Using Automated Search to Generate Test Data for Matlab

Search-Based Software Engineering Track

Sion Ll. Rhys
Dept. of Computer Science
University of York
York YO10 5DD, UK

Simon Poulding
Dept. of Computer Science
University of York
York YO10 5DD, UK
smp@cs.york.ac.uk

John A. Clark
Dept. of Computer Science
University of York
York YO10 5DD, UK
jac@cs.york.ac.uk

ABSTRACT

The critical functionality of many software applications relies on code that performs mathematically complex computations. However, such code is often difficult to test owing to the compound datatypes used and complicated mathematical operations performed. This paper proposes the use of automated search as an efficient means of generating test data for this type of software. Taking Matlab as an example of widely-used mathematical software, a technical framework is described that extends previous work on search-based test data generation in order to handle matrix datatypes and associated relational operators. An empirical evaluation demonstrates the feasibility of this approach.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; I.2.2 [**Artificial Intelligence**]: Automatic Programming; G.4 [**Mathematics of Computing**]: Mathematical Software; G.1.6 [**Numerical Analysis**]: Optimisation

General Terms

Algorithms, Experimentation, Verification

Keywords

Search-Based Software Engineering, Genetic Algorithms, Matlab

1. INTRODUCTION

The generation of test data is arguably the most mature application area within the field of search-based software engineering [5]. A substantial body of work describes the application of a wide range of metaheuristic search techniques—ranging from local search methods to sophisticated nature-inspired algorithms—to the problem of deriving test inputs. Search-based test data generation (SBTDG) has been applied to many different programming languages encompassing a variety of coding paradigms; these include Pascal [6], Ada 95 [15], C [7], C♯ [10], and Java [12, 4]. McMinn [9] provides a comprehensive survey of the state-of-the-art in SBTDG.

This paper extends the use of SBTDG techniques to software and programming languages that perform mathematically complex computations. Many modern computing applications rely on software that performs such computations. Examples include digital signal processing in MP3 music players, digital television transmission, and medical imaging; financial market and other economic models; and a wide range of computer-based simulations that model real-world physics, such as weather forecasting. Errors in the mathematical computations at the heart of these systems can lead to poor sound quality, video artefacts, unreliable diagnoses, financial loss and inaccurate storm predication. There is a strong motivation to test code that performs mathematical computations, but its very nature—compound datatypes such as vectors, matrices and complex numbers; and manipulations of input variables using complicated mathematical operators—can make this a difficult task. However, the preponderance of numeric datatypes in mathematical software suggests that the use for search-based test data generation, which has demonstrated success with these datatypes, is appropriate.

We use Matlab [1] as an exemplar of mathematical software. Matlab is a mathematical computing platform extensively used by mathematicians, scientists and engineers to analyse and visualise data, implement algorithms, and build models. The core capability of Matlab is the ability to store data as matrices and efficiently manipulate the data using linear algebra. An extensive range of 'toolkits' provide additional functionality in specific application areas, such statistical analysis, optimisation, image processing, signal processing, symbolic mathematics and wavelet analysis. Simulink [3]—a simulation and model-based design tool—uses Matlab as its mathematics engine. A significant amount of academic research and industrial engineering relies on results calculated using Matlab and Simulink, and so there is an obvious benefit in efficiently generating test data that can verify software written for the Matlab platform.

The novel capabilities and domain-specific usages of mathematical software in general, and of Matlab in particular, present new challenges to search-based test data generation. It is these challenges that this paper addresses by contributing the following solutions:

GECCO'09, July 8–12, 2009, Montréal Québec, Canada.

- a technical framework for instrumenting Matlab code and performing dynamic test data generation;
- a method for representing matrix data types in the genotype of a genetic algorithm;
- an extended set of *branch distance* functions to support relational operations on matrix data types;
- a rank-based selection mechanism that combines *branch distance* and *approach level* cost component without the need for scaling;
- the identification of linear algebra functions that present particular difficulties when present in relational predicates.

The paper is structured as follows. The next section summarises related work on dynamic structural testing using automated search. Section 3 describes the technical architecture of an instrumenter for Matlab code. Section 4 details the framework used to implement search-based test data generation in Matlab, and suggest novel cost metrics to accommodate matrix datatypes. An implementation of a genetic algorithm using this framework is described in section 5. An empirical evaluation is performed to demonstrate the practicality and efficacy of generating Matlab test data using the genetic algorithm. The experimental design and results of this evaluation are described in section 5.

2. RELATED WORK

The work described in this paper follows the search-based approach to dynamic test data generation first described by Korel [6] and later extended by Tracey *et al.* [13, 14, 15], Wegener *et al.* [17], and others.

2.1 Branch Distance

In this approach, the *cost* (or, fitness) of a candidate test input is determined dynamically by running an instrumented version of the software under test (SUT). Often the objective is to identify test data that executes a particular branch, path or similar structural element in the SUT, and so the instrumentation determines a *branch distance*: a measure of how 'close' the code came to executing a target branch when it is run with particular input values. The branch distance depends on the values taken by the expressions on each side of relational predicates that form the branch predicate in question, and on the nature of the relational operators. In this paper we use the branch distance metrics suggested by Tracey which return a value of 0 if a relational predicate evaluates as true, otherwise a positive non-zero value using formulae listed in Table 1. When the branch predicate is formed by a conjunction or disjunction of relational predicates, the overall branch distance is calculated using the formulae in the final two rows of the table.

As an example, consider the Matlab function shown in Fig. 1. Assume for the moment that the datatypes of the input variables a and b are scalars, rather than matrices, so that Tracey's branch distance metrics may be applied without extension. The testing objective is to exercise the true branch of the `if` statement at line 02. For the candidate inputs $a = 10$, $b = 8$, the single relational predicate in the branch predicate $a \leq b$ evaluates to false, and so the branch distance metric is calculated as $a - b + K = 10 - 8 + K = 2 + K$.

Table 1: Branch distance metrics defined by Tracey *et al.*. K is a small positive non-zero constant.

Predicate	Branch Distance when False
$x < y$	$x - y + K$
$x \leq y$	$x - y + K$
$x > y$	$y - x + K$
$x \geq y$	$y - x + K$
$x = y$	$\text{abs}(x - y) + K$
$x \neq y$	K
$P \wedge Q$	$\text{branchDist}(P) + \text{branchDist}(Q)$
$P \vee Q$	$\min\{\text{branchDist}(P), \text{branchDist}(Q)\}$

```
01 function c=comp(a,b)
02     if a<=b
03        if a==b
04             c=0;
05         else
06             c=-1;
07         end;
08     else
09         c=1;
10     end;
```

Figure 1: Matlab function using scalars.

In this paper, new branch distance metrics are proposed for relational predicates involving matrix variables: see section 4.4.

2.2 Approach Level

In addition to the branch distance, Tracey and Wegener also consider a further component of the cost function: the *approximation* or *approach level*. When the instrumented SUT is run with candidate test inputs, the execution path may fail to reach the branch under consideration. In this case, the approach level counts the number of branching statements between the target branch and the earlier branch at which the actual execution path diverged from the desired path.

Consider again the function in Fig. 1 with a new testing objective of exercising the true branch of the `if` statement at line 03. With the candidate inputs $a = 10$, $b = 8$, the branching statement at line 02 is false, so line 03 is not reached. The approximation distance in this case is therefore 1.

Both Tracey and Wegener calculate an overall cost as a combination of approach level and branch distance, where the branch distance is calculated for the branching statement at which the desired and actual execution paths diverged. Tracey and Wegener differ in the formula used to combine the approach level and branch distance. McMinn [9] argues that Wegener's formulation results in a cost landscape that is easier to navigate. However, this formulation must 'normalise' the branch distance into the range $[0, 1]$ to avoid it overwhelming the approach level component.

This paper proposes that an overall cost function need not be calculated, so avoiding the issue of normalisation. Instead approach level and branch distance are considered as separate components of cost that are used to rank candidate input values. This mechanism is described in section 5.2.

2.3 Search Method

A suitable search method is applied to minimise the overall cost. The objective is to locate input values for which the overall cost is zero indicating that the target branching statement is executed *and* that it evaluates to the desired value. Both the branch distance and approach level are designed to 'guide' the search method: a branch distance of $1 + K$ indicates that the input values are in some sense 'closer' to taking the desired path at a branching statement than input values that return a branch distance of $2 + K$.

Korel employs an alternating variable technique: each input variable is considered in turn and if it is found that changing the variable affects the branch distance, a local search is used to find the value of the variable that minimises the branch distance. The variable is then held at this value while the other input variables are considered. Tracey applies simulated annealing as the search technique in [13, 14] and evolutionary algorithms in [15]. Wegener [17] also applies an evolutionary algorithm.

Based on the success reported by Tracey and Wegener, the empirical work in this paper uses a genetic algorithm as the search method. Novel aspects of the algorithm are the use of a representation and genetic operators that accommodate matrix variables (section 5.1), and a selection mechanism that avoids the need to calculate a single overall cost value for a candidate input value (section 5.2).

3. INSTRUMENTATION

This section describes the technical architecture of an automated instrumenter for software written in the Matlab programming language.

The instrumenter is written in Java. This is a natural choice since the Matlab engine itself executes in a Java Virtual Machine (JVM), and Java classes may be invoked from within Matlab code. This not only enables the instrumenter to be run within the Matlab environment, but also allows supporting data derived during instrumentation to be stored within the Matlab variable workspace. This data is used during automated search to interpret the dynamic execution of the instrumented Matlab code.

Functions and scripts written in the Matlab programming language are stored as text files typically referred to as *m-files*. The name of the m-file to be instrumented, and the datatypes of its arguments, are passed to the instrumenter which processes the file using three components:

1. A parser and lexer constructs an abstract syntax tree for the Matlab code. This component uses Java classes generated automatically using the ANTLR tool [11] from a grammar describing the Matlab programming language. Since no complete grammar for the language has been published, a weaker, partially complete specification was used for parsing m-files. This specification places a few constraints on the code permitted in the m-files, but any code violating these constraints can be adapted using minor syntax changes that preserve functionality.

2. The second component walks the syntax tree and adds new Matlab code that enables instrumentation. This additional code takes the form of Java method calls to classes provided by the instrumenter. These calls identify the branch predicate being executed, and if the branching statement occurs within the body of a loop, the iteration of the loop is also recorded. For each relational predicate in the branch predicate, the method calls record the value of the expressions on each side of the relational operator and operator itself. An alternative approach would have been to calculate the branch distance directly in the instrumenting code and store only this value. However, recording the value of each expression in the branch predicate, and then calculating the branch distance from these values within the search algorithm, allows greater flexibility as to the form of the branch distance metrics that are applied.

3. The third component of the instrumenter uses the syntax tree to write a new m-file that combines the original code with the new instrumenting code.

4. SEARCH METHOD FRAMEWORK

This section describes a flexible framework used to implement search methods for generating input data in Matlab.

Like the instrumenter, the framework is written in Java and runs within the JVM used by the Matlab environment. The framework provides functionality common to all search methods. A search method, implemented as a Java class, can be 'plugged' into the framework and take advantage of the features provided by the framework. The features are described in the subsections below.

4.1 Input Variable Datatypes

Since Matlab uses dynamic typing, the datatypes of the input variables cannot be derived automatically from the software under test and so must be defined explicitly to the framework.

Currently, the following datatypes are supported: integer and real scalars; integer and real matrices; and Booleans. For all but the Boolean datatype, the permitted domain of the variable is specified as a bounded interval. For matrix datatypes, the matrix dimensions—the number of rows and columns—are specified, and all the matrix elements are assumed to have the same domain.

4.2 Testing Objective

The framework permits the user to specify the testing objective. The objective specifies not only the branching statement that the test inputs should exercise and the desired outcome of the branch predicate, but also the branches to be taken to reach the target branch. Where branches occur within loops, the user is able to define which branch is to be taken on each iteration of the loop. The testing objective is therefore defined as partial path from the entry to the function to the target branch. This is a stronger coverage criterion than the branch coverage discussed in the section 2.

4.3 Instrumented Code Execution

The framework provides a mechanism to execute the instrumented software with specified input variables, and to record the branches executed and the values taken by expressions in the corresponding branch predicates (the latter as described in section 3). The instrumentation results are stored in the Matlab variable workspace.

Note that although the framework is initiated from the Matlab command line, it executes as Java code running in the Matlab JVM. Therefore the framework must make a call

Table 2: Branch distance metrics for relational predicates applied to matrices.

Predicate	Branch Distance when False
$A < B$	$\sum_i \sum_j \max\{(a_{ij} - b_{ij}), 0\} + K$
$A \leq B$	$\sum_i \sum_j \max\{(a_{ij} - b_{ij}), 0\} + K$
$A > B$	$\sum_i \sum_j \max\{(b_{ij} - a_{ij}), 0\} + K$
$A \geq B$	$\sum_i \sum_j \max\{(b_{ij} - a_{ij}), 0\} + K$
$A = B$	$\sum_i \sum_j \text{abs}(a_{ij} - b_{ij}) + K$
$A \neq B$	K

'back' to Matlab engine in order to run the instrumented Matlab code. This is achieved using a Java library, the Java Matlab Interface, supplied as part of the Matlab platform.

4.4 Matrix Branch Distance Metrics

Using the data obtained by executing the instrumented Matlab code, the framework calculates the branch distance for each of the branch predicates on the target path. As will be described in section 5.2, there is no need to explicitly calculate the approach level.

For scalar and Boolean datatypes, the branch distance metrics are those defined by Tracey and listed in Table 1. However, the branch distance metrics must be extended to accommodate relational operators applied to matrices. In Matlab, such relational predicates are only valid if the expressions on either of the operator resolve to matrices having the same dimensions, i.e. the same number of rows and columns. The predicate is then *true* if and only if the relation is true when applied to all the corresponding elements of the two matrices.

If matrices A and B have the same dimensions, and the matrix elements are denoted as $\{a_{ij}\}$ and $\{b_{ij}\}$ (where i indicates the row, and j the column), then the relational operators for matrices can be expressed more formally as:

$$A \text{ op } B \quad \Leftrightarrow \quad a_{ij} \text{ op } b_{ij} \quad \forall i, j \tag{1}$$

using op to denote the relational operator.

Thus the predicate,

$$\begin{bmatrix} 1 & 5 \\ 2 & 3 \end{bmatrix} < \begin{bmatrix} 4 & 2 \\ 6 & 8 \end{bmatrix} \tag{2}$$

is false since when considering the elements at row 1, column 2, the predicate $5 < 2$ is false.

If a matrix relational predicate is false, the cost associated with the predicate is given by the new metrics listed in Table 2. Given the definition of matrix relational operators in (1), these new metrics are a natural extension to those defined by Tracey for scalars. The summations are applied across all rows (index i) and columns (index j) of the matrices. As for scalars, if the predicate is true, the cost is zero.

5. GENETIC ALGORITHM

The framework described above is capable of supporting a wide range of search methods. To enable an empirical evaluation (see section 6), two search methods were implemented: a generational genetic algorithm with elitism, and random search. The objective of the experimental work is a practical demonstration of the instrumentation and search method framework, and to illustrate the feasibility of using automated search to generate test data for mathematical software, such as Matlab code. We do not claim that this choice of algorithm, its operators and parameters, is most efficient search method for this problem; rather, we employ—for the purpose of demonstration—a straightforward implementation of a search method that is often used in search-based test data generation.

Nevertheless, the genetic algorithm has two novel features: the representation and associated operators accommodate matrix datatypes, and the selection method avoids the complication of normalising and combining the approach level and branch distance costs. It is these features that are described in this section.

5.1 Representation and Operators

The GA represents candidate test inputs—values for each of the input variable for the software under test—with a genome constructed as a vector of real numbers. This representation supports all of the datatypes listed in section 4.1. Valid values for the each of vector's elements are determined by the datatype: integers for integer datatypes, {0,1} for Booleans; and by the domain of each variable specified by the user.

For example, if the input variables are a real scalar, an integer 2×2 matrix, and a Boolean, then the genome is a vector of reals with 6 components: $(g_1, g_2, g_3, g_4, g_5, g_6)$. In order to run the software under test, the genome vector of an individual is mapped to variables with required datatypes, creating the input variables:

$$A = g_1, \quad B = \begin{bmatrix} g_2 & g_3 \\ g_4 & g_5 \end{bmatrix}, \quad C = \begin{cases} \text{true} & \text{if } g_6 = 1 \\ \text{false} & \text{if } g_6 = 0 \end{cases} \tag{3}$$

This representation influences the choice of initialisation, crossover and mutation operators used by the GA.

5.1.1 Initialisation

Each element of individual's genome vector is initialised by randomly sampling a value from a uniform probability distribution across the corresponding variable's domain.

5.1.2 Crossover

The GA uses n-point crossover. The value of n is a parameter to the algorithm. Crossover points are permitted between vector elements that map to the same matrix (e.g. between g_2 and g_3 in the example above). Since the dimensions of input matrix variables are fixed, the size of the genome and its mapping to input variables are also fixed, and so such crossover points do not lead to invalid individuals.

5.1.3 Mutation

Individuals are mutated at a rate determined by a parameter to the algorithm. A value is sampled from a uniform distribution over the range $[-m, m]$, and the value is added to a randomly selected element in the individual's genome. The value of m is a parameter to the algorithm. If a mutation causes an element in the genome to take an invalid value (such as one outside the corresponding input variable's domain), it is adjusted to the nearest valid value.

Table 3: Individuals ranked by branch distances.

Rank	Branch 1	Branch 2	Branch 3
6	0	0	3
5	0	0	8
4	0	4	–
3	0	18	–
2	1	–	–
1	24	–	–

5.2 Selection

In related work on search-based test data generation, selection is based on a single numerical value obtained by combining the two cost components: the approach level and branch distance (see section 2). The branch distance must be 'normalised'—to the range $[0, 1]$, for example—in order to avoid large branch distances overwhelming the approach level.

An alternative approach is taken by the work described here that avoids the need to combine the two cost components. Selection uses a ranking of individuals according to the branch distance calculated at each branch predicates on the target execution path.

Individuals are ranked in ascending order of the branch distance at the first branch in the code. Individuals that evaluated the first branch correctly have a branch distance of zero and so occur at the top of the list. Where individuals have the same branch distance for the first branch, they are ranked in ascending order of the branch distance at the second branch. Any individuals with the identical branch distances for the first two branches are ranked according to the branch distance at the third branch, and so on until the end of the target path.

Note that there is no need to calculate the approach level since the ranking method implicitly rewards individuals that have come closest to executing the branch at the end of the target path.

An example of ranked individuals is given in Table 3. In this example, there are three branches in that target path.

Selection based on the rank of individuals occurs at two places in the algorithm. Firstly, a fixed number, k, of elite individuals—those at the top of the list—are carried forward unchanged to the next generation. Secondly, parent individuals are sampled for reproduction by crossover using 'roulette wheel' selection based on an individual's rank: those with a higher rank (towards the top of the list) are more likely to be selected.

6. EMPIRICAL EVALUATION

The sections above describe a method of instrumenting Matlab code; a framework for implementing search methods that dynamically execute the instrumented code; extensions to branch distance metrics, genome representation, and genetic operators to accommodate matrix datatypes; and a selection method that avoids complications arising from combining approach level and branch distance costs.

The objectives of the experimental work described in this section of the paper are to demonstrate:

- a practical implementation of the instrumenter and search method framework;
- the feasibility of using automated search to generate test data for Matlab code;
- the effectiveness of the proposed branch distance metrics, representation, operators, and selection method.

6.1 Experimental Design

Our approach is to apply two search methods—the genetic algorithm described in section 5, and random search—to four Matlab functions. The random search method simply samples input values using a uniform distribution across the domain of the input variables. We compare the number of evalutions, i.e. the number of times the instrumented software is executed by each of the search methods, until input data are found that execute the target path in code. The random search method acts a baseline: we intend to show that use of appropriate branch distance metrics, representation, operators, and selection methods by the genetic algorithm is effective at guiding the algorithm to a solution using, on average, fewer evaluations than unguided, random search.

Four Matlab functions are used to illustrate the capabilities of the instrumenter and search method framework. The salient features of each function are described here; the source code of each is provided in the appendix.

binary_search This is a real-world function taken from the Matlab Central File Exchange [2]. The input variables are a list of real numbers—supplied as a 1×512 matrix—and a real scalar. The function returns the location of the scalar in the list. The scalar and matrix elements have a domain of $[0, 20]$. The function includes branching statements inside nested loops. The search method framework permits the user to specify different branches to be taken on each iteration of loop. We specify two such target paths: one of 12 branches, and one of 16 branches.

matrix_ops This constructed function takes six input variables, all of which are 2×2 real-valued matrices with elements in the range $[0, 20]$. We define a target path that requires the input data to satisfy a series of branch predicates that contain relational operators applied to matrices.

is_singular This simple function takes a single input variable: a 3×3 integer matrix where each element has the domain $[-50, 50]$. We define a target path that is satisfied if and only if the matrix is singular, i.e. has a determinant of zero.

is_rank_deficient This function also takes a single input variable: a 3×3 integer matrix where each element has the domain $[-50, 50]$. We define a target path that is satisfied if and only if the rank of the matrix (the number of linearly independent rows or columns) is not equal to the number of rows (or columns). We will compare the difficulty of generating test data for this function with that of **is_singular**.

The two search methods–genetic algorithm and random search—are applied to the four Matlab functions. The GA parameters used for each function are listed in Table 4. We do not claim these parameters choices to be optimal: instead, the parameters were chosen based on some limited

'trail-and-error' investigation to give sufficient GA performance to enable practical experimentation.

A limit on the number of evaluations is applied to both algorithms in order to prevent long running algorithms trials: this limit is also listed in Table 4. It is set so that (where practical) the majority of runs of at least of one the algorithms find a solutions before this maximum. The statistical analysis applied to the experimental results (see section 6.2) is able accommodate trials that are terminated at this maximum number of evaluations.

Since the performance of each search method is dependent on the set of random numbers within the algorithm, 20 trials are performed for each combination of algorithms and function under test. For each trial, a different seed is supplied to pseudo-random number generator. The response data measured for each trial is the number of evaluations the search method requires to find input data that executes the target path.

6.2 Results and Analysis

A summary[1] of the results are given in Table 5 which reports the median number of evaluations that each algorithm requires to locate test data that executes the target path.

In addition, we analyse two types of significance described in the subsections below.

6.2.1 Statistical Significance

We analyse statistical significance in order to determine whether the observed difference between the algorithms is likely to have occurred by chance.

We apply the Mann-Whitney-Wilcoxon or rank-sum test [18]. This is a non-parametric test: it makes no particular assumptions as to the distribution of the response. An equivalent parametric test, such as the t-test, makes specific assumptions about the data—such as a normal distribution—and without careful analysis that the assumptions are met, the results of parametric tests can be unreliable [8]. An additional advantage is that the nature of the test is such that no special provision needs to be made to accommodate data from algorithm trials that reach the maximum number of evaluations.

The null hypothesis for the rank-sum test is that the responses for the two algorithms have identical distributions with equal medians; the alternative hypothesis is that the distributions are different. We use a 5% significance level: if the test returns a p-value of $< 5\%$, the null hypothesis may be rejected, indicating that any observed difference in the number of evaluations is unlikely to have occurred by chance. The p-value is reported in Table 5, and highlighted in bold where it indicates a statistically significant difference.

6.2.2 Scientific Significance

It is possible for the observed difference in algorithm performance to be statistically significant even though the actual magnitude of the difference—the *effect size*—is very small compared to the inherent variability in the results owing to the stochastic nature of the algorithms. To guard against this situation (which can occur when the number of experimental trials is excessive), we also test for scientific significance, i.e. that the effect size is sufficiently large to be scientifically important.

We use the Vargha-Delaney A statistic [16], which is also non-parametric, to assess effect size. This statistic is a value in the range $[0, 1]$. A value of 0.5 indicates no effect size: there is no difference in algorithm performance. Values closer to 0 or 1 indicate increasingly large effect size in favour of one algorithm or the other. For the experiments described here, an A statistic < 0.5 indicate that the genetic algorithm locates input data in fewer evaluations than random search; conversely, values > 0.5 indicate that random search is more efficient. The A statistics are reported in Table 5.

Whether a particular effect size is scientifically significant can depends on the context, but here we apply the guidelines given by Vargha and Delaney in [16]. We consider an A statistic < 0.36 or > 0.64 (which Vargha and Delaney suggest indicate a 'medium' or 'large' effect size) as scientifically significant, and these values are highlighted in bold in the table.

6.3 Discussion

6.3.1 Efficacy of Genetic Algorithm

For three of the five test cases, the genetic algorithm outperforms (in terms of the number of evaluations required to find test data that executes the target path) random search to a degree that is both statistically *and* scientifically significant. Given that no systematic attempt has been made to 'tune' the parameters of the GA, it is possible that even better performance is possible if the optimal parameters were assessed. The random search method, however, has no parameters that may be optimised.

In the remaining two test cases, there is no statistically significant difference in the performance of the genetic algorithm and random search.

This is convincing evidence that search-based software engineering can be applied to Matlab code, and in particular that the novel branch distance metrics proposed for matrices in section 4.4 are effective in guiding the GA to a solution.

It is interesting to note that while the same algorithm parameters were used to find input data that executes both the 12 and 16 branch paths in the **binary_search** function, the GA performs better than random search only for the 16 branch path. It is possible that the algorithm parameters used are far from optimal for the 12 branch, but we also speculate that this more 'simple' problem does not provide sufficient opportunity for the GA to demonstrate superior efficacy over random search.

6.3.2 'Difficult' Linear Algebra Functions

The GA showed significantly better performance over random search when required to locate a singular matrix (the function **is_singular**), but no difference from random search when required to locate a rank deficient matrix (the function **is_rank_deficient**). However, for the square, 3×3, matrices that were defined as input variable datatype for both functions, a singular matrix is mathematically equivalent to a rank deficient matrix. In other words, the test data found for **is_singular** also satisfies **is_rank_deficient**.

Again, it is possible that this difference is caused by the particular choice of algorithm parameters, and that tuning the parameters for **is_rank_deficient** may result in per-

[1]Full (unsummarised) experimental data is available at: http://www.cs.york.ac.uk/~smp/supplemental.

Table 4: Search Method Parameters

Parameter	binary_search (12 branch path)	binary_search (16 branch path)	matrix_ops	is_singular	is_rank_deficient
population size	10	10	100	100	100
crossover points (n)	2	2	2	2	2
mutation probability	5%	5%	5%	5%	5%
mutation range (m)	4	4	4	4	4
elite individuals (k)	2	2	2	2	2
max. evaluations	500	500	5,000	10,000	10,000

Table 5: Experimental Results

Statistic	binary_search (12 branch path)	binary_search (16 branch path)	matrix_ops	is_singular	is_rank_deficient
median evaluations: GA	40	70	1,950	4,800	10,000
median evaluations: random search	22.5	453	5,000	10,000	10,000
rank-sum p-value	9.57%	**0.200%**	**< 0.0001%**	**3.47%**	100%
Vargha-Delaney A statistic	0.655	**0.215**	**0.0525**	**0.306**	0.501

formance equivalent to that demonstrated for **is_singular**. Nevertheless, we believe that there is a fundamental difference in the difficulty of finding test data to satisfy these two linear algebra functions. The determinant function in **is_singular** can return a large range of values (infinitely many if the matrix has real-valued elements). However, the rank function in **is_rank_deficient** only returns a very limited range of integer values: from 0 to the number of rows in the matrix. Hence, the predicate using the determinant is able to provide much more guidance to the search algorithm as to whether changes to the input matrix are moving towards or away from values that would satisfy the relational predicate.

If this is indeed the cause of the difference in algorithm performance, it suggests an approach of transforming mathematical code—in this case, modifying linear algebra expressions to 'easier' expressions which have equivalent solutions—in order to facilitate search-based test data generation.

7. CONCLUSIONS

This paper proposed that, despite the challenges inherent in the nature of mathematical software, automated search is a feasible method for generating test data for such code.

A technical solution for instrumenting Matlab code was described and—with reference to existing work on search-based test data generation—novel cost functions, representations and algorithm operators were proposed that support real and integer matrices and their associated relational operators. The efficacy of this approach was evaluated empirically using four Matlab functions, including a real-world code example. The results show that a genetic algorithm to more effective that random sampling of input data for many functions, but that when some linear algebra functions modify the input variables, it may be particular difficult to guide the algorithm to a solution.

The results suggest further research in a number of areas. The capabilities of the instrumenter and search method framework could be enhanced to support other datatypes, such as complex numbers, Matlab cell arrays, and variable-sized matrices. For the purposes of the empirical work described in the paper, a relatively simple genetic algorithm was implemented. There is scope to research both more sophisticated operators—such a crossover and mutation operators that are specific to the datatypes of the input variables—and other search methods, such as simulated annealing and estimation of distribution algorithms. The difference in algorithm performance on functions that are mathematically equivalent suggests an opportunity to investigate the transformation of mathematical expressions in order to facilitate search-based test data generation.

Nevertheless, the work described in this paper is encouraging: a straightforward genetic algorithm using relatively naive operators is shown to be an effective method of generating test data for mathematically complex software.

8. REFERENCES

[1] MATLAB, version 2007b. The Mathworks, Inc., Natick, MA, USA.

[2] MATLAB Central File Exchange. www.mathworks.com/matlabcentral/fileexchange.

[3] SIMULINK. The Mathworks, Inc., Natick, MA, USA.

[4] A. Arcuri and X. Yao. Search based software testing of object-oriented containers. *Information Sciences*, 178, 2008.

[5] M. Harman. The current state and future of search based software engineering. *Proc. 29th Int'l Conf. Software Eng. (ICSE 2007), Future of Software Engineering (FoSE)*, pages 342–357, 2007.

[6] B. Korel. Automatic software test data generation. *IEEE Trans. Software Eng.*, 16(8):870–879, August 1990.

[7] K. Lakhotia, M. Harman, and P. McMinn. A multi-objective approach to search-based test data generation. In *Proc. 9th Annual Conf. on Genetic and Evolutionary Computation (GECCO'07)*, pages 1098–1105, ACM, 2007.

[8] N. L. Leech and A. J. Onwuegbuzie. A call for greater use of nonparametric statistics. Technical report, US Dept. Education, Educational Resources Information Center, 2002.

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

[10] B. W. Murrill. Automated test data generation and reliability assessment for software in high assurance systems. In *Proc. 10th IEEE High Assurance Systems Eng. Symp. (HASE'07)*, pages 409–410, IEEE Computer Society, 2007.

[11] T. J. Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. The Pragmatic Programmers, 2007.

[12] P. Tonella. Evolutionary testing of classes. In *Proc. 2004 ACM SIGSOFT Int'l Symp. Software Testing and Analysis (ISSTA'04)*, pages 119–128. ACM, 2004.

[13] N. Tracey, J. Clark, and K. Mander. Automated program flaw finding using simulated annealing. *Software Eng. Notes*, 23(2):73–81, 1998.

[14] N. Tracey, J. Clark, and K. Mander. The way forward for unifying dynamic test-case generation: The optimisation-based approach. In *Proc. IFIP Int'l Workshop Dependable Computing and Its Applications (DCIA)*, pages 169–180, 1998.

[15] N. J. Tracey. *A Search-Based Automated Test-Data Generation Framework for Safety-Critical Software*. PhD thesis, Dept. Computer Science, University of York, 2000.

[16] A. Vargha and H. Delaney. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *J. Educational and Behavioral Statistics*, 25(2):101–132, 2000.

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

[18] F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, 1945.

APPENDIX

binary_search

```
% Based on original function written by Aroh Barjaty
function index = binarysearch(x,var)
x=sort(x);
x;
var;
xLen = length(x);
[xRow, xCol] = size(x);
if x(1) > x(xLen)
    if xRow==1
        x = fliplr(x);
    else
        x = flipud(x);
    end
    flipped = 1;
else
  if x(1) < x(xLen)
    flipped = 0;
  else
    return;
  end
end
for i = 1:length(var)
    low = 1;
    high = xLen;
    if var(i) <= x(low)
        index(i) = low;
        continue;
    else
      if var(i) >= x(high)
        index(i) = high;
        continue;
      end
    end
    flag = 0;
    while (low <= high)
        mid = round((low + high)/2);
        if (var(i) < x(mid))
            high = mid;
        else
          if (var(i) > x(mid))
            low = mid;
          else
            index(i) = mid;
            flag = 1;
            break;
          end
    end
        if (low == (high - 1))
            break;
        end
    end
    if (flag == 1)
        continue;
    end
    if (low == high)
        index(i) = low;
    else
     if ((x(low) - var(i))^2 > (x(high) - var(i))^2)
        index(i) = high;
     else
        index(i) = low;
     end
    end
end
if flipped
    index = xLen - index + 1;
end
```

matrix_ops

```
function matrix_ops(A,B,C,D,E,F)
   M=[12.1,9.4;14.5,8.9];
   N=[4.0,18.2;5.2,3.7];
   if(A>=M)
     if(B~=N)
         if(C>magic(2))
            if(D<=14)
              if(E<N)
                if(F>=eye(2))
                   disp('Success');
                end
              end
            end
         end
      end
   end
```

is_singular

```
function r=is_singular(A)
  if (det(A) == 0)
    r = true;
  else
    r = false;
  end
```

is_rank_deficient

```
function r=is_rank_deficient(A)
  if (rank(A) < size(A,1))
    r=true;
  else
    r=false;
  end
```