A Theoretical & Empirical Analysis of Evolutionary Testing and Hill Climbing for Structural Test Data Generation

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

Phil McMinn
University of Sheffield, Regent Court,
211 Portobello St, Sheffield, S1 4DP, UK
p.mcminn@dcs.shef.ac.uk

ABSTRACT

Evolutionary testing has been widely studied as a technique for automating the process of test case generation. However, to date, there has been no theoretical examination of when and why it works. Furthermore, the empirical evidence for the effectiveness of evolutionary testing consists largely of small scale laboratory studies. This paper presents a first theoretical analysis of the scenarios in which evolutionary algorithms are suitable for structural test case generation. The theory is backed up by an empirical study that considers real world programs, the search spaces of which are several orders of magnitude larger than those previously considered.

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

Keywords. Automated test data generation, evolutionary testing, genetic algorithms, hill climbing, schema theory, royal road

1. INTRODUCTION

Evolutionary Testing (ET) is the process of automatically generating test data according to a test adequacy criterion (encoded as a fitness function) using evolutionary search algorithms, whose search is guided by a fitness function. ET has been widely studied in the literature, where it has been applied to many test data generation scenarios including temporal testing [27], stress testing [7], finite state machine testing [8] and exception testing [24]. By far the most common form of evolutionary testing considered in the literature is structural test data generation [5, 6, 12, 13, 14, 17, 22, 25, 26, 29, 30] and it is this topic that forms the subject of this paper.

Despite the considerable level of interest in ET, to date there has been no theoretical analysis that characterizes the types of test data generation scenario for which ET is 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 evolutionary testing tend to consider small, artificial 'laboratory programs' rather than real world programs with large and complex search spaces. This leaves the literature with some important open questions, such as

> "When and why does ET work?"

and

> "How does ET performance compare to other search techniques?"

This paper addresses these questions. It presents a theoretical development of Holland's well known schema theory [23]. The schema theory was later developed by Mitchell et al. [20] in a study of the so-called 'Royal Road' functions, which account for the effect of the all-important crossover operator of genetic algorithms. Both the schema theory and the Royal Road theory were developed purely for binary genetic algorithms and have not been previously adapted for the more complex chromosomes required by ET.

The paper introduces a generalization of both theories that does cater for ET, showing how the generalized theory predicts the kinds of 'Royal Road' search problem for which ET will be well suited. The paper also 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 ET against local search, using Hill Climbing (HC), and Random Testing (RT). The primary contributions of the paper are:

1. The introduction of a schema theory and Royal Road theory for ET that predict the structural test data generation problems to which ET will be well suited.
2. An empirical validation of the predictions of the theory that provides evidence to support the claim that ET 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 ET compared to HC and RT. This empirical study has several findings, some of which are surprising:
   (a) The results support the view that RT can find test data for many cases, but leaves some hard–to–cover branches for which more intelligent search is required.

ISSTA'07, July 9–12, 2007, London, England, United Kingdom.

```
        void all_zeros(int list[], int size) {
            int total = 0, i;
(1)         for (i=0; i < size; i++) {
(2)             if (list[i] == 0) {
(3)                 total ++;
                }
            }
(4)         if (total == size) {
(5)             printf("All zeros \n");
            }
        }
```

Figure 1: The illustrative `all_zeros` function for demonstrating how ET works

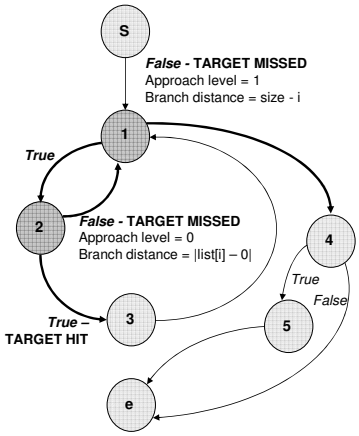

Figure 2: CFG for the `all_zeros` function of Figure 1, showing how the fitness is calculated for coverage of the true branch from node 2

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

(c) Though HC outperforms ET in non Royal Road scenarios, there do exist Royal Road scenarios in which ET is successful and HC and random search fail. This indicates that for best overall performance, a hybrid approach will be required.

The rest of the paper is organised as follows. Section 2 provides a detailed description of the ET and HC algorithms used in the paper to facilitate replication. Section 3 introduces the schema and Royal Road theory for ET, while Section 4 presents the results of the empirical study that both validates the theory and addresses performance questions. Section 5 presents related work, while Section 6 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 test data generation.

For branch coverage, a separate search is conducted to find test data to cover each uncovered branch. The fitness function, to be minimized, combines a measure known as the 'approach level' with the 'branch distance'. The approach level is a measure of how close a candidate test input was to executing the desired branch. It is a count of how many of the branch's control dependent nodes were *not* encountered in the path executed by the input. Suppose the target is the true branch from node 2 in Figure 1. The control flow graph is presented with fitness computation information in Figure 2. If `size <= 0`, node 2 is not encountered and the approach level is 1. However, if the loop body is executed, node 2 will be encountered and the approach level is 0.

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 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 2 of the program in Figure 1, the branch distance is computed using the formula `|list[i] - 0|`. The closer the values of `list[i]` and `0`, the 'nearer' 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. The complete fitness value is traditionally computed by normalizing the branch distance and adding it to the approach level [26].

The search approaches described here work to optimize a fixed-length vector of real-valued variables (referred to as the 'search vector') according to the fitness function. The tester provides a mapping from a program's inputs to the search vector by fixing array lengths and fixing data structures involving pointers. For example, if the array length of `list` in Figure 1 is fixed at 5, the search vector would be `<list[0] ,..., list[4], size>`. That is, the first five values of the vector correspond to the five values of the `list` array, followed by the value of the `size` variable. The tester must also set the range and accuracy (in decimal places) of each variable. For example, if the array length of `list` is fixed at 5, the `size` variable would have the range 0-5. If the `size` variable takes on values of less than 5 in the course of the search, latter elements of the `list` array will simply be ignored in the course of program execution.

The next two subsections describe the implementation of ET and HC in detail to facilitate replication.

2.1 Genetic Algorithms (GA) and Evolutionary Testing (ET)

Genetic algorithms (GAs) belong to the family of evolutionary algorithms, which work to evolve superior candidate solutions (known as 'individuals', denoted by chromosomes) using mechanisms inspired by the processes of natural Darwinian evolution. The search simultaneously evolves several individuals in a population, creating a global search. It is this application of evolutionary algorithms to search-based testing that has become known as Evolutionary Testing (ET). The 'chromosome' making up each individual is the search vector (as defined in the last section). The 'genes' of the chromosome are the chromosome input variable values.

The term 'genetic algorithm' is often reserved for the evolutionary computation in which the chromosome representation is necessarily a bit string, while test data generation requires chromosomes that must respect typing information embodied in any valid input type [26]. Therefore, in this

paper the term ET is used in preference to the term GA. As will be seen in the next section, the richer chromosome types required by ET entail a generalization of Holland's GA schema theory in order to render it applicable to ET.

A distinguishing feature of both GAs and ET is the importance of the crossover (or recombination) operation, which loosely models the exchange of genetic information that takes place during reproduction in the natural world. A series of 'crossover points' are used to decide where two parent chromosomes are to be spliced in order to form the composite chromosomes of two children. As with all evolutionary computation, the hope is that the children will combine the best features of both parents to create super-fit children from fit parents. Where this fails to take place, selection pressure ensures that less fit children tend to die out. The example below shows two (parent) input vectors to the `all_zeros` program being subjected to the crossover operation at position 3 to produce two children (offspring).

Parents

list[0]	list[1]	list[2]	list[3]	list[4]	size
0	*0*	*0*	*20*	*20*	*4*
20	20	20	0	0	5

→

Offspring

list[0]	list[1]	list[2]	list[3]	list[4]	size
0	*0*	*0*	0	0	5
20	20	20	*20*	*20*	*4*

The rest of this section presents the details of the algorithm implemented for ET used in this paper. The approach is based on a careful replication of the DaimlerChrysler system for ET, which has been widely studied in the literature [4, 5, 26]. 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 ET. The aim of using this ET approach is to ensure that the results for ET do, indeed, represent the state of the art. This lends additional weight to any findings that reveal superior performance by the comparatively straightforward HC approach, to which ET is compared. As will be seen (in Section 4), the empirical study does indeed yield such results.

The population is divided into six subpopulations, maximizing the effect of the mutation operator, the range of which depends on the subpopulation. Subpopulation 1 employs a relatively large range of possible mutations, whilst subpopulation 6 chooses values from a small range. Subpopulations compete for a share of the number of individuals that they can evolve, and so the search concentrates resources where the most progress is being made.

A GA is a loop in which each iteration evolves subsequent 'generations' of the population with the aim of generating fitter input vectors. The stages of each generation are described below, comprising *selection* of individuals for *crossover* and *mutation*, and *reinsertion* of individuals into the population for the next generation. At selected generations, sub-populations exchange individuals (*migration*), and compete for resources (*competition*).

The search continues until a test case has been found, or resources have been exhausted.

Selection. Selection is the process of choosing parents for crossover. A selection strategy is generally biased towards the best individuals. However, weaker individuals retain a small selection probability, thereby maintaining diversity and avoiding premature sub-optimal convergence. Stochastic universal sampling [2] is used; the probability of an individual being selected for reproduction is proportionate to its fitness. Ranked fitness values are used to promote diversity. Ranked values depend on the individual's position, sorted by fitness. Using *linear ranking* [28], fitness values are assigned in such a way 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.

Crossover. Parents are taken two at a time for crossover. Discrete recombination [21] is used to generate offspring. 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.

Mutation. The breeder genetic algorithm [21] mutation operation is used. An input variable x_i is mutated at a probability of $p_m = 1/len$, where len is the length of the input vector. 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 to be $domain_i \cdot step_p$. The new value z_i is computed using: $z_i = x_i \pm range_i \cdot \delta$. 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.

Reinsertion, Migration and Competition. The next generation is 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. Every 20 generations, 10% of the individuals of each subpopulation are randomly selected to migrate to another randomly selected subpopulation, such that no subpopulation receives individuals from more than one other subpopulation.

The competition algorithm ensures 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 $0.9 \cdot progress_{g-1} + 0.1 \cdot rank$. 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 they can never completely die out.

2.2 Hill Climbing (HC)

Hill climbing (HC) 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.

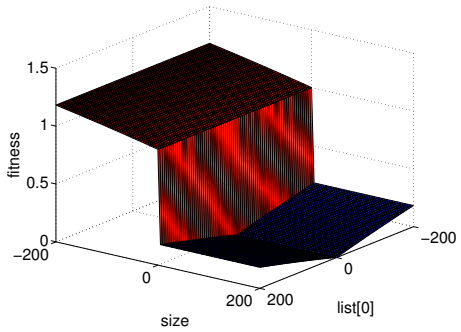

Figure 3: Fitness landscape (for two inputs) for the coverage of the true branch at node 2 in Figure 1

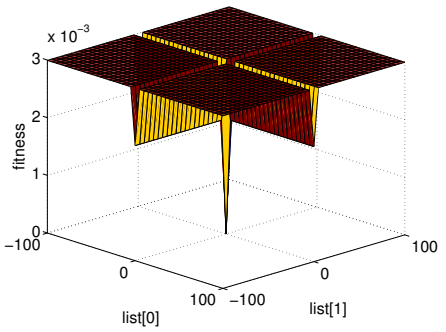

Figure 4: Fitness landscape (for two inputs) for the coverage of the true branch from node 4 of Figure 1

Since the fitness is to be minimized in this case, the equivalent term 'gradient descent' is potentially less confusing. For example, the coverage of the true branch from node 2, of Figure 1, is represented by the valley touching zero on the z axis of the fitness function surface (see Figure 3 for a visualization).

As with ET, there are many choices in the formulation of an HC 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 [12], hereinafter referred to simply as 'HC'. HC 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 targeted, and so on, until no modification of input values results in an improved fitness.

Initially, an input vector is generated at random. The first input value is selected, and its neighbourhood probed. Suppose the input vector $<< 50, 100, 20, 50, 30 >, 5 >$ was generated whilst attempting coverage of the true branch from node 2 in the `all_zeros` program from Figure 1. The first value of the `list` array is probed through 'exploratory' moves. A small decrease of the variable is tested, followed by a small increase. As `list` is an array of integers with the first value stored being 50, the values 'probed' in this way, are 49 and 51.

Recall how the fitness calculation works; the branch distance is the smallest distance encountered at node 2. The third element of `list` is 20, and the closest to 0. Therefore modifying `list[0]` has no effect on fitness. Similarly, adjusting the next element, `list[1]`, results in no change. The search then selects `list[2]`, for which a decreased value - 19 - does have an effect, reducing the branch distance. 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 $m_i = s^i \cdot 10^{-acc_v} \cdot dir$, where acc_v is the accuracy of the v^{th} variable in decimal places, and s is the *repeat base* ($s = 2$ for experiments in this paper). Successive values of `list[2]` therefore are 19 (the initial move), 17, 13, 5, and -11. At -11, the fitness becomes worse, and so this move is ignored. The search continues to re-establish a new direction through exploratory moves in order to make further pattern moves. Eventually the value of zero is found for `list[2]`.

A well-known problem with local search methods like HC is their tendency to become trapped in a local minima. For example, recall the `all_zeros` function from Figure 1 and assume `size` is fixed; its values may not be probed using the alternating variable method. The relationship between inputs and values for node 4's predicate is not as simple and direct as that for node 2. For the most part, exploratory moves for values of `list` have no effect on `total`, resulting in areas of undistinguished fitness (See Figure 4 for a visualization). On encountering one of these 'fitness invariant plateaux', HC terminates without finding the required test data. In order to address this problem, HC is restarted at a new randomly chosen start point many times, until a budget of fitness evaluations has been exhausted.

3. THEORETICAL FOUNDATIONS

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

3.1 The Schema Theory of Genetic Algorithms

In a binary GA, 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.

In a particular generation of a GA, g, the population will be a set of chromosomes, denoted $P(g)$. The GA will not be able to determine the true fitness of a schema $\mathbf{h}$, because it will not necessarily contain all possible instantiations. How-

ever, 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. The defining length of a schema $\mathbf{h}$, will be denoted $\delta(\mathbf{h})$. As the GA executes, it evaluates the fitness of the chromosomes in each generation. Every time an individual is evaluated, 2^λ schemata are evaluated and so the GA processes a large number of schemata, far larger than the number of individual chromosomes that it evaluates. This observation can be made more formal, by considering the number of instances of a schema that pertain at each generation of the GA. 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:

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

Both mutation and crossover disrupt schemata in a population. 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 [11]. 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 GA. It also indicates that crossover can disrupt the schemata. This may seem counter-intuitive, because it is from the crossover operator that GAs are intended to derive much of their capability [20]. Fortunately, this issue is addressed by the Royal Road theory of GAs, which clarifies the important role of crossover.

3.2 Schema Theory for Test Data Generation by Genetic Algorithms

ET typically does not use binary GAs, so the schema theory is not directly applicable. A new form of schema theory for ET therefore has to be constructed. Fortunately, the input vectors used in ET can be captured by a generalization of Holland's schema theory. These ET 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 ET, 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 ET 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.

ET 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 ET, 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 ET 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 [20] introduced the theoretical study of Royal Road landscapes in order to capture formally the intuition underlying the 'folk theorem' that GA will outperform a local search such as HC, because of the way in which GA 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 artificial 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 5. 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 GA to follow a tree of schemata that lead directly to the global optimum. This tree of ever fitter schemata form the 'Royal Road'. Thus royal road fitness functions can be formally defined as follows:

Definition - Royal Road Fitness Function. A royal road fitness function is one for which fitness is monotonic in the sense that the fitness of a schema that combines a set of above average schemata, $s_1, \ldots, s_n$, is guaranteed to be greater than the fitness of any of the individual schema, s_i, $1 \leq i \leq n$.

s_1 : 11111111**********...**********

s_2 : ********11111111**********...**********

s_3 : ****************11111111**********...**********

s_4 : ************************11111111**********...**********

s_5 : ********************************11111111**********...**********

s_6 : **11111111**********

s_7 : **...**********11111111********

s_8 : **...**********11111111

s_9 : 1111111111111111**********...**********

s_{10} : ****************1111111111111111**********...**********

s_{11} : ********************************1111111111111111**********

s_{12} : **...**********1111111111111111

s_{13} : 11111111111111111111111111111111**********...**********

s_{14} : ********************************1111111111111111111111111111111111

s_{15} : 1111111111111111111111111111111111111111111111111111111111111111

Figure 5: Royal Road Function of 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 GA 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 ET are input variables and the schemata denote constraints on these variables, there is an additional property that can be seen to hold for ET 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 ET. 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, $\forall i.1 \leq i \leq 16.s_1(i) = 1$.

The implication is that for an ET approach to exhibit a Royal Road property, the more fit schemata must be expressed as conjunctions of lower order schemata involving disjoint sets of input variables. Where this property holds, the ET Royal Road theory predicts that ET 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 trails than less fit schema.

The archetype of this Royal Road property 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 pres-

Table 1: Test object details

Test Object / Function	Lines of Code	Number of Branches	Approx. Domain Size (10^x)
bibclean-2.08			
check_ISSN		42	120
check_ISBN		42	120
Total	178		
eurocheck-0.1.0			
main		22	31
Total	70		
gimp-2.2.4			
gimp_rgb_to_hsl_int		14	7
gimp_rgb_to_hsv		10	40
gimp_hsv_to_rgb		16	40
gimp_hsv_to_rgb_int		16	7
gimp_rgb_to_hsv_int		14	7
gimp_rgb_to_hsl		14	40
gimp_rgb_to_hsv4		18	7
gimp_hwb_to_rgb		18	30
gimp_hsv_to_rgb4		16	30
gradient_calc_radial_factor		6	21
gradient_calc_square_factor		6	21
gradient_calc_conical_sym_factor		8	31
gradient_calc_conical_asym_factor		6	31
gradient_calc_bilinear_factor		6	34
gradient_calc_spiral_factor		8	37
gradient_calc_linear_factor		8	31
Total	867		
space			
addscan		32	539
fixgramp		8	23
fixport		6	132
fixselem		8	132
fixsgrel		68	544
fixsgrid		22	104
gnodfind		4	70
seqrotrg		32	207
sgrpha2n		16	457
Total	2210		
spice			
cliparc		64	44
clip_to_circle		42	30
Total	269		
tiff-3.8.2			
TIFF_SetSample		14	10
TIFF_GetSourceSamples		18	15
PlaceImage		16	38
Total	182		
Grand Total	**3,776**	**640**	

ence 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.

4. EMPIRICAL STUDY

An empirical study was conducted on almost 4,000 lines of production C code, comprising 33 individual functions and a total of 640 branches, with search space sizes ranging from 10^7 to 10^{544}. It represents one of the largest ET studies to date. 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. Further details can be found in Table 1. The IGUANA tool [16] was used to perform the searches.

4.1 Research Questions

The empirical study addresses three research questions, described below:

Research Question 1 - Validation of ET 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 HC algorithm?

Research Question 2 - Validation of crossover hypothesis. According to the theory, the reason for ET'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 ET perform on Royal Road functions when the effects of the crossover operator are removed?

Research Question 3 - Performance for non Royal Road functions. For predicates that do not have a Royal Road fitness function, the genetic algorithm may perform no better, and possibly worse than HC. The theory is concerned with effectiveness not efficiency and so it cannot make predictions about how ET will perform relative to HC and RT, 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 Test objects

The subjects used in the empirical study were six real world programs for which summary data are presented in Table 1. The rest of this section provides a brief overview of these subject programs.

`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 and ISSN codes used to identify publications. `eurocheck-0.1.0` is also 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, 9]. Nine functions were tested. `spice` is an open source general purpose analogue circuit simulator. 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.

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. 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 after-

wards. 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 minimum and maximum values of the input variables (along with their precision in the case of floating point numbers), were specified for the search process. From this information, the input domain size - i.e. the search space size - can be computed. This is recorded in Table 1. As can be seen, the search space sizes can be extremely large, ranging up to 10^{544}; a number considerably larger than the number of atoms in the observable universe (typically estimated at 10^{80}).

4.3 Experiments

The test data generation experiments for branch coverage used ET, HC and RT. Each search was terminated after 100,000 fitness evaluations if test data had not been found. The ET and HC algorithms were described in the previous section. The RT algorithm simply constructs 100,000 valid random inputs. The test data search involving each search method and each branch was repeated 30 times using an identical list of 30 fixed seeds for random number generation. In this way the experiments provide a basis for assessment of the statistical significance of the results.

Figure 6 summarises the branches covered by the different search methods. Of the 640 branches, 532 branches (83%) were covered by all search methods. 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 is required. Of the 44 branches not covered during all 30 random search runs, 41 were covered by either HC or ET, 10 of which were covered by ET only and 5 by hill climbing only. A further 4 branches were covered by random search only, but only in 3 or fewer of the 30 runs.

The final 63 branches were infeasible or simply uncovered by the search techniques. In many instances, such as a large proportion of branches from `eurocheck-0.1.0`, 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 overall input domain, RT also fails. Such 'difficult' fitness landscapes are studied further elsewhere in the literature [3, 10, 17, 18].

Clearly there is no point in attempting to use an 'intelligent' (and therefore more expensive) metaheuristic search, when random testing will do. Therefore, in answering the research questions, only those branches for which RT fails (on all 30 attempts) are considered. 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. If there were an automated decision procedure for finding Royal Roads, then automated test data generation would not be as hard as it is.

Answers to Research Questions

Research Question 1 - Validation of ET Theory. The Royal Road property is found in branches of the `bibclean` test object. The `check_ISSN` and `check_ISBN` (Figure 7) functions both read a string of 30 characters. The function sequentially searches through the characters in order to find those valid for an ISSN or ISBN number. When such a character is found, a counter variable is incremented. When this counter is equal to 8 (`check_ISSN`) or 10 (`check_ISBN`), 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 ET will perform well for this example.

The empirical results (presented in Table 2) support this predication. ET almost always succeeds in finding test data, whilst HC always fails. By contrast, HC 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 improvement in fitness, unless c happens to be adjacent to a block of valid characters.

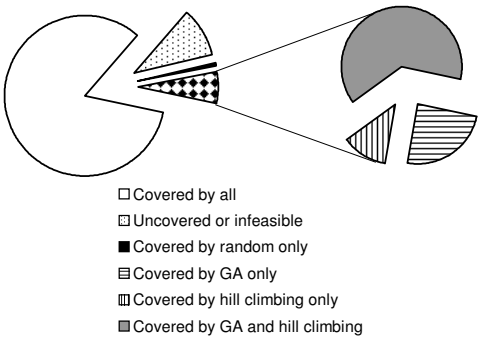

Figure 6: Pie chart showing proportions of branches covered by the different search methods

Table 2: Branches with Royal Road properties. Success rate is the number of runs in which test data was found by the search. AE is the average number of fitness evaluations, SD is the standard deviation

Test Object (Branch ID)	GA Success Rate (AE/SD)	Hill Climbing Success Rate (AE/SD)
bibclean-2.08		
check_ISBN (23F)	97% (8,091/3,042)	0% (n/a)
check_ISBN (27T)	97% (8,091/3,042)	0% (n/a)
check_ISBN (29F)	97% (9,377/3,184)	0% (n/a)
check_ISBN (29T)	97% (8,091/3,042)	0% (n/a)
check_ISSN (23F)	100% (5,154/1,606)	0% (n/a)
check_ISSN (27T)	100% (5,154/1,606)	0% (n/a)
check_ISSN (29F)	100% (6,155/1,654)	0% (n/a)
check_ISSN (29T)	100% (5,226/1,621)	0% (n/a)

```
...
for (..., checksum = 0, k = 0, n = 1; current_value[n+1]; ++n)
{
  ...
  switch (current_value[n])
  {
    ...
    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 (branches 23T and 23F) */
        {
          ISBN[k] = current_value[n];
          checksum += ISBN_DIGIT_VALUE(ISBN[k]) * k;
          break;
        }
        else if (k == 10) /* Node 27 (branches 27T and 27F) */
        {
          ISBN[k] = current_value[n];

          /* Node 29 (branches 29T and 29F) */
          if ((checksum % 11) != ISBN_DIGIT_VALUE(ISBN[k]))
              bad_ISBN(ISBN);
          ...
```

Figure 7: Code snippet from the `check_ISBN` function

Table 3: Comparing discrete recombination with no recombination for branches with Royal Road properties. AE is the average number of fitness evaluations, SD is the standard deviation

Test Object Function (Branch ID)	Discrete Success Rate (AE/SD)	None Success Rate (AE/SD)
bibclean-2.08		
check_ISBN (23F)	97% (8,091/3,042)	0% (n/a)
check_ISBN (27T)	97% (8,091/3,042)	0% (n/a)
check_ISBN (29F)	97% (9,377/3,184)	0% (n/a)
check_ISBN (29T)	97% (8,091/3,042)	0% (n/a)
check_ISSN (23F)	100% (5,154/1,606)	0% (n/a)
check_ISSN (27T)	100% (5,154/1,606)	0% (n/a)
check_ISSN (29F)	100% (6,155/1,654)	0% (n/a)
check_ISSN (29T)	100% (5,226/1,621)	0% (n/a)

Research Question 2 - Validation of crossover hypothesis. The recombination operator was disabled, and the ET experiment was re-run for `bibclean`. Test data generation failed in 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.

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. Test data was generated for the remaining branches with reasonable consistency, but with almost ten times as many fitness evaluations. Statistical significance was tested for using a $t-$test with α level 0.01. In all comparable cases the test revealed that the results were statistically significant. The p values are listed in Table 4.

Research Question 3 - Performance for non Royal Road functions. Table 5 shows branches not covered by random search and which do not exhibit Royal Road properties. The finding is that HC significantly outperforms ET in many of these cases. This is a surprising finding given the high degree of attention paid to ET, compared to the simpler HC approach.

As these results show, there are only 2 branches for which ET is successful and for which HC fails. The first branch, 20T of the function `PlaceImage` in `tiff-3.8.2` is covered inconsistently (on only 3 occasions). The second branch, 12T of the function `gimp_hwb_to_rgb` has a fitness landscape containing a series of plateaux, making it hard for HC 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 ET 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.

There are 5 branches in `spice` that HC can cover that ET cannot in the 100,000 fitness evaluations limit. Here it seems that the fitness landscape must be navigated with extreme precision, as it appears that the test data lies in narrow sub-regions of the search space with exceptionally steep inclines. These results provide some evidence that there are at least some examples where the simple HC approach is more successful that the more sophisticated (and expensive) ET approach.

The greater expense of the ET approach is also clearly revealed by the remaining cases (for which both ET and HC achieve coverage of the target branch). For these 19 branches, HC is usually more efficient by an order of a magnitude. The results of $t-$tests show that HC is significantly better than ET for all comparable results sets at the α level 0.01.

For a further 10 branches, HC has a higher success rate than ET. For all of these branches, the fitness landscapes involved appear to be smooth, and largely free from plateaux and 'ruggedness' which hinder the HC algorithm.

5. RELATED WORK

Miller and Spooner [19] were the first authors to dynamically generate test data, defining an objective function to be optimized using numerical maximization techniques. Korel [12] was the first to use the strategy described as hill climbing in this paper. Xanthakis et al. [29] were the first to apply evolutionary computation to test data generation for the execution of paths. This work has been extended by various authors [14, 22, 24, 26] for branch coverage. In 2004 there was a sufficiently large body of work in search-based testing to warrant a detailed survey of the field [15]. However, despite this large volume of work, this is the first paper to provide a theoretical explanation of why and where evolutionary approaches work. This paper also contains the largest empirical study conducted to date comparing the performance of ET, HC and RT, when applied to real world code.

Many empirical studies in the literature compare ET with RT alone [14, 22, 26], finding that ET achieves the highest levels of coverage, and more efficiently. Though 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.

Table 4: Comparing discrete recombination using normal parents with discrete recombination using a randomly generated second parent, which is not a member of the current population. AE is the average number of fitness evaluations, SD is the standard deviation

Test Object / Function (Branch ID)	Normal Parents Success Rate (AE/SD)	Random 2nd Parent Success Rate (AE/SD)	Observerd Significance Level ($p-$value)
bibclean-2.08			
check_ISBN (23F)	97% (8,091/3,042)	0% (n/a)	n/a
check_ISBN (27T)	97% (8,091/3,042)	0% (n/a)	n/a
check_ISBN (29F)	97% (9,377/3,184)	0% (n/a)	n/a
check_ISBN (29T)	97% (8,091/3,042)	0% (n/a)	n/a
check_ISSN (23F)	100% (5,154/1,606)	77% (73,766/19,489)	10^{-15}
check_ISSN (27T)	100% (5,154/1,606)	77% (73,766/19,489)	10^{-15}
check_ISSN (29F)	100% (6,155/1,654)	7% (67,941/22,383)	n/a
check_ISSN (29T)	100% (5,226/1,621)	77% (73,766/19,489)	10^{-15}

Table 5: Branches not exhibiting Royal Road properties. AE is the average number of fitness evaluations, SD is the standard deviation

Test Object / Function (Branch ID)	GA Success Rate (AE/SD)	Hill Climbing Success Rate (AE/SD)	Observed Significance Level ($p-$value)
gimp-2.2.4			
gimp_hwb_to_rgb (12T)	100% (1,299/658)	0% (n/a)	n/a
gimp_hwb_to_rgb (3T)	100% (10,798/2,428)	100% (126/28)	0.0
gimp_rgb_to_hsl (4T)	100% (11,336/1,343)	100% (153/31)	0.0
gimp_rgb_to_hsv (5F)	100% (7,696/1,300)	100% (142/33)	0.0
gimp_rgb_to_hsv4 (11F)	100% (4,767/1,269)	100% (963/1,126)	10^{-12}
gimp_rgb_to_hsv_int (10T)	100% (4,767/1,269)	100% (963/1,126)	10^{-12}
gradient_calc_bilinear_factor (8T)	100% (13,509/3,086)	100% (195/35)	0.0
gradient_calc_conical_asym_factor (3F)	100% (21,189/4,403)	100% (246/32)	0.0
gradient_calc_conical_sym_factor (3F)	100% (21,189/4,403)	100% (246/32)	0.0
gradient_calc_spiral_factor (3F)	100% (21,387/4,292)	100% (246/32)	0.0
space			
seqrotrg (17T)	37% (73,127/16,953)	100% (24,769/21,260)	n/a
seqrotrg (22T)	37% (73,127/16,953)	100% (24,769/21,260)	n/a
seqrotrg (27F)	83% (38,602/25,633)	100% (2,543/3,072)	10^{-7}
spice			
clip_to_circle (1F)	100% (11,615/3,188)	100% (105/16)	0.0
clip_to_circle (36T)	7% (26,390/23,139)	57% (36,456/26,978)	n/a
clip_to_circle (49T)	0% (n/a)	37% (52,003/33,643)	n/a
clip_to_circle (4F)	100% (11,687/3,079)	100% (135/23)	0.0
clip_to_circle (62T)	0% (n/a)	43% (37,992/29,778)	n/a
clip_to_circle (68F)	0% (n/a)	77% (43,501/29,176)	n/a
cliparc (13F)	100% (10,356/2,822)	100% (569/475)	0.0
cliparc (15F)	100% (11,050/2,667)	100% (767/754)	0.0
cliparc (15T)	100% (11,454/4,126)	100% (1,108/856)	10^{-14}
cliparc (22T)	0% (n/a)	10% (53,281/38,259)	n/a
cliparc (24T)	3% (35,448/0)	97% (5,502/9,508)	n/a
cliparc (63F)	0% (n/a)	100% (21,168/19,916)	n/a
tiff-3.8.2			
PlaceImage (16T)	33% (8,816/1,813)	13% (71,660/20,390)	n/a
PlaceImage (20T)	10% (28,249/34,400)	0% (n/a)	n/a
TIFF_GetSourceSamples (7T)	100% (11,669/3,097)	100% (85/39)	0.0
TIFF_GetSourceSamples (9T)	100% (11,083/3,460)	100% (76/29)	0.0
TIFF_SetSample (11T)	100% (8,398/2,439)	100% (60/12)	0.0
TIFF_SetSample (5T)	100% (8,740/2,055)	100% (80/40)	0.0
TIFF_SetSample (7T)	100% (8,463/2,074)	100% (62/13)	0.0
TIFF_SetSample (9T)	100% (8,030/2,343)	100% (63/16)	0.0

There have also been several studies comparing ET, HC, RT and simulated annealing for a variety of criteria. However, these studies tend to report results on small numbers of programs, each with limited complexity. For instance, Wang and Jeng [25] compare ET with HC and memetic algorithms for branch coverage on six examples. Memetic algorithms are found to outperform HC, which in turn outperforms ET. However, only a small number of branches are investigated, and none of these contain Royal Road properties. Mansour and Salame [13] compare ET, HC and simulated annealing for test data generation for path coverage, finding that HC discovers test data faster than ET and simulated annealing, but that ET and simulated annealing can cover more paths. They also report that simulated annealing performs better than ET. However, HC 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. [30] compare ET with simulated annealing for condition-decision coverage, finding that ET is consistently the best performer. However, once again, the study is small scale, featuring test objects of limited complexity.

The present paper is the first to combine theoretical analysis grounded in theory (generalized from the literature of evolutionary computation [23]) 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.

6. 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 ET 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 ET 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.

Acknowledgments

Mark Harman is supported by EPSRC Grants EP/D050863, GR/S93684 & GR/T22872, by EU grant IST-33472 (EvoTest) and also by the kind support of DaimlerChrysler Berlin and Vizuri Ltd., London. The authors are also grateful to Kathy Harman, Kiran Lakhotia, Afshin Mansouri, Rebecca McMinn and Xin Yao for their comments on earlier versions of this paper.

7. REFERENCES

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

[2] 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.

[3] A. Baresel, D. Binkley, M. Harman, and B. Korel. Evolutionary testing in the presence of loop-assigned flags: A testability transformation approach. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2004)*, pages 43–52, Boston, Massachusetts, USA, 2004. ACM.

[4] A. Baresel and H. Sthamer. Evolutionary testing of flag conditions. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2003)*, volume 2724 of *LNCS*, pages 2442–2454, Chicago, 12-16 July 2003. Springer-Verlag.

[5] 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, San Francisco, CA 94104, USA, 9-13 July 2002. Morgan Kaufmann Publishers.

[6] L. Bottaci. Instrumenting programs with flag variables for test data search by genetic algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002)*, pages 1337–1342, New York, 9-13 July 2002. Morgan Kaufmann Publishers.

[7] 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), Washington DC, USA, June 25-29, 2005*, pages 1021–1028. ACM, 2005.

[8] K. Derderian, R. Hierons, M. Harman, and Q. Guo. Automated Unique Input Output sequence generation for conformance testing of FSMs. *The Computer Journal*, 49(3):331–344, 2006.

[9] 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, Oct. 2005.

[10] 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.

[11] J. H. Holland. *Adaption in Natural and Artificial Systems*. MIT Press, Ann Arbor, 1975.

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

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

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

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

[16] P. McMinn. IGUANA: Input generation using automated novel algorithms. A plug and play research tool. *Technical Report, Department of Computer Science, University of Sheffield*, 2007.

[17] 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.

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

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

[20] M. Mitchell, S. Forrest, and J. H. Holland. The royal road for genetic algorithms: Fitness landscapes and GA performance. In F. J. Varela and P. Bourgine, editors, *Proceedings of the First European Conference on Artificial Life*, pages 245–254, Cambridge, MA, 1992. MIT Press.

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

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

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

[24] N. Tracey, J. Clark, and K. Mander. Automated program flaw finding using simulated annealing. In *International Symposium on Software Testing and Analysis (ISSTA 98)*, pages 73–81, March 1998.

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

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

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

[28] 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 3rd International Conference on Genetic Algorithms*, pages 116–121, San Mateo, California, USA, 1989. Morgan Kaufmann.

[29] 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.

[30] 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.