception conditions. *Software - Practice and Experience*, 30(1):61–79, 2000.

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

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

[58] J. Wegener, H. Sthamer, B. F. Jones, and D. E. Eyres. Testing real-time systems using genetic algorithms. *Software Quality Journal*, 6(2):127–135, 1997.

[59] 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*, pages 116–121, San Mateo, California, USA, 1989. Morgan Kaufmann.

[60] 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.

[61] M. Xiao, M. El-Attar, M. Reformat, and J. Miller. Empirical evaluation of optimization algorithms when used in goal-oriented automated test data generation techniques. *Empirical Software Engineering*, 12(2):183–239, 2007.

Biographies

Mark Harman

Mark Harman is professor of Software Engineering in the Department of Computer Science at King's College London. He is widely known for work on source code analysis and testing and he was instrumental in the founding of the field of Search Based Software Engineering, a field that currently has active researchers in 24 countries and for which he has given 14 keynote invited talks. Professor Harman is the author of over 140 refereed publications, on the editorial board of 7 international journals and has served on 90 programme committees. He is director of the CREST centre at King's College London, and is the principal investigator for a current research grant portfolio of £3.3m.

Phil McMinn

Phil McMinn has been a Lecturer in Computer Science at the University of Sheffield, UK, since October 2006. He was awarded his PhD from Sheffield in January 2005, which was funded by DaimlerChrysler Research and Technology. He has published several papers in the field of search based testing. His research interests cover software testing in general, program transformation and agent-based systems and modelling. Dr McMinn is currently funded by the UK Engineering and Physical Science Research Council (EPSRC) to work on testing agent-based systems and the automatic reverse engineering of state machine descriptions from software.