How Does Program Structure Impact the Effectiveness of the Crossover Operator in Evolutionary Testing?

Phil McMinn
University of Sheffield, Department of Computer Science
Regent Court, 211 Portobello, Sheffield, S1 4DP, UK

***Abstract*—Recent results in Search-Based Testing show that the relatively simple Alternating Variable hill climbing method outperforms Evolutionary Testing (ET) for many programs. For ET to perform well in covering an individual branch, a program must have a certain structure that gives rise to a fitness landscape that the crossover operator can exploit. This paper presents theoretical and empirical investigations into the types of program structure that result in such landscapes. The studies show that crossover lends itself to programs that process large data structures or have an internal state that is reached over a series of repeated function or method calls. The empirical study also investigates the type of crossover which works most efficiently for different program structures. It further compares the results obtained by ET with those obtained for different variants of hill climbing algorithm; which are found to be effective for many structures considered favourable to crossover, with the exception of structures with landscapes containing entrapping local optima.**

***Keywords*-Evolutionary Testing, Crossover, Search-Based Test Data Generation**

I. Introduction

Researchers in the field of Evolutionary Computation have devoted much attention to characterising the class of fitness functions for which different search operators perform well. The Royal Road fitness functions, proposed by Mitchell *et al.* [1], [2], were an early attempt to identify the types of fitness landscapes in which crossover worked well. Watson *et al.* [3] later proposed the H-IFF ('Hierarchical If-and-only-if') fitness functions, which are mutation-deceptive and result in Genetic Algorithms with crossover outperforming hill climbing. Later work by Jansen and Ingo Wegener [4] gave rise to the 'Real' Royal Road fitness functions, for which crossover was shown to be provably essential. Conversely, recent work by Richter *et al.* [5] produced the 'Ignoble Trail' fitness functions, which are characterisations of crossover-deceptive landscapes for which crossover is shown to be provably harmful.

The Search-Based Software Engineering community has only begun to investigate theoretical issues relating search-based optimisation to software engineering problems [6], [7], [8], [9], with the aim of developing a clearer understanding of why different techniques may be more effective for different types of problem, and how they might be improved.

This paper concerns the characterisation of programs for which structural test data search by Genetic Algorithms (GAs), referred to as Evolutionary Testing (ET), will perform well. The primary difference between GAs and simpler hill climbing approaches is the ability of a GA to exchange information between candidate solutions in a population, through the crossover operator. Recent studies by Harman and McMinn [8], [9] on a selection of open source and industrial programs revealed that simple hill climbing search in the form of Korel's Alternating Variable Method (AVM) was able to outperform ET in covering the vast majority of branches considered. For the small number of branches for which ET outperformed the AVM, a 'Royal Road'-type property was found to be present. The *constraint-schema* theory for ET was developed, based on the schema theory of binary GAs, and used to characterise Royal Road-type landscapes for ET. However, despite the characterisation of *search landscapes* for which ET is predicted to do well, no clear understanding presently exists regarding the types of *programs and program structures* that give rise to these landscapes.

This paper, therefore, is the first to ask the following research questions:

- What types of programs and program structure enable ET to perform well, through crossover, and how?
- Which form of crossover is best suited to such programs and structures, and why?

This paper addresses the above questions with theoretical and empirical studies. Its primary contributions include:

1. A theoretical analysis of the types of program structure that will favour crossover in ET. The analysis indicates that crossover is likely to perform well when a program has a large number of input variables which discretely impact conditions surrounding the target with a low probability. The types of program likely to benefit from the use of the crossover operator, and therefore search by GAs, are those that process large data structures or have an internal state that is reached by a series of repeated function or method calls.
2. An empirical study involving a number of artificial case studies designed to validate the theoretical study.
3. An empirical study to determine which type of crossover operator works best for different programs. The results show that the operator used by ET is often outperformed by standard uniform crossover.

4. A further empirical study comparing variants of hill climbing on the programs used to test the effectiveness of crossover; including Random Mutation Hill Climbing (RMHC). As with GA Royal Roads [2], RMHC outperforms ET on landscapes for which crossover performs well, but are free of deception in the form of entrapping, local optima. Program structures do exist with the latter property, for which ET will outperform RMHC.

The paper begins by reviewing important background.

II. Background

A. Search-Based Testing (SBT)

Search-Based Testing (SBT) generates test data through optimisation of a fitness function that underpins the test criterion of interest. This paper concentrates on branch coverage, where each uncovered branch is the focus of an individual test data search. The search space is the input domain of the program under test. The fitness function scores input vectors on the basis of how close they were to executing the branch of interest. The first obstacle for the generated input is to penetrate the layers of statements in which the target branch is nested. The *approach level* [10] measures the number of control graph nodes on which the branch is control dependent, but were not executed in the path taken by a particular input. An example approach level computation is shown in Figure 1. If the path diverges from the target at some control dependency, the *branch distance* calculation is performed using the values of variables appearing in the control dependency's condition. The calculation reflects how close the conditional was to being executed with the alternative, desired outcome. For example, if a condition '`a == b`' needs to be executed as true, the branch distance is found using the formula $|a - b|$ [11]. The overall fitness function to be minimised for a branch target t and for assessing an input vector v is $fit(t, v) = approach_level(t, v) + normalise(branch_distance(t, v))$. When the fitness value is zero, t is executed by v. Experiments in this paper use the normalisation function $1 - 1.001^{-d}$ for a distance d.

The **Alternating Variable Method (AVM)** [11] was one of the earliest algorithms used for SBT. An input vector is initially chosen at random. The algorithm then optimises the fitness function for each input variable of the vector in turn. First, an 'exploratory' move is performed by increasing and decreasing the value by a small amount k ($k = 1$ for experiments with integer types in this paper). If an exploratory move results in an improvement in fitness, further 'pattern' moves are made in the direction of improvement with increasing step sizes of 2^i for the ith successive move. Pattern moves continue until fitness fails to improve, where upon exploratory moves are recommenced. If exploratory moves fail to yield improvement, the focus moves to the next input variable in the vector. Moves are made until a target-executing input is found or until exploratory moves

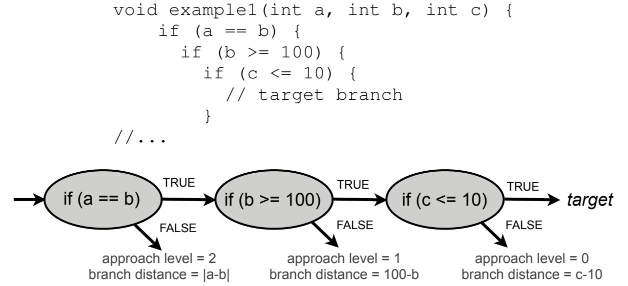


Figure 1. Approach level and branch distance calculation

have been attempted on each input variable without an improvement in fitness. At this point the search may be restarted with a new random input. The search continues until the target is covered or the pre-determined budget of fitness evaluations has been exhausted.

Evolutionary Testing (ET) is the name given to SBT techniques where Evolutionary Algorithms are used as the optimisation technique. A widely-used GA in the literature is that of Joachim Wegener *et al.* [10], which formed part of DaimlerChrysler's ET System. Instead of using a binary encoding, input vectors are optimised directly with input values forming the 'genes' of each individual chromosome. The overall population of 300 individuals is initially divided equally over 6 competing subpopulations. Individuals are linearly ranked [12] for selection with a pressure Z=1.7, with stochastic universal sampling as the selection mechanism.

Of particular interest to this paper is the use of *discrete recombination* as the crossover operator. Discrete recombination is similar to uniform crossover, but differs in one important aspect. With uniform crossover, each gene is copied into exactly one of the offspring, decided with an even probability. With discrete recombination, however, a gene may be copied into both children, one child, or neither child; with an equal probability.

The Breeder GA [13] mutation operator is used and applied at an inverse of chromosome length. Genes are mutated by the addition or subtraction of values chosen from a range decided by the subpopulation in which the individual currently resides. The range for a subpopulation p, $1 \leq p \leq 6$, is 10^{-p}. Elitist reinsertion is applied, with the top 10% of the current generation retained, while the remaining individuals are discarded and replaced with the best offspring. A progress value, $p_g = 0.9 \cdot p_{g-1} + 0.1 \cdot r$, is computed for each subpopulation at the end of the gth generation, where r is the subpopulation's average ranked fitness following linear ranking of its individuals using $Z = 1.7$. After every 4th generation, subpopulations are ranked according to their progress value and a new share of the overall population computed for each, with weaker subpopulations transferring individuals to stronger ones. A

subpopulation cannot lose its last 5 individuals, preventing its extinction. Finally, individuals migrate every 20th generation, with subpopulations exchanging 10% of their individuals at random.

B. Evolutionary Testing Constraint-Schemata

Harman and McMinn [8], [9] introduced the concept of *constraint-schemata* in order to develop a formal understanding of the crossover operator for ET. The notion of a constraint-schema is a generalisation of Holland's binary GA schema [14]. Whereas binary GA schemata are 'templates' denoting the possible chromosomes that the schema may instantiate, constraint-schemata represent explicit sets of chromosomes, defined in terms of constraints over the input variables of a program. Example constraint-schemata for the program of Figure 1, for example, include $P_1 = \{(a, b, c) \mid a = b\}$ and $P_2 = \{(a, b, c) \mid a = 0\}$. Constraint-schemata, as with their binary predecessors, allow for reasoning about the fitness of chromosomes that may be prevalent in the current population of the GA. For example, chromosomes (input vectors) belonging to P_1 will have a higher average fitness than those belonging to P_2, since P_1 defines the set of chromosomes traversing the initial approach level en route to the target. The schema theory [14] (and its ET generalisation [9]) predicts schemata of above average fitness will proliferate in successive generations of the search.

Harman and McMinn show that the crossover operator will work well for ET when the test data generation problem has a structure such that chromosomes of simpler constraint-schemata can be recombined to produce chromosomes of more specific schemata with higher average fitness. This is effectively a restating of the building block hypothesis for ET. Higher fitness, specific schemata are modelled through the conjunction of the constraints of more general schemata. For example, with the program of Figure 2, chromosomes belonging to $Q_1 = \{(a, b, c) \mid a = b\}$ may be recombined with those of $Q_2 = \{(a, b, c) \mid c \leq 10\}$ to produce chromosomes belonging to $Q = \{(a, b, c) \mid a = b \wedge c \leq 10\}$. Q has a higher average fitness than either Q_1 or Q_2, as the value of `count` will be at least 2, as opposed to 1; resulting in a lower and more favourable branch distance at the condition guarding the target. In general, if some schemata $S_1 = \{I \mid c_1\}$ and $S_2 = \{I \mid c_2\}$ are combined to produce a fitter schema $S = \{I \mid c\}$, $c = c_1 \wedge c_2$, S must respect the constraints of the more general schemata from which it was created. That is, $c \Rightarrow c_1$ and $c \Rightarrow c_2$. This is equivalent to stipulating that S is a subset (*subschema*) of S_1 and S_2; *i.e.* $S \subset S_1$ and $S \subset S_2$. Correspondingly, S_1 and S_2 are referred to as *superschemata* of S.

Surprisingly, in a study of programs comprising of 760 different branches in open source and production code, Harman and McMinn [9] found only 8 branches that allowed the crossover operator to work effectively. As such, although constraint-schemata give a clearer picture regarding the type of fitness landscape that will allow crossover to work effectively for ET, the issue of what types of programs result in such landscapes is less well understood. This is the subject under investigation for the remainder of this paper.

```
void example2(int a, int b, int c) {
  int count = 0;
  if (a == b) count ++;
  if (b >= 100) count ++;
  if (c <= 10) count ++;

  if (count == 3) {
    // target branch
    // ...
```

Figure 2. Target branch not nested but reached under the same conditions as the program of Figure 1

III. A Theoretical Analysis of Program Structures Favouring the Crossover Operator

This section investigates the factors underlying programs and target structures which will allow the crossover operator to work effectively in test data searches. The foundation for this is the concept of constraint-schemata introduced in the last section.

A. Number of conjuncts in the input condition

The *input condition* for covering a structural target in a program, such as a branch, is a constraint over a program's input variables that describes when the target will be executed. The input condition for the target of the program of Figure 2, for example, is $a = b \wedge b \geq 100 \wedge c \leq 10$. The input condition is equivalent to the constraint of the schema that describes the chromosomes (input vectors) that will cover a target structure:

Definition 1 (Covering Constraint-Schema)**.** *A constraint-schema S is said to be the* covering constraint-schema *for a target t if all chromosomes of S execute t (and there do not exist superschemata of S for which this is also true).*

Depending on the structure of the program concerned, the covering constraint-schema may be generalisable into a number of distinct superschemata; the constraints of which denote simpler sub-test data generation problems that must be solved individually in order to reach the final solution (a target-executing input vector). A schema that defines a set of chromosomes which make a step towards the test goal is said to be *fitness-affecting*:

Definition 2 (Fitness-affecting Constraint-Schema)**.** *Let w be the worst fitness value for a structure t in a program p obtained by an input vector drawn from p's input domain. A constraint-schema S is* fitness-affecting *if there does not exist some chromosome $l \in S$ where $fit(l, t) = w$.*

The simplest, most general fitness-affecting schemata encapsulate the building blocks of the test data generation problem.

Definition 3 (Building Block Constraint-Schema). *Let vars(S) be the set of input variables involved in the constraint c of a constraint-schema $S = \{I \mid c\}$. Recall w from Definition 2. A constraint-schema $S_1 = \{V \mid c_1\}$ is said to be a* building block constraint-schema *if it is fitness-affecting (Definition 2) and for all superschemata S_2 of $S_1, S_2 = \{V \mid c_2\}, c_1 \Rightarrow c_2, vars(S_1) \supset vars(S_2)$, there exists some chromosome $l \in S_2, fit(l,t) = w$.*

For example, the covering schema $\{(a,b,c) \mid a = b \wedge b \geq 100 \wedge c \leq 10\}$ for the target of the program of Figure 2, may be generalised into three distinct building block schemata: $\{(a,b,c) \mid a = b\}$, $\{(a,b,c) \mid b \geq 100\}$ and $\{(a,b,c) \mid c \leq 10\}$.

The covering constraint-schema for a target must be generalisable into at least two distinct building blocks in order for crossover to have the opportunity to do any work and contribute to the progress of an ET search. In practice, this is still not enough to guarantee that crossover will have any discernible effect in progressing the search; chromosomes of the covering constraint-schema may be more easily discoverable through mutation. However, it follows that the larger the number of distinct building block constraint-schemata inherent in a test data generation problem the more *opportunity* crossover has to positively impact the progress of the search. This is because there is greater scope for crossover to arrive at the covering constraint-schema through the recombination of building block constraint-schemata; potentially via intermediate schemata that act as stepping stones between building blocks and final solutions.

The number of building block constraint-schemata inherent in the test data generation problem is closely linked to the target's input condition. It follows that the larger the number of conjuncts it has, the larger the number of building blocks may be involved in arriving at a solution.

B. Input condition conjuncts over disjoint sets of variables

If two chromosomes are recombined from two constraint-schemata that reference different input variables in their respective constraints, more specific constraint-schema may be reached by simply copying the genes of input variables referenced in the constraints of each of the original schemata. The constraint-schemata $\{(a,b,c) \mid a = b\}$ and $\{(a,b,c) \mid c \leq 10\}$ for the program of Figure 2, for example, and the input vectors $<a = 10, b = 10, c = 105>$ and $<a = 50, b = 20, c = 5>$ belonging to the former and latter schemata respectively, may be crossed over to produce the offspring $<a = 10, b = 10, c = 5>$, a member of the more specific schema $\{(a,b,c) \mid a = b \wedge c \leq 10\}$.

Recombination is more awkward for *contending* constraint-schemata, however, where one or more input variables are referenced in the respective constraints of each schema.

Definition 4 (Contending Constraint-Schemata). *Recall the function vars(S) from Definition 3. Two constraint-schema S_1 and S_2 are said to be* contending *if their constraints reference one or more of the same input variables, i.e. $vars(S_1) \cap vars(S_2) \neq \emptyset$.*

The ability of the crossover operator may be impaired if contending superschemata are inherent in a test data generation problem. The target of the program of Figure 2, for example, involves the contending schemata $R_1 = \{(a,b,c) \mid a = b\}$ and $R_2 = \{(a,b,c) \mid b \leq 0\}$, which both contain references to the variable b in their respective constraints. Crossover of chromosomes of these schemata is not guaranteed to respect the conjunction of their constraints. For example, recombination of $<a = 10, b = 10, c = 5>$ of R_1 and $<a = 50, b = 20, c = 105>$ of R_2 cannot result in offspring that satisfy $a = b \wedge b \leq 0$.

As such, for a target structure to have a fitness landscape easily exploitable by crossover, not only should the number of input condition conjuncts be numerous, but the conjuncts should involve disjoint sets of variables. This further implies that the input vector to the program under test should also be numerous.

C. Difficulty of discovering input variable values

The harder the input values belonging to fitness-affecting constraint-schemata are to discover, the scarcer they will be in any given generation of a test data search. This situation favours the crossover operator, since it has the capability to build larger pieces of a solution from building blocks that are in existence across different chromosomes in the population. The chances of mutation generating solutions containing from all the required building blocks for an individual chromosome are much smaller. However, if the discovery of individual building blocks is too hard, the genetic material will not come into existence in the first place for crossover to then be able to make use of it.

The difficulty of discovering chromosomes belonging to a constraint-schema depends on two factors. The first factor is the probability of generating inputs at random that satisfy the schema's constraint. Some types of constraint are easy to satisfy at random, *e.g.* the constraint $a > b$ for two input variables a and b of type `int`, which has a probability of 0.5 of being satisfied randomly. A constraint such as $a = 0$, however, is harder to satisfy through random generation of values for a as its domain size increases.

The second factor is the shape of the fitness landscape. Where the landscape has a smooth gradient, providing the search with good guidance to inputs satisfying the constraint, the search will easily find the required input values (genes), regardless of the probability of finding inputs by pure chance. Figure 3 shows a program which takes an array as input. When an individual array value is in a certain range, the `count` variable is incremented. When every value of the array is in range, *i.e.* `count` equals the size of the array, the target branch is executed. The first variant of the program increments the `count` variable by a whole amount,

resulting in a fitness landscape with flat steps down to the global optimum, as depicted in part B of the figure. Array elements are found to be in range through the pure trial and error of mutation. As such the ratio of 'in-range' values to the size of the domain of each array element decides the rate of success the mutation operator will have in finding building block genes that contribute to the overall final solution. The alternative version of the program increments `count` by an amount proportional to the nearest in-range value. This feeds into the branch distance calculation of the target branch, and the corresponding fitness landscape (part C of the figure) is instead formed of downward gradients to the global optimum. The target of the program is executed under exactly the same conditions, yet this variant of the program will ultimately be 'easier' for the search than its counterpart.

D. Absence or Limited Use of Nesting and Short-Circuiting

Nested structures and conditions composed using short-circuiting operators are known to cause problems for SBT [15], [16]. This is because the input condition for executing the target is only revealed to the fitness function step-by-step, as each individual condition is satisfied in turn. For the program of Figure 1, for example, inputs cannot be optimised to fulfil the condition $b \geq 100$ until it is reached in the code, *i.e.* the predicate $a = b$ has been satisfied. This is reflected in the fitness function. It is not until $a = b$ is satisfied that later predicates become the subject of the branch distance calculation, and chromosomes close to evaluating them as true are rewarded.

This impacts the crossover operator. Chromosomes satisfying (or close to satisfying) constraint-schemata for 'later' conditionals will not be rewarded by the fitness function, and as such will not proliferate in the population for eventual use in recombination. This can be explained in terms of constraint-schemata and the schema theorem: consider a target $inner$ with input condition c nested inside a structure $outer$ with input condition a. Let b represent the part of the input condition particular to $inner$, *i.e.* $c = a \wedge b$. For crossover to produce inputs to cover $inner$, there must exist constraint-schemata present in the current population $S_a = \{V \mid a\}$ and $S_b = \{V \mid b\}$, which can be combined to produce chromosomes belonging to the required constraint-schemata $S_c = \{V \mid a \wedge b\}$. S_b must contain some chromosome of worst possible fitness, since $\{V \mid \neg a \wedge b\}$, whose chromosomes cannot traverse the initial approach level, is a subschema of S_b. Therefore S_b cannot be fitness-affecting (Definition 2) for $inner$. Consequently, chromosomes of S_b will not be of above average fitness and not prevalent in the population for recombination with chromosomes of S_a. This reasoning can easily be re-phrased for conditions co-joined using the short-circuiting 'and' operator ('`&&`' in C).

Nesting and short-circuiting limit the search to solving one condition at a time, rather than as parallel sub-problems whose solutions may be recombined.

E. Summary Variables and Internal States

Since nesting and short-circuiting can hinder a search, it follows that crossover will be at its most effective if input variables influence one atomic condition guarding a target, rather than being spread over several co-joined or nested conditions. If more than two input variables are to affect one condition, giving rise to multiple building block constraint schemata, those inputs must do so via an intermediate variable. The target branch of Figure 2, for example, is covered under exactly the same circumstances as Figure 1, except that all inputs directly impact one condition guarding the target through the `count` variable. This allows the search to find solutions to each individual sub-condition necessary to cover the target in parallel. This is because input vectors satisfying $b \geq 100$ and $c \leq 10$ are now rewarded by the fitness function regardless of whether other conditions were previously satisfied. The reward comes in the form of a lower branch distance at the branching condition concerned. Crossover may then recombine these input vectors to find an overall solution; *i.e.* a target-executing input vector.

The types of programs that involve large numbers of inputs and utilising intermediate variables, include programs that process large data structures, storing the results in 'summary' variables, or programs with an internal state that is only reached via long sequences of function or method calls. Furthermore, the conditions under which these intermediate values are modified may give rise to coarse landscapes (as with the program of Figure 3A and the landscape in Figure 3B) further favours crossover since the building blocks are more difficult to discover, as discussed in Section III-C.

F. Conclusions of the Theoretical Analysis

In conclusion, the theoretical analysis predicts that for the crossover operator to be most effective in test data searches:

- The overall input condition for a target must be decomposable into a number of sub-problems, whose solutions can be sought in parallel. These partial solutions form building blocks that can be used by crossover. Ideally then, the **target program structure needs to have an input condition composed of several different conjuncts**.
- To avoid clashes of input vector values during crossover, each conjunct should reference different input variables. Therefore the **program needs to have several input variables** to accompany an abundant number of conjuncts.
- Input condition conjuncts cannot be solved in parallel if they are nested, and conjuncts should be relatively hard to solve randomly, suggesting that **inputs affect a single condition guarding the target via internal variables modified under 'special' circumstances**.

Although the theoretical analysis can suggest conditions favourable for crossover, only an empirical study can establish the extent to which they are true. The results of such a study are presented in the next section.

IV. Empirical Study

A. Research Questions

An empirical study was designed to answer a number of research questions to validate the claims of the theoretical analysis presented in the previous section.

Q1: Number and difficulty of input condition conjuncts. The theory predicts that the more conjuncts in the input condition for covering a target, the more crossover will be able to exploit the resulting fitness landscape. Is this the case in practice, and how many conjuncts are required before crossover noticeably impacts the search? The theory also predicts that the greater difficulty of satisfying individual conjuncts, the more beneficial crossover will be to progressing the search. Is this the case in practice, and under what circumstances does this effect occur?

Q2: Input condition conjuncts with non-disjoint sets of variables. The theory predicts that crossover will be less effective for test data generation problems involving 'contending' constraint-schemata. Is this the case in practice?

Q3: Nested conditionals. The theory predicts that nesting prevents input condition conjuncts from being solved in parallel, thus reducing the number of building blocks available to crossover and rendering it less effective. Is this the case in practice?

In addition to those raised by the theoretical analysis, two further research questions were investigated:

Q4: Performance of ET compared to hill climbing. Over a range of different search problems designed to test crossover for ET, how does hill climbing perform? In what situations will ET outperform hill climbing and vice-versa?

Q5: Crossover type. Across a range of different search problems, which type of crossover performs best for ET?

B. Case studies

In order to answer the research questions above, four case studies were designed.

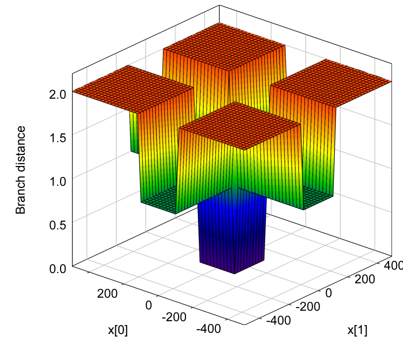
Q1 - Case Study 1 (Figure 3)**.** With this case study, the target branch is executed when every integer in an array is within a certain range. The input conditions conjuncts therefore correspond to individual array elements that are 'in-range', *i.e.* $\{(x[i])|x[i] \geq MIN_RANGE \wedge x[i] \leq MAX_RANGE\}$. The satisfaction of each individual conjunct forms an individual building block for the search.

The domain size of each array element is artificially fixed to 1,000 elements (set from -500 to 499). The variables `MIN_RANGE` and `MAX_RANGE` are adjusted to generate the desired level of probability for generating an input satisfying an input condition conjunct; *i.e.* an 'in-range' array element, a building block for the test data search.

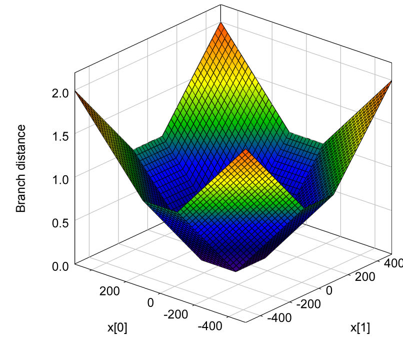
The case study has two variants. When the `count` variable is incremented by whole amounts, a flat fitness landscape results (Figure 3B). In order to investigate the effect of 'easier' landscapes on crossover, a smooth gradient landscape (Figure 3C) can be generated through the inclusion of lines 6-11, where `count` is instead increased by an amount proportional to the distance to the nearest in-range value for a particular array element.

```
(1)   void case_study_1(int x[SIZE]) {
(2)     double count = 0; int i;
(3)     for (i=0; i < SIZE; i++) {
(4)       if (x[i] >= MIN_RANGE && x[i] <= MAX_RANGE)
(5)         count ++;
(6)       else if (x[i] < MIN_RANGE)
(7)         count += 1 - ((MIN_RANGE - x[i]) /
(8)                       (MIN_RANGE - MIN_VAL));
(9)       else
(10)        count += 1 - ((x[i] - MAX_RANGE) /
(11)                      (MAX_VAL - MAX_RANGE));
(12)    }
(13)
(14)    if (count == SIZE) {
(15)        // target branch
(16)    }
(17)  }
```

A. Code



B. Fitness Landscape (lines 6-11 omitted) that is Flat



C. Fitness Landscape (lines 6-11 included) with Downward Gradient

Figure 3. Case study 1: code to assess the impact of different forms of crossover with different numbers of input condition conjunct (through varying array size) and conjunct satisfaction probability (through varying `MIN_RANGE` and `MAX_RANGE`). Lines 6-11 appearing in grey are omitted from the flat landscape version of the experiment, which is pictured for two array elements in part B. Part C depicts the gradient landscape where lines 6-11 are included in the second version of the experiments

Q2 - Case Study 2 (Figure 4)**.** With this case study, the integer elements of the array must be in descending order for the target branch to be executed. The variable `count` keeps track of how many pairs of array elements are in descending order before the branch can be executed. In this way, input condition conjuncts reference one variable appearing in

another; *i.e.* $x[0] > x[1]$, $x[1] > x[2]$. *etc.* As such, the building block schemata are contending. The domain size for each array element is -500 to 499, as for case study 1.

Q3 - Case Study 3 (Figure 5)**.** The target branch of case study 3 is executed under exactly the same circumstances as case study 1, but instead of using a `count` variable, each conjunct of the input condition is nested.

Q4 - Case Study 4 (Figure 6)**.** Case study 4 was deliberately designed as an example to thwart local hill climbing searches. The code is as for case study 1, except that `count` is reset to zero immediately before the array is full of elements that are in-range. This results in a local optimum, as depicted in Figure 6B. ET can overcome the local optimum through crossover, as shown in Figure 6C.

Q5 was assessed using data from each of the above studies.

C. Search Types and Variants Investigated

ET with variants of crossover. ET was applied using the setup described in Section II (with discrete recombination as the crossover operator). To compare ET with and without crossover, ET was also applied without a crossover operator. Parents selected for recombination simply become their offspring. In order to answer Q5, ET was also applied with uniform and one-point crossover.

ET with the Headless Chicken Test (HCT). The 'Headless Chicken Test' (HCT) [17] is used to assess whether GA progress is not merely the result of crossover simply functioning as a macro-mutation operator. The HCT in this paper is ET with discrete recombination, but using a new, randomly-generated individual not drawn from the population, as one of the parents. If ET cannot perform better than the HCT, the search is not actually benefiting from the random exchange of genes between individuals.

Alternating Variable Method (AVM). The AVM was applied as described in Section II.

Random Mutation Hill Climbing (RMHC). With Random Mutation Hill Climbing (RMHC), an input vector is initially selected from the search space at random. Mutations are then by replacing genes with a new value selected at uniform random, with genes mutated at a probability that is the inverse of the chromosome's length. The mutated individual replaces the current individual if it is of improved fitness. When used by Forrest and Mitchell [2], it was found to outperform GAs on Royal Road fitness functions. RMHC is the equivalent of a (1+1) EA.

Random Mutation AVM (RM-AVM). The Random Mutation-Alternating Variable Method (RM-AVM) is a new search novel to this paper, combining the AVM with random mutation re-starts. When the AVM becomes 'stuck', random mutations are made until a better solution is found. RM-AVM therefore incorporates the best features of AVM and RMHC; the ability of the AVM to accelerate down gradients with the ability of RMHC to escape certain local optima.

```
void case_study_2(int x[SIZE]) {
    int count = 0, i;
    for (i=0; i < SIZE-1; i++) {
        if (x[i] > x[i+1])
            count ++;
    }
    if (count == SIZE-1) {
      // target branch
    }
}
```

Figure 4. Case study 2: code to assess the impact of *contending* constraint-schemata. The target branch is executed if the array elements are sorted in descending order. Results with this case study can be found in Table II

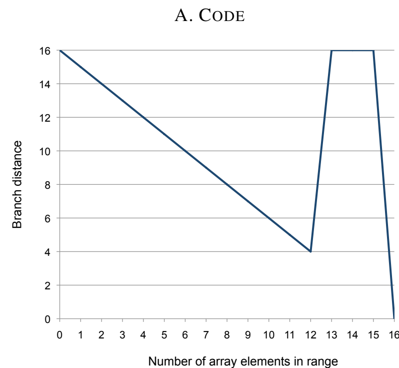
```
void case_study_3(int x[SIZE]) {
  if (x[0] >= MIN_RANGE && x[0] <= MAX_RANGE) {
    if (x[1] >= MIN_RANGE && x[1] <= MAX_RANGE) {
      ...
      // target branch
```

Figure 5. Case study 3: code to assess the impact of crossover on nested structures. The target is executed under exactly the same conditions as case study 1 (Figure 3). Results with this case study can be found in Table III

```
void case_study_4(int x[SIZE]) {
  int count = 0, i;
  for (i=0; i < SIZE; i++) {
    if (x[i] >= MIN_RANGE && x[i] <= MAX_RANGE)
      count ++;
  }
  if (count > SIZE-4 && count < SIZE)
    count = 0;

  if (count == SIZE) {
      // target branch
  }
}
```

A. Code



B. Fitness landscape collapsed to one dimension

C. Crossover of two chromosomes to reach the global optimum

Figure 6. Case study 4: a branch designed to defeat hill climbing but for which ET can succeed with crossover. Part A shows the program code. All integer values in an array must be in a given range to execute the branch, which uses the variable `count` to record the number of array elements for which this is true. However, just before the global optimum is reached, the value of `count` is set to zero, forming a local optimum depicted in part B, a graph plotting branch distance against the number of array values in range (for an array size of 16). ET can successfully execute the branch through crossover of two individuals at the local optimum as illustrated in part C, where an '`O`' represents an array value in range, while an '`X`' represents a value out of range. Results with this case study can be found in Table IV

Table I

RESULTS FOR THE PROGRAM OF FIGURE 3. AS ARRAY LENGTH INCREASES, SO DO THE NUMBER OF INPUT CONDITION CONJUNCTS AND POTENTIAL BUILDING BLOCK CONSTRAINT-SCHEMATA. THE IMPACT OF CROSSOVER IS GREATER AS THE POTENTIAL NUMBER OF BUILDING BLOCKS INCREASES AND THE PROBABILITY OF THEIR RANDOM GENERATION THROUGH MUTATION DECREASES

Success rate (percentage of times the branch was covered over the 50 runs) is reported unless the branch was covered with 100% success, in which case the average number of fitness evaluations is reported. A figure appears in bold if ET (using discrete recombination) was shown to be significantly better than another search when the one-sided Wilcoxon rank-sum test was applied to numbers of fitness evaluations over the respective sets of 50 trials. Conversely, a figure appears in italics if an alternative search was significantly better than ET. Statistical tests were not performed if the success rate for one of the searches fell below 60% (*i.e.* 30 runs).

A. FLAT FITNESS LANDSCAPE

Build. block probability	**Search**	**Array length 2**	**4**	**8**	**16**	**32**	**64**	**128**
0.5	ET	5	14	209	1,247	2,674	4,947	8,660
	ET, uniform crossover	5	14	208	1,228	2,684	*4,882*	*8,062*
	ET, 1-point crossover	5	14	202	**1,498**	**3,902**	**8,985**	48%
	ET, headless chicken test	5	14	215	**4,540**	6%	0%	0%
	ET, no crossover	5	14	2,803	38%	2%	0%	0%
	AVM	**29**	**327**	**10,963**	4%	0%	0%	0%
	RMHC	**6**	14	*56*	*123*	*318*	*775*	*1,819*
	RM-AVM	**22**	**91**	**662**	**2,742**	**13,816**	**98%**	0%
0.2	ET	24	408	1,381	3,103	5,612	9,919	88%
	ET, uniform crossover	24	408	1,417	*2,969*	*5,304*	*8,879*	*14,638*
	ET, 1-point crossover	24	411	**1,713**	**4,339**	**98%**	14%	0%
	ET, headless chicken test	24	431	**7,239**	2%	0%	0%	0%
	ET, no crossover	24	**88%**	24%	0%	0%	0%	0%
	AVM	**214**	**10,709**	0%	0%	0%	0%	0%
	RMHC	21	*74*	*163*	*499*	*1,200*	*2,756*	*6,338*
	RM-AVM	**77**	425	**1,663**	**8,875**	**40,426**	0%	0%
0.1	ET	106	921	2,186	4,339	7,502	96%	10%
	ET, uniform crossover	103	894	2,072	*3,993*	*6,840*	*11,564*	58%
	ET, 1-point crossover	103	919	**2,698**	**6,367**	**70%**	0%	0%
	ET, headless chicken test	106	**1,713**	**29,371**	0%	0%	0%	0%
	ET, no crossover	178	**60%**	6%	0%	0%	0%	0%
	AVM	**617**	56%	0%	0%	0%	0%	0%
	RMHC	*50*	*115*	*394*	*1,027*	*2,702*	*6,116*	14,375
	RM-AVM	**172**	*651*	**3,645**	**17,211**	**78%**	0%	0%
0.01	ET	2,587	94%	78%	46%	2%	0%	0%
	ET, uniform crossover	2,293	94%	*78%*	52%	10%	0%	0%
	ET, 1-point crossover	5,076	94%	**74%**	36%	0%	0%	0%
	ET, headless chicken test	**1,727**	**21,255**	0%	0%	0%	0%	0%
	ET, no crossover	**82%**	16%	0%	0%	0%	0%	0%
	AVM	**86%**	0%	0%	0%	0%	0%	0%
	RMHC	*446*	*1,438*	*4,439*	11,204	27,885	94%	2%
	RM-AVM	1,342	**6,861**	**98%**	16%	0%	0%	0%

B. GRADIENT FITNESS LANDSCAPE

Build. block probability	**Search**	**Array length 2**	**4**	**8**	**16**	**32**	**64**	**128**
0.5	ET	5	14	205	1,388	3,366	6,697	12,143
	ET, uniform crossover	5	14	206	1,319	*3,171*	*6,396*	*11,376*
	ET, 1-point crossover	5	14	215	**1,672**	**5,171**	**12,512**	**36,438**
	ET, headless chicken test	5	14	219	**6,025**	0%	0%	0%
	ET, no crossover	5	14	709	**10,324**	**26,387**	**63,863**	0%
	AVM	**11**	**21**	*42*	*86*	*173*	*345*	*693*
	RMHC	**6**	**15**	*56*	*127*	*339*	*787*	*1,980*
	RM-AVM	**11**	**21**	*42*	*86*	*173*	*345*	*693*
0.2	ET	24	450	1,850	4,536	8,540	16,197	33,393
	ET, uniform crossover	24	394	1,942	*4,245*	*8,147*	*14,528*	*26,256*
	ET, 1-point crossover	24	434	**2,142**	**6,105**	**14,808**	**36,349**	**90%**
	ET, headless chicken test	24	431	**10,666**	2%	0%	0%	0%
	ET, no crossover	24	**1,747**	**9,100**	**24,779**	**57,088**	0%	0%
	AVM	19	*38*	*76*	*157*	*318*	*651*	*1,340*
	RMHC	23	*75*	*192*	*529*	*1,283*	*3,030*	*7,120*
	RM-AVM	19	*38*	*76*	*157*	*318*	*651*	*1,340*
0.1	ET	104	1,227	3,460	7,025	14,393	26,569	53,286
	ET, uniform crossover	107	1,114	*3,092*	*6,696*	*12,113*	*22,483*	*43,717*
	ET, 1-point crossover	105	1,188	**4,033**	**10,435**	**22,862**	**50,760**	6%
	ET, headless chicken test	106	**2,216**	**88%**	0%	0%	0%	0%
	ET, no crossover	129	**4,153**	**12,117**	**31,605**	**71,617**	0%	0%
	AVM	*25*	*49*	*98*	*200*	*401*	*805*	*1,624*
	RMHC	*48*	*126*	*390*	*1,289*	*2,818*	*6,652*	*16,164*
	RM-AVM	*25*	*49*	*98*	*200*	*401*	*805*	*1,624*
0.01	ET	1,876	5,255	10,723	19,746	35,979	66,253	0%
	ET, uniform crossover	1,910	5,054	*9,970*	*17,593*	*32,711*	*57,981*	6%
	ET, 1-point crossover	1,788	5,415	**11,988**	**24,792**	**49,411**	**60%**	0%
	ET, headless chicken test	**2,575**	**37,259**	0%	0%	0%	0%	0%
	ET, no crossover	**3,172**	**9,755**	**23,919**	**56,270**	0%	0%	0%
	AVM	*38*	*76*	*152*	*315*	*632*	*1,285*	2,632
	RMHC	*618*	*1,894*	*4,859*	*13,597*	34,947	**82%**	0%
	RM-AVM	*38*	*76*	*152*	*315*	*632*	*1,285*	2,632

D. Experimental setup

Each experiment was run 50 times using an identical list of random seeds. Each search method was given a budget of 100,000 fitness evaluations. If test data were not found within this limit, the search was deemed to have failed. The **success rate** is the reported percentage of the 50 runs for which a search found test data for a branch. Where success rate is 100% the **average number of fitness evaluations** is reported in each table of results. This figure is the mean number of fitness evaluations that were taken to find test data. Statistical tests were performed with the non-parametric Wilcoxon rank-sum test to compare search performance using numbers of fitness evaluations obtained from each of the 50 runs.

E. Answers to Research Questions

Q1: Number and difficulty of input condition conjuncts. Case study 1 (Figure 3) was run with different array sizes; 2, 4, 8, 16, 32, 64 and 128. A domain of -500 to 499 for each array element, coupled with settings of (`MIN_RANGE`, `MAX_RANGE`) as (-250, 249), (-100, 99), (-50, 49), (-5, 4) and (0, 0) respectively, enabled the testing of different probabilities of finding an array element in range from 0.5 down to 0.01. Each setup was run with both variants of the code to produce flat and gradient landscape types.

The results can be found in Table IA. They show that as the length of the array increases (and the number of input condition conjuncts and potential building block constraint-schemata), the search not only becomes harder (as evidenced by higher average numbers of fitness evaluations and lower success rates) but also the impact of crossover increases.

At a building block generation probability of 0.5, ET with crossover is always 100% successful at generating test data. With small array sizes, test data is generated easily at random, the average number of fitness evaluations is not greater than 300, *i.e.* inputs were found within the first generation on average.

For larger array sizes, the HCT and ET without crossover were not always 100% successful. Where they were, ET with crossover had statistically significantly better performance, with the difference in performance becoming greater and in favour of ET with crossover as array size increases. For large array sizes, the HCT and ET without crossover always

Table II
RESULTS FOR THE PROGRAM OF FIGURE 4 WITH 'CONTENDING' CONSTRAINT-SCHEMATA*

Search	**Array length**								
	4	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
ET	22	125	399	1,202	96%	88%	66%	44%	32%
ET, uniform crossover	22	126	391	1,074	2,023	*84%*	78%	48%	38%
ET, 1-point crossover	22	119	385	*977*	*98%*	*96%*	*68%*	56%	36%
ET, headless chicken test	22	122	434	**2,072**	**9,984**	**90%**	**62%**	26%	6%
ET, no crossover	22	896	64%	26%	10%	10%	2%	0%	0%
AVM	**461**	**2,421**	**19,497**	42%	12%	0%	0%	0%	0%
RMHC	**32**	*75*	*198*	*324*	*810*	*2,669*	*4,300*	9,947	12,685
RM-AVM	**229**	**585**	**1,753**	**3,075**	**8,240**	**98%**	**84%**	70%	48%

Table III
RESULTS FOR THE PROGRAM OF FIGURE 5 WITH NESTING*

Build. block prob.	**Search**	**Array length**						
		2	**4**	**8**	**16**	**32**	**64**	**128**
0.5	ET	5	14	220	2,308	9,985	52%	0%
	ET, uniform	5	14	200	2,174	*8,334*	76%	0%
	ET, 1-point	5	14	211	**3,014**	**35,493**	0%	0%
	ET, HCT	5	14	221	**8,480**	0%	0%	0%
	ET, no crossover	5	14	1,289	**22,449**	**84%**	0%	0%
	AVM	**11**	**21**	*42*	*86*	*173*	345	693
	RMHC	**6**	**19**	*99*	*302*	*1,312*	5,182	21,281
	RM-AVM	**11**	**21**	*42*	*86*	*173*	345	693
0.01	ET	3,139	14,778	45,149	0%	0%	0%	0%
	ET, uniform	3,102	*12,528*	43,668	0%	0%	0%	0%
	ET, 1-point	3,268	14,056	**52,174**	0%	0%	0%	0%
	ET, HCT	*2,365*	**22,237**	0%	0%	0%	0%	0%
	ET, no crossover	**4,796**	**17,050**	**58,131**	0%	0%	0%	0%
	AVM	*55*	*137*	*394*	1,300	4,598	17,052	65,592
	RMHC	*526*	*2,578*	*10,818*	45,337	2%	0%	0%
	RM-AVM	*55*	*137*	*394*	1,300	4,598	17,052	65,592

Table IV
RESULTS FOR THE PROGRAM OF FIGURE 6 WITH LOCAL OPTIMUM*

Search	**Array length**			
	16	**32**	**64**	**128**
ET	2,923	4,087	6,201	10,485
ET, uniform crossover	2,912	3,761	6,010	*9,455*
ET, 1-point crossover	**70%**	54%	22%	2%
ET, headless chicken test	**20,236**	0%	0%	0%
ET, no crossover	0%	0%	0%	0%
AVM	4%	0%	0%	0%
RMHC	6%	2%	0%	0%
RM-AVM	0%	0%	0%	0%

*Success rate (percentage of times the branch was covered over the 50 runs) is reported unless the branch was covered with 100% success, in which case the average number of fitness evaluations is reported. A figure appears in bold if ET (using discrete recombination) was shown to be significantly better than another search when the one-sided Wilcoxon rank-sum test was applied to numbers of fitness evaluations over the respective sets of 50 trials. Conversely, a figure appears in italics if an alternative search was significantly better than ET. Statistical tests were not performed if the success rate for one of the searches fell below 60% (*i.e.* 30 runs).

failed to find test data.

As the probability of building block generation decreases, a similar pattern emerges; ET with crossover has an increasingly higher success rate than the HCT and ET without crossover, or, the average number of fitness evaluations for ET with crossover is significantly better, with the difference becoming greater in favour of ET with crossover.

The search becomes easier for all variants of ET when the landscapes has a downward gradient, as seen in part B of Table I, as opposed to when it is flat (part A). However, the effect on crossover follows the same pattern. While average numbers of fitness evaluations are lower (and success rates higher), ET with crossover is increasingly better than the HCT and ET without crossover, or, it achieves an increasingly higher success rate.

The empirical results therefore confirm the theoretical prediction that the higher number of input condition conjuncts, the more useful crossover can be. Also, as predicted, crossover becomes more useful as a search operator the greater the difficulty of generating building blocks.

Q2: Input condition conjuncts with non-disjoint sets of variables. Case study 2 (Figure 4) was run with a domain size of -500 to 499 for each array element (the probability of generating a building block at random is thus fixed at approximately 0.5), with array sizes of 4-12. Table II reports the results. All searches struggle as array size increases, due to the problem becoming more tightly constrained. Despite the presence of contending schemata, ET still outperforms the HCT and ET without crossover with an increasing margin as array size increases (*i.e.* the number of building blocks increase), and is statistically significantly better in many cases.

Thus, in conclusion, contending schemata may reduce the effectiveness of crossover, but the inclusion of the crossover operator may still result in a significantly better evolutionary search.

Q3: Nested conditionals. Case study 3 (Figure 5) was run with the same array sizes and building block generation probabilities as study 1. Table III reports the results for probabilities of 0.5 and 0.01. Although the input condition for reaching the target branch is identical to that of study 1, all searches clearly have difficulty in finding test data, due to nesting. This can easily be seen by comparing the results in Table III with those for study 1 reported in Table I.

Perhaps surprisingly, crossover still has a discernible impact on the search that is statistically significant in many cases. Crossover seems to allow the population to be filled quickly with many 'good' solutions for the current approach level, which increase the probability of it being penetrated through later mutations. When building block probability is low, there are fewer building blocks in existence, and crossover cannot fill the population so quickly. This leads to a smaller margin of increased performance for ET, in contrast with crossover's behaviour when conditionals are not nested (as with the answer to research question 1). At a probability level of 0.01, the aggressive mutation involved in the HCT leads it to significantly outperform ET at the small array size of 2.

In conclusion, although nesting limits the crossover operator, crossover still has a useful role to play in finding test data by causing 'good' genes to proliferate in the population.

Q4: Performance of ET compared to Hill Climbing. As expected, the AVM performs poorly whenever the fitness

landscape is flat, but is significantly superior to all other searches when a gradient exists. The only exception is the descending sort-check branch of case study 2, where the search becomes stuck due to poor starting positions, coupled with an inability to change the value of more than one input at a time. RMHC, on the hand, performs well on flat landscapes and case study 2, and is significantly better when compared to ET. The RM-AVM, being a combination of RMHC and the AVM, always performs somewhere between the two. For studies 1-3, it is always the case that at least one of the hill climbers outperforms ET with crossover.

Case study 4 was specifically designed to contain a local optimum (Figure 6), as a 'proof of concept' that cases can exist where ET will outperform both the AVM and RMHC (whether such code exists for 'real' is another issue). The results are shown in Table IV with domain size as for case study 1 and a building block generation probability of 0.5. As illustrated in Figure 6C, it is possible for ET to reach the global optimum through crossover of two individuals on the edge of the local optimum. Conversely, the chasm between optima is bridged by mutation alone with an extremely low probability, resulting in a low success rate for RMHC.

The conclusion for this research question, therefore, is that test data generation problems for 'crossover-friendly' programs are not necessarily better solved by ET than a hill climber. Cast study 4, however, does show that test data generation problems can exist where ET can find test data, but which are highly challenging for hill climbers.

Q5: Crossover type. Over all of the case studies, uniform crossover is significantly better than discrete recombination on exactly 35 occasions. Conversely, discrete recombination never outperforms uniform crossover. It seems that discrete recombination is responsible for destroying building blocks, slowing down the progress of the search. Whereas with uniform crossover, genes are always preserved in the offspring, discrete recombination may opt to copy a gene into both children and destroy the other.

Discrete recombination usually outperforms one-point crossover, with the exception of four configurations of case study 2 and the sort-check branch. Adjacent array values are dependent on one another, due to 'contending' building block constraint-schemata. As such discrete recombination can destroy some of the local context through the exchange of short sequences of genes, some of which can be preserved by less-disruptive one-point crossover.

In conclusion, the results strongly indicate that uniform crossover is a better choice of crossover operator than discrete recombination for test data generation. One-point crossover is best suited for test data generation problems involving a high degree of cohesion across input variables.

V. Conclusions

This paper has investigated the types of program structure that cause the crossover operator to progress the search for Evolutionary Testing (ET), both theoretically and empirically. The paper found that program structures executed by an input condition with a high number of conjuncts, each of which are hard to satisfy, result in fitness landscapes that are more easily exploitable by crossover than through mutation alone. This lends ET to programs that process large data structures or have internal states reached through sequences of function or method calls. Although nesting hinders a test data search, crossover may still be useful. Hill climbers can also be efficient on these programs, however program structures exist that result in entrapping local optima, for which local searches are not very effective. Finally, it was found that the discrete recombination operator of ET does not represent the best crossover operator for test data searches, as it is frequently outperformed by uniform crossover.

Acknowledgement. Phil McMinn is supported in part by EPSRC grants EP/G009600/1 and EP/F065825/1.

References

[1] M. Mitchell, S. Forrest, and J. H. Holland, "The royal road for genetic algorithms: Fitness landscapes and GA performance," *Proc. 1st European Conference on Artificial Life*. MIT Press, 1992, pp. 245–254.

[2] S. Forrest and M. Mitchell, "Relative building-block fitness and the building-block hypothesis," *Proc. Foundations of Genetic Algorithms*. Morgan Kaufmann, 1993, pp. 109–126.

[3] R. A. Watson, G. S. Hornby, and J. B. Pollack, "Modeling building-block interdependency," *Proc. PPSN V*. Springer, 1998, pp.97–106.

[4] T. Jansen and I. Wegener, "Real royal road functions - where crossover provably is essential," *Discrete Applied Mathematics*, vol. 149, pp. 111–125, 2005.

[5] J. N. Richter, A. Wright, and J. Paxton, "Ignoble trails - where crossover is provably harmful," *Proc. PPSN X*. Springer, 2008, pp. 92–101.

[6] A. Arcuri, P. K. Lehre, and X. Yao, "Theoretical runtime analyses of search algorithms on the test data generation for the triangle classification problem," *SBST workshop 2008, Proc. ICST 2008*. IEEE, 2008, pp. 161–169.

[7] A. Arcuri, "Longer is better: On the role of test sequence length in software testing," *Proc. ICST 2010*. IEEE, 2010, pp. 469–478.

[8] M. Harman and P. McMinn, "A theoretical & empirical analysis of evolutionary testing and hill climbing for structural test data generation," *Proc. ISSTA 2007*. ACM, 2007, pp. 73–83.

[9] ——, "A theoretical and empirical study of search-based testing: Local, global and hybrid search," *IEEE Transactions on Software Engineering*, vol. 36, pp. 226–247, 2010.

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

[11] B. Korel, "Automated software test data generation," *IEEE Transactions on Software Engineering*, vol. 16, no. 8, pp. 870–879, 1990.

[12] D. Whitley, "The GENITOR algorithm and selection pressure: Why rank-based allocation of reproductive trials is best," *Proc. ICGA 1989*. Morgan Kaufmann, 1989, pp. 116–121.

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

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

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

[16] A. Baresel, H. Sthamer, and M. Schmidt, "Fitness function design to improve evolutionary structural testing," *Proc. GECCO 2002*. Morgan Kaufmann, 2002, pp. 1329–1336.

[17] T. Jones, "Crossover, macromutation and population-based search," *Proc. ICGA '95*. Morgan Kaufmann, 1995, pp. 73–80.