# An Identification of Program Factors that Impact Crossover Performance in Evolutionary Test Input Generation for the Branch Coverage of C Programs

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

**Abstract**

Genetic Algorithms are a popular search-based optimisation technique for automatically generating test inputs for structural coverage of a program, but there has been little work investigating the class of programs for which they will perform well. This paper presents five program factors that affect the performance of crossover, a key search operator in Genetic Algorithms, when searching for inputs that cover the branching structure of a C function.

The paper finds that crossover plays an increasingly important role for programs with large, multi-dimensional input spaces, where the target structure's input condition breaks down into independent sub-problems for which solutions may be sought in parallel. Furthermore, it is found that crossover can be inhibited when the program under test is unstructured or involves nested conditional statements; and when intermediate variables are used in branching conditions, as opposed to direct input values. Each program factor is demonstrated with empirical experiments. The paper presents further empirical experiments that evaluate different types of crossover and the conditions under which Genetic Algorithms will outperform local search techniques, because of crossover.

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

## 1 Introduction

Software testing remains an extremely costly activity in the software engineering lifecycle, and as such, its automation continues to be of high concern. Search-Based Test Data Generation [25] is a means of automatically generating inputs according to a testing criterion, such as inputs that execute all of a program's branches or statements. The testing criterion of interest is expressed as a 'fitness function', which scores inputs on the basis of how close they were to fulfilling the test goal currently under consideration. The fitness function is used by a search-based optimisation technique to evaluate points in the search space (the program's input domain) and guide it to the required test inputs.

The use of Genetic Algorithms as an optimisation technique for test data generation, more commonly referred to as 'Evolutionary Testing' [34, 25], has received much attention in the literature since 1992, when their application was first proposed by Xanthakis *et al.* [36]. One key feature of Genetic Algorithms is the use of crossover as a search operator. The crossover operator helps explore the search space of a problem by generating new candidate solutions through the recombination of the components of sufficiently good, previously evaluated candidate solutions. It is the question of the appropriateness and effectiveness of crossover for the structural test data generation for different types of C program that motivates the investigation in this paper. The question is a pertinent one. Recent studies by Harman and McMinn [14, 15] found that for the majority of branches considered, Evolutionary Testing was *not* the most efficient search technique for the branch coverage of C functions. Nevertheless, a small number of cases existed where Evolutionary Structural Testing was able to find test inputs where other search methods failed. In these instances it was found that the crossover operator was instrumental in test data discovery. However, there has been, until now, little work investigating the factors behind a program,

including its code structure and input domain, that determine whether crossover will be useful in the Evolutionary Search for test inputs.

This paper seeks to address this concern. Five program factors are identified, which are predicted to affect crossover through an outworking of existing theory for Evolutionary Structural Testing. An empirical study is then performed with a set of programs designed to demonstrate the affect of each program factor on crossover performance in practice. The empirical study also investigates the different choices available for a crossover operator, and further evaluates the conditions necessary for Genetic Algorithms, with crossover, to outperform Hill Climbing.

The specific contributions of this paper, therefore, are as follows:

1. The identification of a set of five program factors predicted to affect the performance of the crossover operator;

2. An empirical study showing how crossover is affected for each program factor in practice;

3. An empirical assessment of the various choices available for a crossover operator in Evolutionary Testing, and the operator that works best for each of the five case studies considered. The results show that the discrete recombination operator commonly employed by Evolutionary Testing is often outperformed by standard uniform crossover.

4. Further empirical analysis comparing variants of Hill Climbing local searches against Evolutionary Testing on the case studies designed to test the effectiveness of crossover. At least one variant of Hill Climbing is found to be more efficient for each of the original five case studies, but a sixth case study demonstrates that programs do exist for which Hill Climbing cannot find test data but Evolutionary Testing can, because of crossover.

The rest of this paper is organized as follows. Section 2 introduces Search-Based Test Data Generation and the fitness function used for structural testing. It then goes on to introduce the Evolutionary Testing approach for generating structural test inputs using Genetic Algorithms. The section also introduces key theoretical concepts for Evolutionary Structural Testing. It is the outworking of these theoretical concepts that form the basis for the identification of each program factor predicted to affect the performance of crossover. These program factors are presented in Section 3. Section 4 then details empirical experiments demonstrating the affect on crossover with each program factor using a set of case studies designed to be exemplars of those factors. Finally, Section 6 discusses related work while Section 7 closes with concluding remarks and avenues for future work.

# 2 Search-Based Test Input Generation for the Branch Coverage of C Programs

This section introduces the automatic generation of test inputs for branch coverage of functions written in the C language, using search-based techniques. This paper focuses on the use of Genetic Algorithms as a search method, in a technique referred to as Evolutionary Structural Testing.

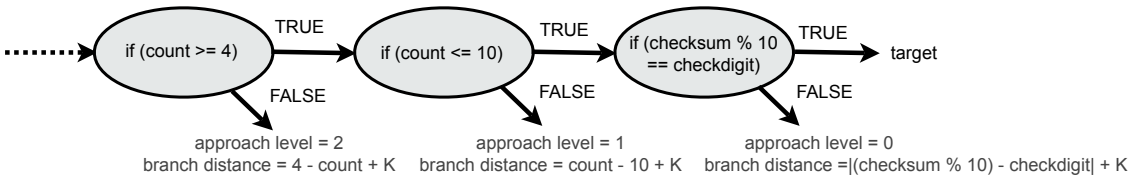
In order to find an test data that executes a branch, the goal of the search is to find an input vector that takes a path which is driven down the branch of interest. The space of candidate solutions in which the search operates is the input domain of the function under test. (In this paper, only fixed-length input vectors are considered – although this is not a constraint of the technique in general – see references [21] and [22].)

## 2.1 Basic concepts

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

```
(1)   int cas_check(char* cas) {
(2)     int count = 0, checksum = 0, checkdigit = 0, pos;
(3)
(4)     for (pos=strlen(cas)-1; pos >= 0; pos--) {
(5)       int digit = cas[pos] - '0';
(6)
(7)       if (digit >= 0 && digit <= 9) {
(8)         if (count == 0)
(9)           checkdigit = digit;
(10)        if (count > 0)
(11)          checksum += count * digit;
(12)
(13)        count ++;
(14)      }
(15)    }
(16)
(17)    if (count >= 4)
(18)      if (count <= 10)
(19)        if (checksum % 10 == checkdigit)
(20)          return 0;
(21)        else return 1;
(22)      else return 2;
(23)    else return 3;
(24)  }
```

(a) Code




(b) Fitness function calculation for executing the true branch from line 19

Figure 1: C routine for demonstrating how the fitness function of Evolutionary Structural Testing works. The function validates CAS registry numbers of chemical substances. For example, '7732-18-5' is the CAS number of water

## 2.2 Fitness function

Search-based optimization techniques find test data through the use of a fitness function. The purpose of the fitness function is to guide the search into promising, unevaluated areas of a potentially vast input domain. For covering individual branches, the fitness function is a function $fit(t, \mathbf{i}) \rightarrow \mathbb{R}$, that takes a target branch $t$ and individual input $\mathbf{i}$. Upon executing the program with the input $\mathbf{i}$, the fitness function returns a real number scoring 'how close' the input was to executing it. The calculation of the fitness function involves two components [34]; the so-called *approach level* and the *branch distance*.

As an example of a fitness function computation for a branch, take the C function of Figure 1. It is a program for evaluating a Chemical Abstracts Service (CAS) registry number assigned to chemicals. Each number is a string of digits separated by hyphens, with the final digit serving as a check digit. The routine takes a pointer to the first character of the string, processes it, and returns zero if the number is valid. An error code is returned in the case the number is not valid.

The approach level assesses the path taken by the input with respect to the target branch by counting the target's control dependencies that were not executed by the path. For structured programs, the approach level reflects the number of unpenetrated levels of nesting levels surrounding the target. Suppose a string is required for the execution of the true branch from line 19, *i.e.* where the string corresponds to a valid registry number. With this target, the approach level will be 2 if no invalid characters are found in the string, but there are too few digits in the string to form a valid CAS number, and the false branch is taken at line 17. If instead the string has too many digits, the true branch is taken at node 17, but the target is then missed because the false branch was taken at node 18, and the approach level is 1. When the checksum calculation is reached at line 19, the approach level is zero.

When execution of a test case diverges from the target branch, the second component, the *branch distance*, expresses how close an input came to satisfying the condition of the predicate at which control flow for the test case went 'wrong'; that is, how close the input was to descending to the next approach level. For example, suppose execution takes the false branch at node 17 in Figure 1, but the true branch needs to be executed. Here, the branch distance is computed using the formula $4 - count + K$, where $K$ is a constant added when the undesired, alternate branch is taken. The closer `count` is being greater than 4, the 'closer' the desired true branch is to being taken. A different branch distance formula is applied depending on the type of relational predicate. In the case of `y >= x`, and the `>=` relational operator, the formula is $x - y + K$. For a full list of branch distance formulae for different relational predicate types, see Tracey *et al.* [32].

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

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

Since the maximum branch distance is generally not known, the standard approach to normalization cannot be applied [1]; instead the following formula is used:

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

## 2.3 Evolutionary Structural Testing

The search-based approach to test data generation is very general, since different fitness functions can be applied for the generation of test data satisfying different testing criteria, while the same fitness function can be used with different search-based optimisation techniques. This section describes the Evolutionary Testing approach to generating test inputs, using Genetic Algorithms. A detailed overview is provided, along with the parameters used, in order to facilitate replication of the empirical study in Section 4.

The Genetic Algorithm used for Evolutionary Structural Testing in this paper is based on careful replication of the approach described by Wegener *et al.* [34]. This approach formed part of Daimler-Chrysler's Evolutionary Testing system, which has been widely studied in the literature [5, 4, 11, 23]. It has been developed over the period of a a decade, and so can be argued to be the 'state of the art' for Evolutionary Testing of procedural C code.

Whereas traditional Genetic Algorithms use a binary encoding of candidate solutions, referred to as *individuals* or *chromosomes*, Evolutionary Testing optimizes input vectors directly, with input values forming the 'genes' of each chromosome. The Genetic Algorithm maintains 300 individuals at any one time in a population. An overview of the main steps of the Genetic Algorithm can be seen in Figure 2.

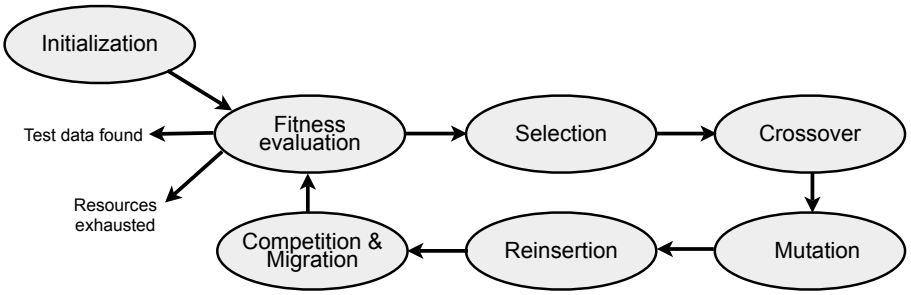



Figure 2: Overview of the Genetic Algorithm used for Evolutionary Testing in this paper

*Initialization.* In the initialisation stage, each of the 300 individuals are initialized to random values, and divided equally over 6 subpopulations of 50 individuals each.

*Fitness evaluation.* The current population is evaluated for fitness, using the fitness function described in the previous section.

*Selection.* Individuals are then selected for the following stages of crossover and mutation. Individuals are ranked according to fitness and assigned an intermediate fitness value based on their rank. Intermediate values are assigned using linear ranking [35], with the best individual receiving a value of $Z$, the median individual a value of 1, and the worst individual receiving a value of $2 - Z$, $1.0 \geq Z \leq 2.0$. The Wegener model uses a value of $Z = 1.7$. Parents are then chosen two at a time for crossover using stochastic universal sampling [2], such that each individual has a probability of being selected proportionate to its intermediate linearly-ranked fitness value.

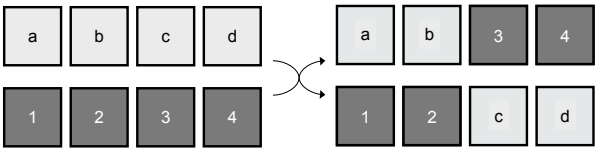
*Crossover.* Figure 3 summarises the two main forms of crossover (one-point crossover and uniform crossover), and that used in the Wegener Genetic Algorithm (discrete recombination), giving examples of each. One-point crossover (part a of the figure) involves splicing the chromosomes at a randomly-chosen crossover point. Uniform crossover (part b) is a less rigid form of crossover, where each point in the chromosome is a potential crossover point. Discrete recombination, as used in the Wegener Genetic Algorithm for Evolutionary Structural Testing, is similar to uniform crossover. With uniform crossover, however each gene from each parent is always copied into exactly one of the offspring, a decision made with an even probability. This differs from discrete recombination, where a gene may instead be copied into both children, the first or second child, or neither child, decided with an equal probability. An example of discrete recombination can be seen in part c of Figure 3.

*Mutation.* Offspring are then mutated in the mutation phase. The Breeder Genetic Algorithm [30] 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 $[0, 10^{-p}]$.

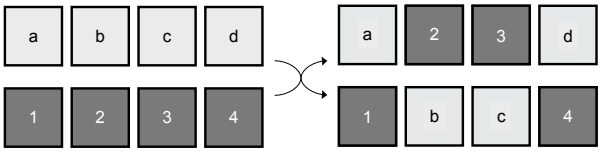
*Reinsertion.* The reinsertion phase then takes place, where the next generation of individuals is created from the current population, and mutated offspring. An elitist reinsertion mechanism is applied, with the top 10% of the current generation retained, while the remaining individuals are discarded and replaced with the best offspring.

*Competition and migration.* Finally, individuals are transferred across subpopulations in the competition and migration phase. 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 $g^{th}$ generation, where $r$ is the subpopulation's average ranked fitness following linear ranking of its individuals using $Z = 1.7$. After every $4^{th}$ 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. In addition, individuals migrate every $20^{th}$ generation, with subpopulations exchanging 10% of their individuals at random.

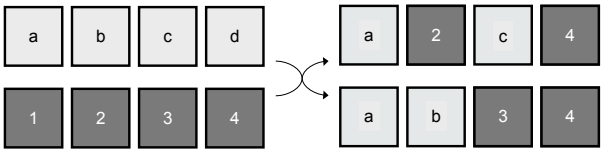
Each individual in the new generation is then evaluated, and the loop continues until test data is found, or resources are exhausted; usually decided by an upper bound on the number of fitness evaluations of the number of generations (*i.e.* cycles of the Genetic Algorithm loop) that may take place.




(a) One point crossover between position 2 and 3




(b) Uniform crossover; every position is potentially a crossover point




(c) Discrete recombination; a gene from one parent can appear in both children

Figure 3: Examples of crossover with different crossover types. The chromosomes are strings of characters, with parents shown on the left of the figure, and created offspring on the right. One-point and uniform crossover are two very common forms of crossover. Discrete recombination is a less common variant of uniform crossover, but the crossover operator employed in the Wegener Genetic Algorithm for Evolutionary Structural Testing

## 2.4 Theoretical Foundations

The theoretical principles developed for Evolutionary Structural Testing by Harman and McMinn [14, 15] are grounded in Genetic Algorithm theory, and Holland's Schema Theorem [17]. Holland's Schema Theorem is based on binary string encodings of chromosomes. A *schema* is an abstraction for a set of chromosomes. A schema is defined a binary string in which some of the elements are 'wildcards'. Wildcards are denoted by an asterisk symbol. For example, `10*1` is an example of a schema. The first two positions and the last position of each chromosome in the set is fixed, but the third gene may freely be either a `1` or a `0`. Thus the set of chromosomes defined by the schema contains two elements, `1001` and `1011`. The *length* of a schema is the distance between the first and last fixed positions; for example, the length of `10*1` is 3, and the length of `*11*` is 1. The *order* of a schema refers to its number of fixed positions; for example the order of `10*1` is 3, while the order of `*11*` is 2.

Holland's theory contends that during fitness evaluation of a chromosome, several schemata are being evaluated at once. The *Schema Theorem* argues that schemata of above average fitness (*i.e.* the average fitness of chromosomes defined by the schema) will increase exponentially in successive generations of the search. The Schema Theorem is also used to justify the so-called *building block hypothesis*, the argument that a Genetic Algorithm should work well when it can combine short, low-order schemata (*building blocks*) to form better candidate solutions [9, 7]. This is because short, low-order schemata are less susceptible to disruption by crossover and mutation.

### Constraint Schemata

Since Evolutionary Testing optimizes input vectors rather than binary string, Harman and McMinn [14, 15] introduced the concept of *constraint schemata* in order to engineer a more formal understanding of how Evolutionary Structural Testing works. The notion of a constraint schema is a generalisation of Holland's binary Genetic Algorithm schema. Whereas binary Genetic Algorithm 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 4a, for example, include $P_1 = \{(a, b, c) \mid a = b\}$ and $P_2 = \{(a, b, c) \mid a = 0\}$. The *order* of a constraint schema is the arity of the schema's constraint, *i.e.* the number of variables it references. The order of the constraint schema $\{(a, b, c) \mid a = b\}$ is 2, while the order of $\{(a, b, c) \mid c = 0\}$ is 1. The *length* of a constraint schema refers to the distance between the first and last constraint variables in the input vector of the program. The length of the schema $\{(a, b, c) \mid a = 0 \wedge c = 0\}$ is 1. The *size* $|c|$ of a constraint schema $c$ is the number of chromosomes in the set that it defines. Since constraint schemata are just generalisations of binary schemata, a binary schema can also be represented in this form, *e.g.* `1*01` may be represented as $\{(l_1, l_2, l_3, l_4) \mid l_1 = 1 \wedge l_3 = 0 \wedge l_4 = 1\}$ [14, 15]. Constraint schemata, as with their binary predecessors, allow for reasoning about the fitness of chromosomes that they define. 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.

Harman and McMinn [14, 15] show that the crossover operator will work well for Evolutionary Testing when a 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 Evolutionary Testing. Higher fitness, specific schemata are modelled through the conjunction of the constraints of more general schemata. For example, with the program of Figure 4c, 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 recombined 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 [15] found only 8 branches that were found to allow the crossover operator to work effectively. As such, although constraint schemata give a clearer picture regarding the type of fitness function that will allow crossover to work effectively for Evolutionary Testing, the issue of what types of programs result in such fitness functions is less well understood. This is the subject under investigation for

```
void nested(int a,
            int b,
            int c) {

  if (a == b) {
    if (b >= 100) {
      if (c <= 10) {
        // target branch
      }
    }
  }
}
```

(a) Nested

```
void jumps(int a,
           int b,
           int c) {

  if (a != b) return;
  if (b < 100) return;

  if (c <= 10) {
    // target branch
  }
}
```

(b) Containing jumps

```
void counter(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
  }
}
```

(c) Using a counter

Figure 4: A function written in three different ways. The target is always executed by the input condition $a = b \wedge b \geq 100 \wedge c \leq 10$. However, only (c) involves all input variable values being evaluated. This allows building block constraint schemata to form, giving the crossover operator opportunity to do useful work

the remainder of this paper, using the theoretical underpinnings introduced in this section together with empirical experiments using the state of the art Wegener model for Evolutionary Structural Testing of procedural C code.

# 3 Program Factors Predicted to Affect the Performance of the Crossover Operator

This section identifies five program factors that are predicted to affect crossover performance in test input searches. The identification of each program factor is based on an outworking of the building block hypothesis involving constraint schemata, introduced in the last section. The potential for each feature to affect the performance of crossover is then empirically assessed in the next section. First, however, some preliminary definitions are given.

## 3.1 Preliminary Definitions

Whether a given input vector executes a certain structure in a program depends on whether the inputs satisfy a certain condition over the variables of the program input vector. This condition is referred to as the *input condition*.

**Definition 1** (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 targets of the programs of Figure 4, 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 2** (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).*

The covering constraint schema may be generalizable into a number of distinct superschemata; the constraints of which denote simpler sub-test data generation problems that are individually solvable.

Any schema that defines a set of chromosomes that are rewarded by the fitness function is said to be *fitness-affecting*:

**Definition 3** (Fitness-affecting Constraint Schema)**.** *Let $w$ be the worst fitness value for covering 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 *for a target $t$ 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 input generation problem:

**Definition 4** (Building Block Constraint Schema)**.** *Recall $w$ from Definition 3. A constraint schema $S_1 = \{I \mid c_1\}$ is said to be a* building block constraint schema *for a target $t$ if it is* a) *fitness-affecting (Definition 3) for $t$;* b) *for all superschemata $S_{sup}$ of $S_1$ there exists some chromosome $l \in S_x, fit(l,t) = w$ and* c) *there exists some other schema $S_2 = \{I \mid c_1\}$ where $S_{sub} = \{I \mid c_1 \wedge c_2\}$ has a higher average fitness than either of $S_1$ and $S_2$ individually.*

In other words, building block constraint schemata capture sets of chromosomes that make the smallest possible impact on fitness for a test data generation problem, that when combined with other schemata, produce subschemata of higher average fitness. The target in the program of Figure 4c involves three distinct building block schemata: $\{(a,b) \mid a = b\}$, $\{(a,b,c) \mid b \geq 100\}$ and $\{(a,b,c) \mid c \leq 10\}$. Any chromosome instantiating one of these schemata will result in the `count` variable being incremented and a lower branch distance at the target.

The following sections investigate how input conditions relate to building block constraint schemata, and the factors of programs that enable or prevent this from happening.

## 3.2 Program Factors

**Program Factor 1. Number of Input Condition Conjuncts**

The covering constraint schema for a target must be generalizable 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 Evolutionary Testing search. It follows, therefore, 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.

If a large number of building block constraint schemata are inherent in the test data search problem, the number of input condition conjuncts must also be large.

**Program Factor 2. Search Difficulty Involved in Satisfying Input Condition Conjuncts**

The presence of multiple input condition conjuncts is not enough, on its own, 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.

The smaller the size of a set defined by a constraint schema, the harder members of the set will be to discover at random. In theory, such a situation favours the importance of a crossover in an Evolutionary Testing search, 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 chances of discovering chromosomes belonging to a constraint schema at random is referred to as the *constraint schema probability*:

**Definition 5** (Constraint Schema Probability)**.** *Let $D$ be the domain of the program containing the target, and $|D|$ and $|c|$ denote the size of the domain and the size of the set of chromosomes defined by the constraint schema $c$, respectively. The probability of the constraint schema $c$ is $\frac{|c|}{|D|}$.*

The size of a constraint schema is heavily dependent on the type of the constraint. Some types constraints have relatively easy to satisfy at random, since there are many inputs from the input domain that may be chosen. For example 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. However there is only one value of $a$ that will satisfy the constraint $a = 0$, and as such the probability of its satisfaction is smaller the larger the size of the domain of $a$.

A further factor influencing the difficulty of discovering input variable values for covering a search target 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 7 shows a program used as one of the case studies appearing in the empirical analysis of the next section. It takes an integer 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 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 Figure 9a. 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 program of Figure 8 shows 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 (Figure 9b) 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, since the search is given more guidance.

**Program Factor 3. Evaluation of the Input Condition with Respect to the Target Structure**

Depending on the structure of the program, the conjuncts of an input condition cannot always decompose into a set of distinct building block constraint schemata. Building block schemata will not form for conjuncts of the input condition that are not evaluated by a program, because there is no reward for them in terms of an improved fitness value. Conjuncts may not be evaluated by programs that involve nested structures, short-circuiting in conditional statements, unstructured jumps and premature termination of looping constructs.

*Nested structures.* Nested program statements lead to partial evaluation of the input condition. For the program of Figure 4a, 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.

*Short-circuit evaluation.* The use of short-circuit evaluation through operators such as '`&&`' and '`||`' in languages such as C and Java have a similar effect to nested control structures. As an example, the input condition for the target of Figure 4 could be rewritten as one if statement '`if (a == b && b >= 100 && c <= 10)`'. With short-circuit evaluation, assessment of the entire condition breaks off early if the evaluation of an earlier subcondition is enough to determine the overall result. For example, if `a` is not equal to `b`, the overall condition will be false no matter if $b \geq 100$ or $c \leq 10$.

*Conditions guarding jumps in the code.* Unstructured jumps in program code, such as the use of `goto`, `break`, `continue` and `return` can also lead to the input condition only being partially evaluated for certain inputs. This can be seen in the code of Figure 4b. When `a` is not equal to `b` in the code, the `return` statement is executed and the function exits, without evaluating the remaining `if` statements. If `a` and `b` are equal, `b >= 100` may now be evaluated, but if it is false, the function again returns.

*Early termination of loops.* A loop appearing in a program, designed to evaluate a multi-dimensional input such as an array, may not full evaluate the input condition if the loop test breaks off before all elements of the input have been assessed. This occurs with the example of Figure 5a, which evaluates an array of integers. If an array element is found to be non-zero, the loop terminates early, without considering elements appearing later in the array. These elements do not have a chance to influence the value of `count`, and as such, the input condition for executing the target branch (where all array elements are zero) is not fully evaluated unless a complete solution has already been found.

Partial evaluation of the input condition prevents the formation of building block constraint schemata for the test data search problem. Take an input condition $c_1 \wedge c_2$. If $c_2$ is not evaluated until $c_1$ is true, then the corresponding constraint schema for $c_2$ cannot be fitness-affecting, and thus cannot be a building block constraint schema. Chromosomes instantiating $c_2$ will never be rewarded unless they first instantiate $c_1$. As such such chromosomes will not proliferate in the population for eventual use in crossover.

This can be seen in all the examples of partial evaluation presented. In the examples of Figures 4a and 4b, if a chromosome is a member of the constraint schema $\{(a, b, c) \mid b \geq 100\}$ it will not be rewarded

```
void all_zeros(int x[SIZE]) {
  int count = 0, all_zeros = 1;

  while (count < SIZE && all_zeros) {
    if (x[count] != 0) all_zeros = 0;
    else count ++;
  }

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

(a) Using counter in target branch

```
void all_zeros(int x[SIZE]) {
  int count = 0, all_zeros = 1;

  while (count < SIZE && all_zeros) {
    if (x[count] != 0) all_zeros = 0;
    else count ++;
  }

  if (all_zeros) {
    // target branch
  }
}
```

(b) Using flag in target branch

Figure 5: A small function evaluating the number of zero-valued array elements. If an one element is found to be non-zero, consideration of the rest of the array breaks off early

unless it is also a member of $\{(a, b) \mid a = b\}$. If it is not a member of $\{(a, b) \mid a = b\}$, the approach level is always 2 and the branch distance is based on the value of `a` and `b`. The same is true for the constraint schema $\{(a, b, c) \mid c \leq 10\}$, which is not rewarded until a chromosome is a member of both $\{(a, b) \mid a = b\}$ and $\{(a, b, c) \mid b \geq 100\}$.

For an input condition be fully evaluated over all inputs to the program, all input variables must influence one atomic condition guarding a target. It therefore follows that 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 internal variable. This is the case with the target branch of Figure 4c, for example, which is covered under exactly the same circumstances as Figure 4a, 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 CAS number validation program of Figure 1 is another example of this. The program keeps count of the number of characters in the string found to be digits, which is used in later branching statements. This enables the search to find digits at each position of the string in parallel, rather than a sequential search beginning at position 1 that moves onto to latter positions one at a time.

**Program Factor 4. Input Condition Conjuncts over Disjoint Sets of Input 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 4c, for example, and the input vectors $<a = 10, b = 10, c = 105>$ and $<a = 50, b = 0, c = 5>$ 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 of the same input variables are referenced in the respective constraints of two or more schemata.

**Definition 6** (Contending Constraint Schemata)**.** *Let vars(S) be the set of input variables involved in the constraint of a constraint schema $S$. 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 4c, for example, involves the contending

```
void direct(int a, int b, int c, int d) {
  int count = 0;

  if (a == 0) count ++;
  if (b == 0) count ++;
  if (c == 0) count ++;
  if (d == 0) count ++;

  if (count == 4) {
    // target branch
  }
}
```

(a) Individual impact

```
void indirect(int a, int b, int c, int d) {
  int count = 0, count1 = 0, count2 = 0;

  if (a == 0) count1 ++;
  if (b == 0) count1 ++;
  if (c == 0) count2 ++;
  if (d == 0) count2 ++;

  if (count1 == 2) count += 2;
  if (count2 == 2) count += 2;

  if (count == 4) {
    // target branch
  }
}
```

(b) Combined impact

Figure 6: Programs demonstrating how different orders of building block constraint schemata can arise for the same input condition

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 $a = b \wedge b \leq 0$.

It follows therefore that the crossover operator will be less useful for programs involving input condition conjuncts that reference overlapping sets of input variables.

**Program Factor 5. Individual Impact of Input Condition Conjuncts on Conditions Guarding the Target**

Input condition conjuncts should be capable of individually influencing the value of the fitness function for the structural target in order for building blocks to form during the search. When the impact of a particular conjunct is 'combined' with other conjuncts with respect to fitness, building block constraint schemata tend to be larger and of higher order. The building block hypothesis predicts that this will inhibit crossover.

The first example of this can be seen in Figure 5. The figure shows two ways of writing the same code to process an array an execute a branch if all elements are zero. In part a of the figure, early zero-valued array elements may directly impact fitness at the condition guarding the target, resulting in a lower branch distance. However, in part b, the predicate guarding the target is replaced with a flag. In this program, *all* array elements must be zero for the fitness value at the target to be affected, and as such the building block schema is the covering constraint schema, of order equal to the size of the inputted array.

The flag problem is a well-known source of poor testability for search-based approaches [12, 6, 4, 3]. However the issue is not limited to flags, as can be seen in Figure 6. Again, this figure contains two programs written in two different ways. Each involves a target executed by the input condition $a = 0 \wedge b = 0 \wedge c = 0 \wedge d = 0$. In part a, any of the input condition conjuncts is capable of changing the value of `count`, which is used in the condition guarding the target enabling a direct impact to be made on the branch distance. Thus each conjunct forms a distinct building block constraint schema, of order 1. In part b, however, both $a = 0$ and $b = 0$ need to be true before `count` is incremented, and an impact made on the branch distance. This not only reduces the number of building blocks (there are now 2 as opposed to 4), and the order of each building block is 2.

The above analysis predicts that the presence and absence of certain factors in a program will allow the crossover operator to work more effectively in the test input search for covering a branch. The empirical study presented in the next section tests each of these predictions.

# 4 Empirical Study

This section describes the results of a series of experiments that were undertaken to empirically demonstrate the effect of each program factor identified in the last section on crossover in Evolutionary Structural Testing. In order to test the effects of each program factor, a series of synthetic programs were designed such that the extent of the presence of each program factor (*e.g.* the number of input condition conjuncts) could be finely controlled and varied, in order to accurately assess its impact on crossover.

The Wegener model of Evolutionary Structural Testing was applied, as described in Section 2 with discrete recombination as a crossover operator. In order to test the impact of crossover, Evolutionary Testing was firstly applied without crossover and secondly with 'headless chicken' crossover, also known as the 'Headless Chicken Test' [19]. The Headless Chicken Test is used to assess whether Genetic Algorithm progress is not merely the result of crossover simply functioning as a macro-mutation operator. With the Headless Chicken Test, crossover is performed as usual, but instead of using two parents drawn from the current population, one of the parents is a new individual generated at random. If Evolutionary Testing cannot perform better than the Headless Chicken Test, the search is not actually benefiting from the random exchange of genes between individuals. For experiments in this paper, headless chicken crossover is discrete recombination, but using a randomly-generated individual and an existing individual as parents for the offspring.

Each particular experiment, involving the coverage of a particular branch in a synthetic function, is repeated 50 times so as to account for stochastic variation. For each batch of 50 runs, two metrics are calculated – the *success rate* and the *average number of fitness evaluations* to find test data. The success rate is the percentage of runs for which test data was successfully found for the target branch, and measures the *effectiveness* of the search. For each successful run, the average number of fitness evaluations can be calculated. This metric indicates the *efficiency* of the search technique for a given branch, since a search finding test data in fewer fitness evaluations will have reached a solution faster with fewer executions of the program under test.

Statistical significance was tested between two experiments with the Wilcoxon rank-sum test, using numbers of fitness evaluations required to find test data over the set of 50 runs. Where test data could not be found in one of the experiments for more than 20 runs (*i.e.* a success rate of 60% or less), Fisher's exact test was applied instead, using the number of successful runs for each experiment. For both types of statistical test, the confidence level was set at 0.999.

## 4.1 Empirical Investigation of Program Factors Predicted to Affect Crossover

### Program Factor 1. Number of Input Condition Conjuncts

In order to assess the effect of program factor 1 on crossover, the function of Figure 7 (Case Study 1) was designed. The program takes an integer array. The code involves a counter (`count`) and a loop. The loop iterates through the array of integer elements and increments `count` every time an array value is found to be equal to be less than the value `R`. The target is executed when all the array elements are less than `R`. The input condition for the target, therefore, is composed of a series of conjuncts where each conjunct corresponds to an in-range array element, *i.e.* $x[0] < R \wedge x[1] < R ... \wedge x[SIZE-1] < R$. The number of input condition conjuncts is varied by changing the value of `SIZE`. Each conjunct forms the constraint of a distinct building block constraint schema for the test data generation problem.

Case Study 1 was run with different array sizes, ranging from 5-50. The domain size of each array element was 0-999, and `R` was set to 500, resulting in a building block probability of 0.5.

For each array size, Evolutionary Testing was performed 50 times, with discrete recombination, headless chicken crossover and no crossover. The results can be found in Table 1. They show that as the length of the array increases, and the number of input condition conjuncts and building block constraint schemata also increase, the search not only becomes harder – as evidenced by higher average numbers of fitness evaluations and lower success rates – but also that crossover becomes increasingly important as a search operator in finding test data. With discrete recombination, test data were always found with a 100% success rate. Evolutionary Testing with discrete recombination significantly outperformed the Headless Chicken Test and no crossover from array sizes of 15 and above. Both headless chicken and no crossover fail to generate test data with 100% success with array sizes greater than 20 and 5 respectively.

The results therefore provide evidence to support the claim that the higher the number of input condition conjuncts, the greater the role crossover plays in finding test inputs in Evolutionary Structural Testing. For higher numbers of input condition conjuncts, inputs could not be found without the crossover operator.

```
void case_study_1(int x[SIZE]) {
  int count = 0, i;

  for (i=0; i < SIZE; i++) {
    if (x[i] < R) count ++;
  }

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

Figure 7: Case Study 1, for assessing the performance of crossover on problems with varying numbers of building block constraint schemata, and varying probabilities of schema constraints being satisfied at random. Each in-range array element forms a building block for finding test data that execute the target branch. The number of building blocks is varied by setting the array length in the code (*i.e.* varying the value of `SIZE`). Building block probability for a particular domain size can be adjusted by changing the value of the global variable `R`

Table 1: Varying inputted array size (number of building block constraint schemata) with the Case Study 1 (Figure 7). The number of average evaluations is recorded unless the target could not be covered with a 100% success rate, in which case success rate is recorded instead. A figure appears in bold if the crossover type was significantly worse than discrete recombination, or in italics if the crossover type was significantly better. The results show that crossover plays an increasingly important role as the number of building block constraint schemata increase

| **Array size / no. of building blocks** | **Crossover type** Discrete | Headless Chicken | None |
|---|---|---|---|
| 5 | 32 | 32 | 32 |
| 10 | 497 | 582 | 68% |
| 15 | 1,071 | **3,178** | **18%** |
| 20 | 1,615 | **14,380** | **4%** |
| 25 | 2,076 | **90%** | **0%** |
| 30 | 2,518 | **12%** | **0%** |
| 35 | 2,941 | **0%** | **0%** |
| 40 | 3,273 | **0%** | **0%** |
| 45 | 3,821 | **0%** | **0%** |
| 50 | 4,054 | **0%** | **0%** |
| 55 | 4,391 | **0%** | **0%** |
| 60 | 4,950 | **0%** | **0%** |
| 65 | 5,095 | **0%** | **0%** |
| 70 | 5,430 | **0%** | **0%** |
| 75 | 5,726 | **0%** | **0%** |
| 80 | 6,043 | **0%** | **0%** |
| 85 | 6,321 | **0%** | **0%** |
| 90 | 6,572 | **0%** | **0%** |
| 95 | 6,828 | **0%** | **0%** |
| 100 | 7,186 | **0%** | **0%** |

```
void case_study_2(int x[SIZE]) {
  int count = 0, i;

  for (i=0; i < SIZE; i++) {
    if (x[i] < R)
      count ++;
    else
      count += 1 - ((x[i]-R)/(DOMAIN-R));
  }

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

Figure 8: Case Study 2, used to assess the impact of landscape on crossover. The target branch is executed under exactly the same condition as Case Study 1 (Figure 7), except the `count` variable is incremented by fractional amounts proportional to the distance the array element was to being less than `R`. This results in a gradient landscape, as seen in Figure 9b

**Program Factor 2. Search Difficulty Involved in Satisfying Input Condition Conjuncts**

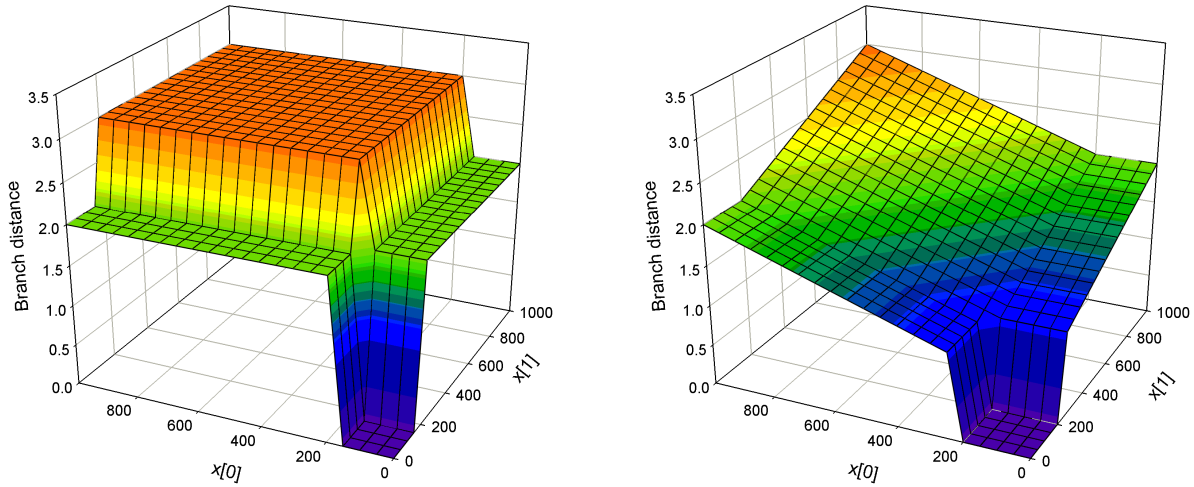
Input condition conjuncts are harder to satisfy the fewer the input domain elements that will make them true. This occurs when the probability of finding inputs at random decreases. The shape of the fitness landscape is also important; when the landscape is flat, the search relies on mutation finding inputs by chance. A gradient landscape, on the other hand, provides guidance to the required values. The last section predicted that the harder input condition conjuncts are to satisfy, the more essential crossover will be as a search operator, as it can recombine existing genetic material in the population at each loci, rather than having to rediscover those values for every individual through mutation.

Experiments were performed with Case Study 1 (Figure 7) with an array size of 50. Different values of `R` were applied with array element domains of 0-999 to give different building block probabilities; *i.e.* a value of `R` = 500 (giving a probability of 0.5), 250 (0.25), 200 (0.2), 100 (0.1), 50 (0.05), and 10 (0.01). The results can be found in Table 2. As the probability of producing an array value less than `R` becomes smaller, the search problem becomes more difficult, and crossover becomes more essential, as predicted by the theory. Searches with discrete recombination were 100% successful down to a probability level of 0.2. Headless chicken crossover generated test data at the 0.5 level with only 90% success, while searches with no crossover were always unsuccessful.

The same experiments were re-performed with Case Study 2 (Figure 8), which is identical to Case Study 1, except it has a much easier gradient landscape for the target – as can be observed by comparing a visualisation of the landscape plots in Figure 9 for the case studies. As expected, mutation becomes more competent at finding in-range array elements, and the search without crossover improves dramatically. Evolutionary Testing with discrete recombination is more effective, with 100% success rates being achieved at all probability levels, and with a lower average of fitness evaluations. Evolutionary Testing without crossover is more successful with the gradient landscape, as mutation is able to find the required values alone. However, as the figures show, even with the gradient landscape, Evolutionary Testing with crossover is always significantly better.

The gradient landscape results in discrete recombination becoming more effective, with 100% success rates being achieved at all probability levels, but interestingly is less efficient than the flat landscape at probabilities of 0.5 to 0.1; requiring a greater number of fitness evaluations on average. This is because of what happens during the selection phase of the Genetic Algorithm. The flat landscape results in a smaller number of 'super' fit individuals, with relatively high number of array elements in range, which are re-selected for breeding. When a gradient landscape is used, the 'best' solutions may not necessarily have a high number of array elements in-range, but a high number of array elements that merely *close* to being in-range.

These results therefore support the claim that the harder the search difficulty involved in satisfying




(a) Case Study 1: flat landscape

(b) Case Study 2: gradient landscape

Figure 9: Fitness function landscapes for case studies 1 and 2, with an array size of 2. The form of the landscape generated by Case Study 1 is flat, compared to that of Case Study 2, which offers a smooth downward gradient to the input values that will execute the target. The test data search is more successful as a result

```
void case_study_3(int x[SIZE]) {
  if (x[0] < R) {
    if (x[1] < R) {
      if (x[2] < R) {
        // ...
        // target branch
```

Figure 10: Case Study 2, used to assess crossover when input conditions may only be partially evaluated. The target branch is executed under exactly the same condition as that of Case Study 1 Figure 7, except the input condition conjuncts are nested

input condition conjuncts positively affects the role that crossover will play in the discovery of test inputs.

**Program Factor 3. Evaluation of the Input Condition with Respect to the Target Structure**

Experiments were performed with Case Study 3 (Figure 10) under the same conditions as those for RQ1, *i.e.* the domain of each array element set to 0-999, `R` = 500, and array sizes ranging from 5 to 50. The target branch of Case Study 3 (Figure 10) is executed under exactly the same circumstances as Case Study 1, but instead of using a `count` variable, conjuncts of the input condition are nested. This means that each conjunct is only evaluated if the previous conjuncts were true, *i.e.* the input condition may only be partially evaluated with respect to the target.

With Case Study 3, each array element introduces an input condition conjunct for executing the target; however building block constraint schemata are not created due to the nested structure of the program. The results can be seen in Table 3, and show that discrete recombination helps in finding test data, significantly outperforming the Headless Chicken Test and no crossover in all but three cases.

As predicted, however, searches are less effective for nested programs than non-nested programs, as seen in Table 4. With Case Study 1, where input condition conjuncts are not nested, Evolutionary Testing with discrete recombination always covers the target, regardless of array size. However, with

Table 2: Varying building block probability with the Case Study 1 (Figure 7) and Case Study 2 (Figure 8). The number of average evaluations is recorded unless the target could not be covered with a 100% success rate, in which case success rate is recorded instead. A figure appears in bold if the crossover type was significantly worse than discrete recombination, or in italics if the crossover type was significantly better. The results show that crossover has an increasing role to play in test data discovery as the probability of finding inputs at random reduces, for both flat and gradient landscapes

(a) Flat landscape

(b) Gradient landscape

| **Building block probability** | **Crossover type** Discrete | Headless Chicken | None | **Crossover type** Discrete | Headless Chicken | None |
|---|---|---|---|---|---|---|
| 0.50 | 2,076 | **90%** | **0%** | 2,432 | **58%** | **27,114** |
| 0.25 | 4,093 | **0%** | **0%** | 5,905 | **0%** | **58,740** |
| 0.20 | 4,683 | **0%** | **0%** | 7,149 | **0%** | **67,681** |
| 0.10 | 6,248 | **0%** | **0%** | 11,346 | **0%** | **96%** |
| 0.05 | 94% | **0%** | **0%** | 16,037 | **0%** | **74%** |
| 0.01 | 12% | 0% | 0% | 23,160 | **0%** | **30%** |

Table 3: Results with Case Study 3 (Figure 10) with nested conditions. The number of average evaluations is recorded unless the target could not be covered with a 100% success rate, in which case success rate is recorded instead. A figure appears in bold if the crossover type was significantly worse than discrete recombination, or in italics if the crossover type was significantly better. The figures show that discrete recombination clearly plays a role in finding test data

| **Array size / level of nesting** | **Crossover Type** Discrete | Headless Chicken | None |
|---|---|---|---|
| 5 | 32 | 32 | 32 |
| 10 | 587 | 638 | **8,715** |
| 15 | 1,790 | **4,697** | **28,649** |
| 20 | 3,411 | **27,535** | **58,965** |
| 25 | 5,987 | **30%** | **76%** |
| 30 | 8,372 | **0%** | **40%** |
| 35 | 14,527 | **0%** | **10%** |
| 40 | 17,491 | **0%** | **0%** |
| 45 | 96% | **0%** | **0%** |
| 50 | 82% | **0%** | **0%** |
| 55 | 66% | 0% | 0% |
| 60 | 38% | 0% | 0% |
| 65 | 22% | 0% | 0% |
| 70 | 2% | 0% | 0% |
| 75 | 6% | 0% | 0% |
| 80 | 0% | 0% | 0% |
| 85 | 0% | 0% | 0% |
| 90 | 0% | 0% | 0% |
| 95 | 0% | 0% | 0% |
| 100 | 0% | 0% | 0% |

Table 4: Comparing results with Case Studies 1 and 3 (Figures 7 and 10 respectively). The number of average evaluations is recorded unless the target could not be covered with a 100% success rate, in which case success rate is recorded instead. The target is executed in the same condition for both programs, but the structure is different – in Case Study 1 input condition conjuncts are not nested whereas with Case Study 3 they are. A figure appears in bold for Case Study 3 if the search performed significantly worse when compared with Case Study 1, or in italics if the reverse was true. The results show that Evolutionary Testing is clearly is more effective and efficient for the non-nested Case Study 1

| **Array size** | **Case Study 1** (Non-nested) | **Case Study 3** (Nested) |
|---|---|---|
| 5 | 32 | 32 |
| 10 | 497 | 587 |
| 15 | 1,071 | **1,790** |
| 20 | 1,615 | **3,411** |
| 25 | 2,076 | **5,987** |
| 30 | 2,518 | **8,372** |
| 35 | 2,941 | **14,527** |
| 40 | 3,273 | **17,491** |
| 45 | 3,821 | **96%** |
| 50 | 4,054 | **82%** |
| 55 | 4,391 | **66%** |
| 60 | 4,950 | **38%** |
| 65 | 5,095 | **22%** |
| 70 | 5,430 | **2%** |
| 75 | 5,726 | **6%** |
| 80 | 6,043 | **0%** |
| 85 | 6,321 | **0%** |
| 90 | 6,572 | **0%** |
| 95 | 6,828 | **0%** |
| 100 | 7,186 | **0%** |

the nested conditionals in Case Study 3, Evolutionary Testing with discrete recombination is only 100% successful up to an array size of 35. The search is significantly better with the non-nested Case Study 1 for an array size of 15 elements or more. Figure 11 plots the average number of fitness evaluations required to find a target-executing input vector.

The last section showed how nesting prevents input condition conjuncts forming building block constraint schemata, and this can be seen visually with Figure 12. The greyscale heat maps in this figure correspond to two searches with the non-nested Case Study 1 (Figure 12a) and its nested equivalent, Case Study 3 (Figure 12b). In these heat maps, building block probability is lower for Case Study 1 (0.1) to produce a search of similar duration for easy comparison with that with the nested Case Study 2. Each cell at each grid position of the map corresponds to a particular array element at a given index (a loci in the chromosome) at each generation of the search. The grid position is darker the number of individuals in that generation with that array index element in-range. The cell is black if all 300 individuals have an array index element in-range at a given generation. For the non-nested search, all in-range array index elements are building blocks of the problem, and as such individuals are rewarded if any of their array elements are in-range at any point in the search. Thus the heat map becomes darker moving from left to the right of the figure, *i.e.* as the search progresses. This contrasts with the nested search. Here, an individual only receives fitness reward for an in-range array index element if all the elements before it in the array are also in-range, giving a staircase appearance to the figure. Despite this, crossover can still positively impact the search for nested programs. The heat map shows how crossover allows 'good' genes, or in-range array elements, to be spread across the population over time, providing more opportunities for mutation to penetrate the next level of nesting in the next generation.

The results from the experiments for partial evaluation of the input condition therefore do support the claim that crossover is affected when the full input condition does not undergo full evaluation for a branch target. (Section 5 discusses ways in which this problem may be alleviated.)

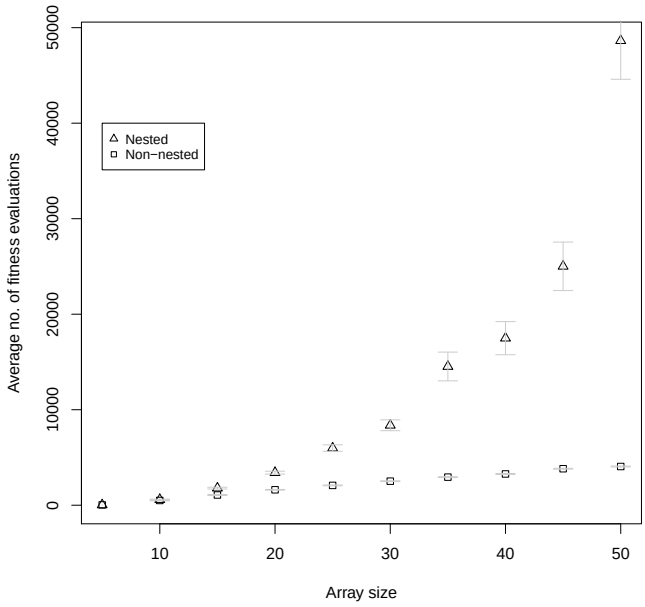



Figure 11: Comparing average fitness evaluations for discrete recombination when constraints are nested (Case Study 2) and when they are not (Case Study 1). Average fitness evaluations increases exponentially for nested input condition conjuncts whereas the trend for non-nested conjuncts is more linear

**Program Factor 4. Input Condition Conjuncts over Disjoint Sets of Input Variables**

In order to answer RQ4, Case Study 4 (Figure 13) was designed. The function involves input condition conjuncts with non-disjoint sets of variables, *i.e.* the involvement of contending constraint schemata. The number of contending building block constraint schemata are controlled through the value of `NUM_CONTENDING`. The target is executed when the array elements at indexes 1 to `NUM_CONTENDING` are equal to the first array element at index zero. The dependence of subsequent array elements on the first element's value creates contending constraint schema of the form $\{(x[0], x[1], x[2] \ldots) \mid x[0] = x[1]\}$, $\{(x[0], x[1], x[2] \ldots) \mid x[0] = x[2]\}$ *etc.*. The remainder of the array from indexes `NUM_CONTENDING`+1 to `SIZE`-1 must be equal to zero.

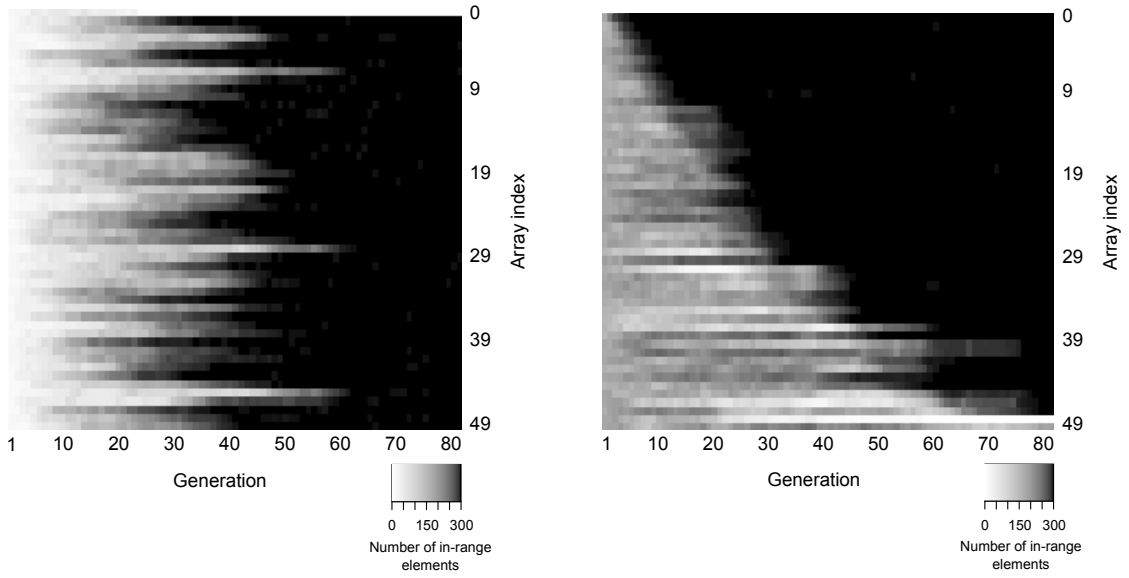
Experiments were performed using an array size of 50, with the number of contending building block schemata increased by varying the value of `NUM_CONTENDING` from 0 to 50. Three different building block probabilities were tried, 0.5, 0.25 and 0.1. This was achieved by widening the domain of each array element from 0 to 1, up to -2 to 3 and -4 to 5.

The results can be seen in Table 5. Across all three probability levels tested, the number of contending building block constraint schemata increase, average evaluations increases or the success rate decreases. One reason for this is the crossover operator failing to effectively produce offspring in presence of constraints across array elements. The results support the prediction that contending schemata negatively impact the performance of the crossover operator.

**Program Factor 5. Individual Impact of Input Condition Conjuncts on Conditions Guarding the Target**

Case Study 5 (Figure 14) was designed to empirically test the question of the effect on crossover of the number of input variables involved in the constraint of each building block constraint schema. The value of `NUM_VARS` allows the number of input variables involved in the constraint of each building block constraint schema to be varied, since it controls how many array elements should be less than `R` before `count` is incremented.

In order to keep the building block probability constant, the probability of each array element having




(a) Non-nested

(b) Nested

Figure 12: Greyscale heat maps showing the progress of two example evolutionary searches of comparable duration. The first is with Case Study 1 (Figure 7), with non-nested input condition conjuncts, and the second for Case Study 3 (Figure 10) with nested conjuncts. The darkness of each cell in the map denotes the number of chromosomes in a generation for which a particular array element was in-range (*i.e.* a particular constraint schema was present). Without nesting, all constraint schema are potentially rewarded, so their multiplicity increases evenly over time; aided by crossover. Hence for all array indices, the number of in-range elements increases over each generation (heat map becomes darker moving left to right). With nesting, however, constraint schema are only rewarded one at a time; *i.e.* if previous constraint schema in the nesting structure have been satisfied. Thus the heat map takes on a 'stepped' looking appearance, since an in-range element will only increase in multiplicity if there are a suitable number of in-range elements earlier in the array rewarded by the nested structure of the program

```
void case_study_4(int x[SIZE]) {
  int count = 1, i, j;

  for (i=1; i <= NUM_CONTENDING; i++) {
    if (x[i] == x[0]) count ++;
  }

  for (j=NUM_CONTENDING+1; j < SIZE; j++) {
    if (x[j] == 0) count ++;
  }

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

Figure 13: Case Study 4, used to assess the performance of the crossover operator in the presence of *contending* constraint schemata – schemata whose constraints reference disjoint sets of input variables. The target is executed when the array elements at indexes 1 to `NUM_CONTENDING` are equal to the first array element at index 0. The remaining elements at indexes `NUM_CONTENDING` to the end of the array need to be equal to zero, independently of the other array elements

Table 5: Results for discrete recombination with Case Study 4 (Figure 13). The number of average evaluations is recorded unless the target could not be covered with a 100% success rate, in which case success rate is recorded instead. As the number of building block contending constraint schemata increase, the search becomes more difficult, as evidenced by a decreasing success rate or an increasing number of average fitness evaluations required to reach a solution

| **No. of Contending Constraint Schemata** | **Probability** 0.5 | 0.25 | 0.1 |
|---|---|---|---|
| 0 | 1,595 | 2,712 | 5,034 |
| 1 | 1,663 | 2,802 | 5,146 |
| 2 | 1,598 | 2,892 | 5,547 |
| 3 | 1,821 | 3,001 | 5,960 |
| 4 | 1,830 | 3,067 | 6,095 |
| 5 | 1,896 | 3,127 | 94% |
| 6 | 1,898 | 3,110 | 94% |
| 7 | 1,905 | 3,178 | 84% |
| 8 | 1,937 | 3,240 | 94% |
| 9 | 1,970 | 3,193 | 72% |
| 10 | 1,994 | 3,228 | 82% |
| 11 | 2,000 | 3,314 | 74% |
| 12 | 2,097 | 3,250 | 66% |
| 13 | 2,156 | 3,350 | 66% |
| 14 | 2,148 | 3,347 | 52% |
| 15 | 2,123 | 3,365 | 60% |
| 16 | 2,132 | 3,279 | 56% |
| 17 | 2,130 | 3,415 | 58% |
| 18 | 2,134 | 3,416 | 42% |
| 19 | 2,088 | 3,403 | 50% |
| 20 | 2,014 | 3,347 | 40% |

```
void case_study_5(int x[SIZE]) {
  int count = 0, i, j;

  for (i=0; i < SIZE; i += NUM_VARS) {
    int count2 = 0;

    for (j=i; j < i + NUM_VARS; j++) {
      if (x[j] < R) count2 ++;
    }

    if (count2 == NUM_VARS)
      count += NUM_VARS;
  }

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

Figure 14: Case Study 5, used to assess the performance of crossover when the order of building block constraint schemata (the number of variables involved in the constraint of a constraint schema) is varied

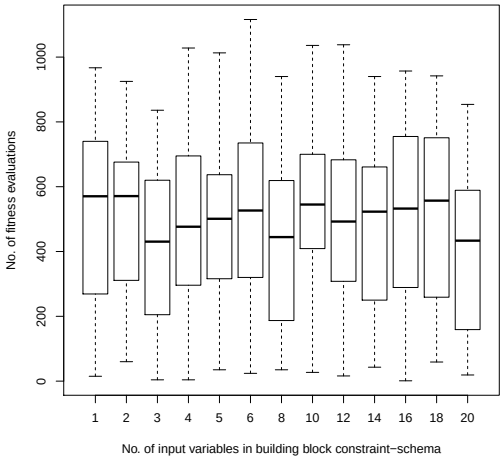
the required value increases in proportion with the number of array elements involved in each block. This is controlled by computing the element probability, $e_p$, and scaling it the domain of each array element. The element probability is computed using the formula $e_p = p_b^{\frac{1}{nv}}$, where $p_b$ is the building block probability, $nv$ is the number of variables involved in the building block's constraint. The value of `R` is then $round(min\_domain + ((max\_domain - min\_domain + 1) * e_p) - 1)$; where $min\_domain$ is the lowest value that each of the array elements can have, $max\_domain$ is the largest.

For example, suppose there is one input variable per building block, and building block probability is 0.5. If the domain size of each array element is 0-999, `R` is 499. However if there are two variables per building block, $e_p = 0.5^{\frac{1}{2}}$ and `R` = 706.

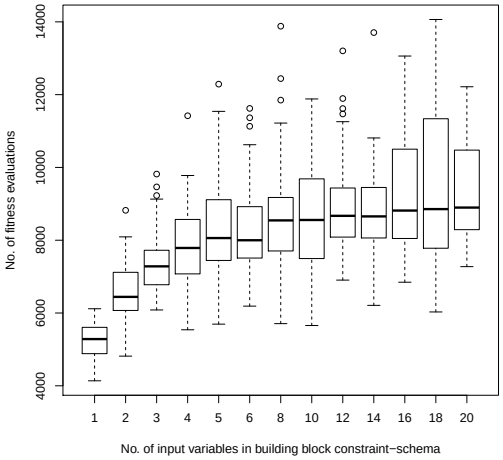
Experiments were performed by varying the number of array elements in each building block constraint schema by increasing the value of `NUM_VARS` from 1 to 20. Three different configurations of the experiment were attempted. In the first and second, the probability of each building block constraint schema forming at random was fixed at 0.5. This was achieved by fixing `MIN_RANGE` and `MAX_RANGE` according to the number of array elements in each building block, so that the overall probability of all the array elements involved in the building block was fixed even though the number of array elements involved in it was changing. In the third experimental configuration, the probability was lowered to 0.1. Furthermore, the number of building blocks was varied. The first and third configurations involved 10 building block constraint schemata each, resulting in array sizes of 10 up to 200. The second configuration involved 20 building block constraint schemata, resulting in array sizes of 20 up to 400.

Each experimental configuration with each value of `NUM_VARS` was repeated 50 times using Evolutionary Testing with discrete recombination. Box plots for the distribution of fitness evaluations over each of the 50 runs can be found in Figure 15. With 10 building blocks and a probability of 0.5 (Figure 15a), no significant variation was observed. However with a larger number of building blocks or a lower probability, the number of variables in each building block constraint schema matters. In both cases the jump from one variable to two variables is significant. There is then a trend of a higher average number of fitness evaluations required to reach a solution up to a variable number of 5. The crossover operator finds it easier to operate when there is only one variable involved in each building block constraint schema, but above this level it becomes less probable that the variables involved in each block will be crossed over into the same offspring.

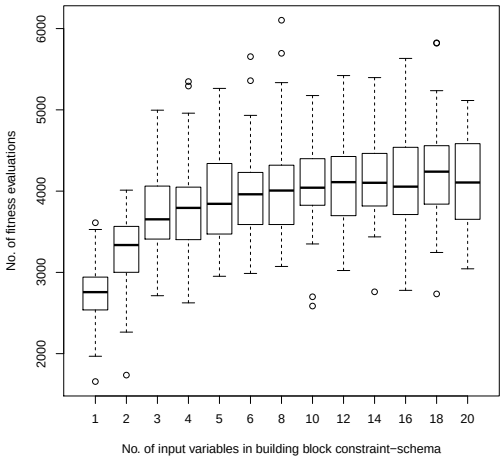
The above findings therefore support the claim that increasing the number of variables in each building block constraint schema will have a negative impact on the performance of crossover in the search.




(a) No. of building blocks = 10, building block probability of 0.5




(b) No. of building blocks = 20, building block probability of 0.5




(c) No. of building blocks = 10, building block probability of 0.1

Figure 15: The effect of the number of variables involved in each building block constraint schema for Case Study 5 (Figure 14). With larger array sizes and lower probabilities there is a pronounced difference between just one variable and more than one variable

## 4.2 Further experiments

Further experiments were conducted to determine the effect of different crossover types, namely one-point crossover and uniform crossover, and also comparing the performance of Evolutionary Search with alternative Hill Climbing approaches.

### 4.2.1 Crossover type

Experiments were performed with different forms of crossover with Case Study 1 (Figure 7), the results of which can be found in Table 6 and Table 7. Discrete recombination almost always significantly outperformed one-point crossover. However, on several occasions uniform crossover was significantly better than discrete recombination, particularly with large numbers of building blocks and low building block formation probabilities.

A similar pattern emerges for Case Study 4 (Figure 13), where the number of contending building block constraint schema are varied. The results can be found in Table 8. One-point crossover was almost always significantly outperformed by discrete recombination, which was in turn almost always significantly outperformed by uniform crossover.

There was no significant difference between discrete recombination and uniform crossover when the number of input variables involved in each building block constraint schema was varied, with Case Study 5 (Figure 14), as can be seen in Table 9. However one-point crossover was significantly worse for small numbers of input variables, and its performance did not significantly change when the number of input variables increased.

The poor performance of one-point crossover can be attributed to its inflexibility, particularly with building block constraint schema involving few input variables. A freer exchange of input variables across chromosomes allows the search to progress faster.

Over all of the case studies, uniform crossover is significantly better than discrete recombination on several occasions. Conversely, discrete recombination never outperforms uniform crossover. Discrete recombination does not guarantee to copy all parental genes into their offspring, and in this way, seems to be responsible for destroying valuable building blocks, slowing down the progress of the search. This is not the case with uniform crossover, which always preserves genes in newly-created offspring.

### 4.2.2 Performance of Evolutionary Testing compared to Variants of Hill Climbing

Experiments with Case Study 1, 2 and 4 were performed with the Alternating Variable Method (AVM), Random Hill Climbing (RHC) and the Random Hill Climbing-Alternating Variable Method (RHC-AVM). The *Alternating Variable Method* (AVM) [20] was one of the earliest algorithms used for Search-Based Testing. 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 $i^{th}$ 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 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.

With *Random Hill Climbing* (RHC), an input vector is initially selected from the search space at random. Mutations are then made by replacing input variable values with a new value selected at uniform random. Input variables are mutated at a probability that is the inverse of the input vector's length. The mutated individual replaces the current individual if it is of improved fitness. RHC is the equivalent of a (1+1) EA – an Evolutionary Algorithm with only one individual and hence no crossover.

The *Random Hill Climbing-Alternating Variable Method* (RHC-AVM) combines the Alternating Variable Method with random mutation restarts (referred to as RM-AVM in [24]). When the Alternating Variable Method becomes 'stuck', random mutations are made until a better solution is found. RHC-AVM therefore incorporates the best features of Alternating Variable Method and Random Hill Climbing; the ability of the Alternating Variable Method to accelerate down gradients with the ability of RHC to escape certain local optima.

Table 6: Comparing crossover operators for Case Study 1 (Figure 7), varying array size but keeping constraint schema probability constant at 0.5. The number of average evaluations is recorded unless the target could not be covered with a 100% success rate, in which case success rate is recorded instead. A figure appears in bold if the crossover type was significantly worse than discrete recombination, or in italics if the reverse was true. The results show that uniform crossover is often the most effective operator, often significantly outperforming discrete recombination (which in turn often significantly outperforms one-point crossover)

| **Array size / no. of building blocks** | **Crossover type** | | |
|---|---|---|---|
| | Discrete | Uniform | One-point |
| 5 | 32 | 32 | 32 |
| 10 | 497 | 462 | 493 |
| 15 | 1,071 | 1,084 | 1,258 |
| 20 | 1,615 | 1,638 | **2,063** |
| 25 | 2,076 | 2,098 | **2,834** |
| 30 | 2,518 | 2,502 | **3,701** |
| 35 | 2,941 | 2,916 | **4,324** |
| 40 | 3,273 | 3,238 | **5,223** |
| 45 | 3,821 | 3,656 | **5,875** |
| 50 | 4,054 | 3,946 | **6,688** |
| 55 | 4,391 | 4,314 | **7,309** |
| 60 | 4,950 | *4,643* | **8,177** |
| 65 | 5,095 | 4,942 | **9,107** |
| 70 | 5,430 | 5,230 | **9,882** |
| 75 | 5,726 | *5,542* | **10,695** |
| 80 | 6,043 | *5,743* | **90%** |
| 85 | 6,321 | *6,003* | **11,879** |
| 90 | 6,572 | 6,371 | **92%** |
| 95 | 6,828 | *6,512* | **90%** |
| 100 | 7,186 | *6,811* | **84%** |

Table 7: Varying building block probability with the Case Study 1 (Figure 7) and Case Study 2 (Figure 8). The number of average evaluations is recorded unless the target could not be covered with a 100% success rate, in which case success rate is recorded instead. A figure appears in bold if the crossover type was significantly worse than discrete recombination, or in italics if the reverse was true. The results show that crossover has an increasing role to play in test data discovery as the probability of finding inputs at random reduces, for both flat and gradient landscapes

(a) Flat landscape

(b) Gradient landscape

| **Building block probability** | **Crossover type** | | | **Crossover type** | | |
|---|---|---|---|---|---|---|
| | Discrete | Uniform | One-point | Discrete | Uniform | One-point |
| 0.50 | 2,076 | 2,098 | **2,834** | 2,432 | 2,496 | **3,602** |
| 0.25 | 4,093 | 3,895 | **6,235** | 5,905 | *5,389* | **9,283** |
| 0.20 | 4,683 | *4,357* | **7,282** | 7,149 | *6,335* | **11,401** |
| 0.10 | 6,248 | *5,781* | **96%** | 11,346 | *10,132* | **18,756** |
| 0.05 | 94% | *7,136* | **62%** | 16,037 | *14,405* | **23,270** |
| 0.01 | 12% | 28% | 0% | 23,160 | *21,289* | **30,021** |

Table 8: Comparing crossover types with Case Study 4 (Figure 13), at a constraint schema probability level of 0.1. The number of average evaluations is recorded unless the target could not be covered with a 100% success rate, in which case success rate is recorded instead. A figure appears in bold if the crossover type was significantly worse than discrete recombination, or in italics if the reverse was true. Again, uniform crossover often significantly outperforms discrete recombination

| **No. of Contending Constraint Schemata** | **Crossover type** | | |
|---|---|---|---|
| | Discrete | Uniform | One-point |
| 0 | 5,034 | *4,615* | **8,094** |
| 1 | 5,146 | *4,792* | **7,757** |
| 2 | 5,547 | 5,179 | **8,058** |
| 3 | 5,960 | *5,514* | **94%** |
| 4 | 6,095 | *5,453* | **82%** |
| 5 | 94% | *98%* | **54%** |
| 6 | 94% | *98%* | **60%** |
| 7 | 84% | 98% | 50% |
| 8 | 94% | 94% | **52%** |
| 9 | 72% | 92% | 48% |
| 10 | 82% | 92% | **30%** |
| 11 | 74% | *82%* | 34% |
| 12 | 66% | 90% | 26% |
| 13 | 66% | 74% | 24% |
| 14 | 52% | 78% | 20% |
| 15 | 60% | 72% | 14% |
| 16 | 56% | 64% | 20% |
| 17 | 58% | 66% | 8% |
| 18 | 42% | 62% | 8% |
| 19 | 50% | 60% | 18% |
| 20 | 40% | 64% | 2% |

Table 9: Crossover types and the number of input variables involved in each building block. A figure appears in bold if the crossover type was significantly worse than discrete recombination, or in italics if the reverse was true. Again, uniform crossover often significantly outperforms discrete recombination

| **No. of input variables per building block** | **Crossover type** | | |
|---|---|---|---|
| | Discrete | Uniform | One-point |
| 1 | 2,749 | 2,621 | **3,783** |
| 2 | 3,249 | 3,260 | **98%** |
| 3 | 3,691 | *3,388* | 4,038 |
| 4 | 3,815 | 3,670 | 4,142 |
| 5 | 3,940 | 3,832 | 4,132 |
| 6 | 3,949 | 3,693 | 4,018 |
| 8 | 4,052 | 3,950 | 3,919 |
| 10 | 4,071 | 3,999 | 4,181 |
| 12 | 4,127 | 3,950 | 4,189 |
| 14 | 4,161 | 3,948 | 4,048 |
| 16 | 4,164 | 3,920 | 4,046 |
| 18 | 4,250 | 4,024 | 4,055 |
| 20 | 4,120 | 4,208 | 4,060 |

Tables 10-12 report the results of these hill climbing approaches compared with the best configuration for Evolutionary Testing. Experiments in the previous section found uniform crossover to perform best. Further experimentation found that a population size of 120 represented the optimal size of population. Despite these best settings for Evolutionary Testing, it was always out performed by one of the hill climbers for each of the case studies.

As expected, the Alternating Variable Method performs poorly with the flat fitness landscape generated with Case Study 1 (Table 10), but is significantly superior to all other searches when a gradient exists, as with Case Study 2 (Table 11). The Alternating Variable Method also struggles with Case Study 4, as seen in Table 12, as it is not able to change more than one input variable value at once. RHC, on the hand, performs best with case studies 1 and 4, and is significantly better when compared to Evolutionary Testing. It is significantly worse than Evolutionary Testing for the gradient landscape of Case Study 2. The RHC-AVM, being a combination of RHC and the Alternating Variable Method, always performs somewhere between the two. In summary for case studies 1, 2 and 4, it is always the case that at least one of the hill climbers significantly outperforms Evolutionary Testing with crossover.

Case Study 6 was specifically designed to contain a local optimum (Figure 16), to demonstrate that cases of program can exist where Evolutionary Testing will consistently outperform both the Alternating Variable Method and RHC. Each in-range element increments the `count` variable, improving the branch distance at the target. However, when the number of elements that are less than `R` rises to a certain level (defined by the length of the array minus the value of the `GAP_SIZE` variable), `count` is reset to zero. It only increases again when all elements of the array are less than `R`. This causes a local optimum to form in the fitness landscape, as depicted in part b of the figure. Evolutionary Testing may overcome the local optimum by crossover of two individuals with in-range elements at different positions, as shown in part c of the figure. The first individual has in-range elements at the beginning of the chromosome, whilst the second has in-range elements at the end. A one-point crossover in the middle of the chromosome will result in a child having all the array elements in range, and the target will be executed. The results with the Case Study are shown in Table 13 with domain size as for Case Study 1 and a building block generation probability of 0.5. As illustrated in Figure 16c, it is possible for Evolutionary Testing 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 RHC.

The conclusion for this research question, therefore, is that test data generation problems for 'crossover-friendly' programs are not necessarily better solved by Evolutionary Testing than a hill climber. Cast study 5, however, does show that test data generation problems can exist where Evolutionary Testing can find test data, but which are highly challenging for hill climbers. However, how many real examples of this type exist in practice is an open question.

## 4.3 Threats to Validity

The results of the empirical study provide evidence that the performance of the crossover operator is affected for the set of program factors outlined in Section 3, using the set of case studies designed to evaluate them.

The purpose of the empirical study was to demonstrate the causal relationships between different program factors and the performance of the crossover operator. This section discusses potential threats to validity for the experiments performed. The first consideration is that of the internal validity of the experiments; that is, whether there has been a bias in the experimental design that could affect the results of the study and the conclusions that have been drawn. One such source of potential bias comes from the inherent stochastic behaviour of the search algorithms under scrutiny. A common and reliable technique for overcoming this source of variability is to perform tests for statistical significance on a sufficiently large sample of result data. Such a test is required whenever one wishes to make the claim that one technique produces different results to another. A set of results are obtained from a set of runs of the search algorithm using different random seeds. To show that one crossover type or search algorithm is more *effective* than another, Fisher's Exact Test – a test for categorical data – was used to compare the number of successful searches that covered a particular branch. To show that one crossover type or search algorithm is more *efficient* than another, the Wilcoxon rank-sum test was used to compare numbers of fitness evaluations required by each search algorithm in order to find inputs that cover a particular program branch. The lower the number of fitness evaluations, the quicker and more easily the search was to find an input. Both tests were applied with the confidence level set at 99.9% to see if there is a statistical significant difference in the means of each set of results. Both tests are non-parametric,

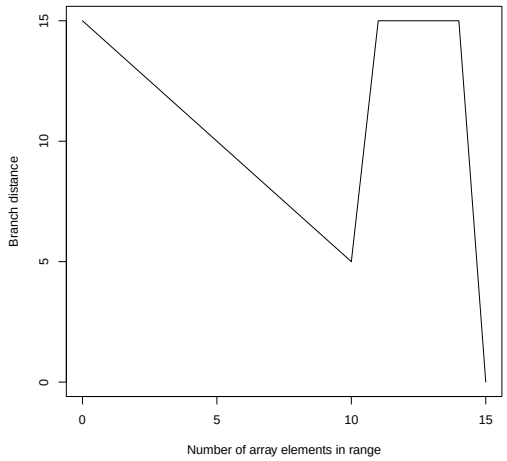
```
void case_study_6(int x[SIZE]) {
  int count = 0, i;

  for (i=0; i < SIZE; i++) {
    if (x[i] < R) count ++;
  }

  if (count > SIZE-GAP_SIZE && count < SIZE) {
    count = 0;
  }

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

(a) Code




(b) Local optimum in fitness landscape

```
00000000|0000XXX        000000000000000
XXXX0000|0000000        XXXX00000000XXX
```

(c) Crossover of two chromosomes to reach the global optimum

Figure 16: Case Study 6. The code, shown in part a, is similar to that of Case Study 1. The `count` variable is increased until a certain value, upon which it is reset to zero. It does not increase again until all array elements are less than `R`. This causes a local optimum to form in the fitness landscape as shown in part b of the figure. It is possible for Evolutionary Testing to overcome the local optimum through crossing over two individuals with in-range elements at different positions, for example as shown in part c of the figure with one-point crossover, where an '`0`' represents a value less than `R` while an '`X`' represents a value that is out of range

Table 10: Comparing Hill Climbing with Evolutionary Testing on Case Study 1 (Figure 7) by varying array size (number of building blocks for Evolutionary Testing). The optimal setup for Evolutionary Testing (Uniform 120 – uniform crossover, 120 individuals in a population) is used, as found in the answer to RQ7. The number of average evaluations is recorded unless the target could not be covered with a 100% success rate, in which case success rate is recorded instead. A figure appears in bold if the search was significantly worse than Evolutionary Testing with discrete recombination, or in italics if the reverse was true. RHC is the best performer, and is significantly better than Evolutionary Testing in the majority of cases

| **Array size / no. of building blocks** | **Search type** Uniform 120 | AVM | RHC | RHC-AVM |
|---|---|---|---|---|
| 5 | 32 | **857** | 22 | **174** |
| 10 | 287 | **88%** | *67* | **970** |
| 15 | 550 | **6%** | *123* | **2,622** |
| 20 | 779 | **0%** | *180* | **4,997** |
| 25 | 1,030 | **0%** | *196* | **6,809** |
| 30 | 1,256 | **0%** | *306* | **12,494** |
| 35 | 1,414 | **0%** | *353* | **16,772** |
| 40 | 1,631 | **0%** | *429* | **23,274** |
| 45 | 1,851 | **0%** | *434* | **25,808** |
| 50 | 2,105 | **0%** | *532* | **35,878** |
| 55 | 2,183 | **0%** | *681* | **98%** |
| 60 | 2,332 | **0%** | *722* | **96%** |
| 65 | 2,521 | **0%** | *803* | **94%** |
| 70 | 2,704 | **0%** | *965* | **72%** |
| 75 | 2,933 | **0%** | *1,018* | **48%** |
| 80 | 3,186 | **0%** | *1,076* | **38%** |
| 85 | 3,149 | **0%** | *1,059* | **28%** |
| 90 | 3,521 | **0%** | *1,215* | **14%** |
| 95 | 3,596 | **0%** | *1,281* | **4%** |
| 100 | 3,806 | **0%** | *1,373* | **2%** |

Table 11: Comparing Hill Climbing with Evolutionary Testing by varying constraint schema probability size for Case Study 1 (part a) and Case Study 2 (part b), with a gradient landscape. The optimal setup for Evolutionary Testing (Uniform 120 – uniform crossover, 120 individuals in a population) is used, as found in the answer to RQ7. The number of average evaluations is recorded unless the target could not be covered with a 100% success rate, in which case success rate is recorded instead. A figure appears in bold if the search was significantly worse than Evolutionary Testing with discrete recombination, or in italics if the reverse was true. RHC is significantly the best performer for Case Study 1, while the Alternating Variable Method is significantly the best performer with Case Study 2

(a) Flat landscape

| **Building block probability** | **Search type** Uniform 120 | AVM | RHC | RHC-AVM |
|---|---|---|---|---|
| 0.50 | 3,946 | **0%** | *532* | **35,878** |
| 0.25 | 6,723 | **0%** | *1,469* | **80%** |
| 0.20 | 7,410 | **0%** | *2,006* | **46%** |
| 0.10 | 9,797 | **0%** | *4,691* | **0%** |
| 0.05 | 74% | **0%** | *9,253* | **0%** |
| 0.01 | 0% | 0% | *98%* | 0% |

(b) Gradient landscape

| **Search type** Uniform 120 | AVM | RHC | RHC-AVM |
|---|---|---|---|
| 2,637 | *294* | *592* | *294* |
| 9,176 | *420* | *1,769* | *420* |
| 10,631 | *444* | *2,304* | *444* |
| 17,022 | *492* | *5,229* | *492* |
| 20,392 | *519* | *11,263* | *519* |
| 26,580 | *540* | **54,299** | *540* |

Table 12: Comparing Hill Climbing with Evolutionary Testing with the contending constraint schema example of Case Study 4 (Figure 13). The optimal setup for Evolutionary Testing (Uniform 120 – uniform crossover, 120 individuals in a population) is used, as found in the answer to RQ7. The number of average evaluations is recorded unless the target could not be covered with a 100% success rate, in which case success rate is recorded instead. A figure appears in bold if the search was significantly worse than Evolutionary Testing with discrete recombination, or in italics if the reverse was true. RHC is the best performer, and is significantly better than Evolutionary Testing

| **No. of contending constraint schemata** | **Search type** Uniform 120 | AVM | RHC | RHC-AVM |
|---|---|---|---|---|
| 0 | 2,623 | **18,137** | *350* | *2,035* |
| 1 | 2,709 | **18,661** | *348* | *1,975* |
| 2 | 2,766 | **20,483** | *355* | *2,182* |
| 3 | 2,818 | **23,590** | *349* | *2,284* |
| 4 | 2,921 | **23,799** | *372* | *2,285* |
| 5 | 2,969 | **98%** | *372* | *2,106* |
| 6 | 3,101 | **94%** | *343* | *2,282* |
| 7 | 3,031 | **96%** | *362* | *2,169* |
| 8 | 3,102 | **96%** | *370* | *2,196* |
| 9 | 3,077 | **94%** | *384* | *2,484* |
| 10 | 3,168 | **94%** | *397* | *2,446* |
| 11 | 3,103 | **96%** | *376* | *2,409* |
| 12 | 3,163 | **96%** | *364* | *2,419* |
| 13 | 3,193 | **92%** | *360* | *2,334* |
| 14 | 3,194 | **94%** | *373* | *2,444* |
| 15 | 3,224 | **90%** | *386* | *2,321* |
| 16 | 3,292 | **86%** | *391* | *2,576* |
| 17 | 3,270 | **86%** | *403* | *2,483* |
| 18 | 3,333 | **84%** | *383* | *2,472* |
| 19 | 3,272 | **86%** | *404* | *2,408* |
| 20 | 3,478 | **84%** | *413* | *2,429* |

Table 13: Hill Climbing and Evolutionary Testing with Case Study 6 (Figure 16) involving a local optimum. The number of average evaluations is recorded unless the target could not be covered with a 100% success rate, in which case success rate is recorded instead. A figure appears in bold if the search was significantly worse than Evolutionary Testing with discrete recombination, or in italics if the reverse was true. With this case study, Evolutionary Testing with discrete recombination or one-point crossover always outperform each Hill Climbing algorithm, and significantly so in the majority of cases

| **Size of Local Optimum** | **Evolutionary Testing** Discrete recombination | Uniform crossover | One-point crossover | Headless chicken crossover | No crossover | **AVM** | **RHC** | **RHC-AVM** |
|---|---|---|---|---|---|---|---|---|
| 0 | 4,054 | 3,946 | **6,688** | **0%** | **0%** | **0%** | *532* | **35,878** |
| 1 | 4,139 | 4,026 | **7,585** | **0%** | **0%** | **0%** | **11,113** | **16%** |
| 2 | 4,491 | 4,399 | **84%** | **0%** | **0%** | **0%** | **2%** | **0%** |
| 3 | 5,207 | 4,969 | **32%** | **0%** | **0%** | **0%** | **2%** | **0%** |
| 4 | 10,007 | 9,436 | **10%** | **0%** | **0%** | **0%** | **0%** | **0%** |
| 5 | 62% | 82% | 4% | 0% | 0% | 0% | 0% | 0% |
| 6 | 26% | 24% | 0% | 0% | 0% | 0% | 0% | 0% |
| 7 | 4% | 4% | 0% | 0% | 0% | 0% | 0% | 0% |
| 8 | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% |
| 9 | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% |

which avoids the need to make assumptions regarding the normality of the sample means – *i.e.* that the conditions for a parametric test have been met. Such additional analysis could introduce further possible sources of error into the study.

Another source of bias comes from the choice of case studies used. This affects not only the internal validity of the empirical study, but also its external validity – the extent to which it is possible to generalise from the results obtained. The empirical study performed in this paper involves case studies that were specifically designed to test the effect of a particular program factor on the performance of the crossover operator. The creation of synthetic programs for the experiments allowed for results to collated reliably. The presence of a program factor is guaranteed by construction, allowing it to be investigated in isolation and without interference from other artefacts in the code – as might be the case with a program used from an external source. The synthetic design also allowed for the experiments to be tightly controlled. Variations could be made to the program structure easily in the required manner, with changes in crossover performance accurately observed. However, the case studies are merely exemplars of the program factors studied. They do not represent all cases in which a particular crossover-over affecting factor may manifest itself into a program. Caution is required, therefore, before making claims as to whether these results would be observed on other programs. However, the results show that cases do exist where there is a statistically significant relationship between the program factors and the crossover operators studied.

# 5 Discussion

This section summarises each program factor and discusses the potential ways in which that program factor may manifest itself in programs in practice, followed by a discussion of the wider impact of the findings made.

## 5.1 Program Factors

**Program Factor 1. Number of Input Condition Conjuncts**

In order for crossover to do useful work, there needs to be a sufficient number of constraint schemata for the crossover operator to recombine. This means there should be an accompanying sufficient number of input condition conjuncts, from which building block constraint schemata can form. If there are few input condition conjuncts, it is unlikely crossover will make any contribution the test input search. Program branches may involve several input condition conjuncts where the branch is nested within a loop, with each loop iteration adding a new conjunct to the condition. Loops tend to operate over data structures composed of several primitive types, for example arrays of numbers or strings of characters. Large numbers of input condition conjuncts may also arise for modules with state behaviour [28], where an action needs to be repeated several times in order for a target branch to be executed, for example the filling up of a stack, and for programs operating over dynamic data structures.

**Program Factor 2. Search Difficulty Involved in Satisfying Input Condition Conjuncts**

Crossover will become more important in progressing the search when input condition conjuncts are hard to satisfy, *i.e.* they cannot be satisfied easily at random, often due to a coarse fitness landscape. or the fitness landscape is coarse. Coarse fitness landscapes tend to be the result of the use of intermediate variables, which potentially serve to 'squash' a range of input variable values to a smaller number of distinct fitness values. This lends crossover to short functions, where the variables pertaining to branch predicates are more likely to be input variables to the function.

**Program Factor 3. Evaluation of the Input Condition with Respect to the Target Structure**

Building block schemata will not form for input condition conjuncts not always evaluated by the program. Nested structures, short-circuit evaluation, guarded jumps and premature termination of loops lead to partial evaluation of the input condition. Such structures inhibit the crossover operator in positively contributing to the search – any test input is likely to be found through the sole use of mutation alone.

**Program Factor 4. Input Condition Conjuncts over Disjoint Sets of Input Variables**

To avoid clashes of input vector values during crossover, each conjunct should reference different input variables. Therefore the program needs to have a large, multi-dimensional input space to accompany an abundant number of independent input condition conjuncts for crossover to work well.

**Program Factor 5. Individual Impact of Input Condition Conjuncts on Conditions Guarding the Target**

Building block constraints should involve as few input variables as possible, to avoid their possible disruption when input vectors are recombined. Building blocks involve few input variables when they involve few input condition conjuncts, *i.e.* the program is structured so that each input condition conjunct can make a direct impact of the value of the fitness function for executing the target. Again, this implies programs involving conditions that relate directly to individual input variables, rather than intermediate variable values that are dependent on the values of several inputs.

## 5.2 Implications for Further Research

The identification of the above program factors and their relationship to crossover performance has the potential to impact the following areas of research:

**Choice of Search Technique**

The presence or absence of the above program factors explain why Evolutionary Search may or may not represent a good choice of search technique. If crossover is unlikely to impact the progress of an Evolutionary Search, simpler and faster search techniques like Hill Climbing may represent a better choice of algorithm.

**Metrics for Evolutionary Structural Testability**

The above program factors may help form the basis of metrics for Evolutionary Structural Testability. Previous work has attempted to link traditional software engineering metrics with Evolutionary Structural Testability without success. This fact, coupled with the findings presented in this paper, suggest that new metrics are needed that are based on the structure of the input domain and the program itself.

**A Basis for New Testability Transformations**

Testability transformations are source-to-source program transformations that aim to change a program so that test inputs may be more easily generated using search-based techniques [13]. Testability transformations have been developed, for example, to remove flat fitness landscapes as a result of intermediate boolean flag variables [3]. Once test inputs have been successfully generated for the target branch, the transformed version may be thrown away. The findings of this paper show how crossover is affected by different program factors. Where crossover is adversely affected, the program could be transformed so that the crossover operator becomes more effective. As such, the identification of program factors affecting crossover in this paper may help form the basis of the development of new Testability Transformations which can improve the Evolutionary Search for structural test inputs. It is worth noting that two such Testability Transformation has already been developed – for removing nesting in a program [27] and the removal of intermediate boolean flag variables which flatten the fitness landscape [3, 13]. This paper brings an understanding as to why these transformations improve the Evolutionary Search process in the context of crossover. Further Testability Transformations may be developed, for example, to decompose the input condition down into further input condition conjuncts – to aid crossover – and to break down input condition conjuncts so that they are focussed on disjoint sets of input variables.

# 6 Related work

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.* [29, 8], were an early attempt to identify the types of fitness landscapes in which crossover worked well. Watson *et al.* [33] 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 [18] 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.* [31] produced the 'Ignoble Trail' fitness functions, which are characterisations of crossover-deceptive landscapes for which crossover is shown to be provably harmful.

This paper has focussed on the characterisation of programs for which structural test data search by Genetic Algorithms, referred to as Evolutionary Testing, will perform well. Recent studies by Harman and McMinn [14, 15] on a selection of open source and industrial programs revealed that simple hill climbing search in the form of Korel's Alternating Variable Method (Alternating Variable Method) was able to outperform Evolutionary Testing in covering the vast majority of branches considered. For the small number of branches for which Evolutionary Testing outperformed the Alternating Variable Method, a 'Royal Road'-type property was found to be present. The present paper builds on work presented in [24], showing how programs give rise to fitness functions that cause crossover to perform well.

By contrast, there has been a comparatively large volume of work devoted to when search-based techniques *do not* work very well for certain program structures. This is usually because the fitness landscape is flat, offering no guidance to the search technique. This is the case with the so-called 'flag' problem [12, 6, 4, 3]. Testability transformation [13, 10] has been proposed to amend the program as a temporary measure so as to remove these awkward landscape features so that the search can work more effectively. Nested structures [26, 27, 5], short-circuit evaluation [5] and unstructured programming [16] have been shown to cause problems for Evolutionary Testing, however, this is the first paper to show both theoretically and empirical why this is the case with respect to the crossover operator of Genetic Algorithms.

# 7 Conclusions and future work

This paper has investigated how programs affect the efficacy of the crossover operator for test data generation using Genetic Algorithms (Evolutionary Testing). The paper found several characteristics that a program needs to have in order for crossover to work well. The characteristics were identified by theoretical analysis based on modelling building blocks of a test data search using Evolutionary Testing constraint schemata [15], and were confirmed through an empirical study.

The empirical study also found that the crossover operator traditionally used with the popular Wegener model of Evolutionary Testing [34], discrete recombination, is frequently outperformed by standard uniform crossover. It also found that Hill Climbing tends to outperform Evolutionary Testing for many of the programs for which Evolutionary Testing works well, with the exception of those with entrapping local optima.

The paper has presented results using Evolutionary Testing with procedural C programs. Further work needs to investigate the efficacy of crossover in the context of programs written in other languages and paradigms. The work has also focussed on branch coverage – further analysis is required for other structural coverage types. Whilst the main, common forms of crossover have been investigated (one-point, uniform and discrete recombination), further work would be needed to evaluate the affects on other types of crossover, for example operators that average values from parent chromosomes.

Future work may seek to develop algorithms to detect the presence or non-presence of certain features in a program, in order to decide which search technique is best to apply. The identification of different program factors may serve as a basis for a rigorous set of metrics that help to define the evolutionary testability of a program. A related avenue for future work is the potential use of Testability Transformations to improve the performance of crossover as a search operator. Testability transformations have already been proposed, for example, to remove nesting and unstructured-ness from programs, identified in this paper to be sources of inhibiting crossover operator performance. Additional work in this area could target further barriers to full input condition evaluation, such as short-circuiting and premature exiting of loops over inputted data structures. Furthermore, Testability Transformation may help ensure that input condition conjuncts have direct impact on the value of the fitness function, reducing the size of building blocks for crossover.

# Acknowledgements

Phil McMinn is supported in part by EPSRC grants EP/G009600/1 (Automated Discovery of Emergent Misbehaviour) and EP/F065825/1 (RE-COST: REducing the Cost of Oracles for Software Testing).

The author would like to thank the referees of the conference version of this manuscript for their helpful comments; as well as attendees at SSBSE 2010 who gave useful feedback and suggestions on the work following its presentation, including Riccardo Poli, Simon Poulding and Mark Harman.

# References

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

[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), Lecture Notes in Computer Science vol. 2724*, pages 2442–2454, Chicago, USA, 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, New York, USA, 2002. Morgan Kaufmann.

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

[7] E. Falkenauer. *Genetic Algorithms and Grouping Problems*. Wiley, 1998.

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

[9] D. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1989.

[10] M. Harman, A. Baresel, D. Binkley, R. Hierons, L. Hu, B. Korel, P. McMinn, and M. Roper. Testability transformation - program transformation to improve testability. In *Formal Methods and Testing, Lecture Notes in Computer Science*, volume 4949, pages 320–344. Springer-Verlag, 2008.

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

[12] M. Harman, L. Hu, R. Hierons, A. Baresel, and H. Sthamer. Improving evolutionary testing by flag removal. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002)*, pages 1359–1366, New York, USA, 2002. Morgan Kaufmann.

[13] 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.

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

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

[16] R. Hierons, M. Harman, and C. Fox. Branch-coverage testability transformation for unstructured programs. *The Computer Journal*, 48(4):421–436, 2005.

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

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

[19] T. Jones. Crossover, macromutation and population-based search. In *Sixth International Conference on Genetic Algorithms*, pages 73–80. Morgan Kaufmann, 1995.

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

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

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

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

[24] P. McMinn. How does program structure impact the effectiveness of the crossover operator in evolutionary testing?

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

[26] P. McMinn, D. Binkley, and M. Harman. Testability transformation for efficient automated test data search in the presence of nesting. In *Proceedings of the UK Software Testing Workshop (UKTest 2005)*, pages 165–182. University of Sheffield Computer Science Technical Report CS-05-07, 2005.

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

[28] P. McMinn and M. Holcombe. The state problem for evolutionary testing. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2003), Lecture Notes in Computer Science vol. 2724*, pages 2488–2497, Chicago, USA, 2003. Springer-Verlag.

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

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

[31] J. N. Richter, A. Wright, and J. Paxton. Ignoble trails - where crossover is provably harmful. In *Parallel Problem Solving from Nature (PPSN X), Lecture Notes in Computer Science Vol. 5199*, pages 92–101, Dortmund, Germany, 2008.

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

[33] R. A. Watson, G. S. Hornby, and J. B. Pollack. Modeling building-block interdependency. In *Parallel Problem Solving from Nature V, Lecture Notes in Computer Science Vol. 1498*, pages 97–106, Amsterdam, The Netherlands, 1998.

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

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

[36] 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.