Automated Software Engineering manuscript No.
(will be inserted by the editor)

Analysing the Fitness Landscape of Search-Based Software Testing Problems

Aldeida Aleti · I. Moser · Lars Grunske

the date of receipt and acceptance should be inserted later

Abstract Search-based software testing automatically derives test inputs for a software system with the goal of improving various criteria, such as branch coverage. In many cases, evolutionary algorithms are implemented to find near-optimal test suites for software systems. The result of the search is usually received without any indication of how successful the search has been. Fitness landscape characterisation can help understand the search process and its probability of success. In this study, we recorded the information content, negative slope coefficient and the number of improvements during the progress of a genetic algorithm within the EvoSuite framework. Correlating the metrics with the branch and method coverages and the fitness function values reveals that the problem formulation used in EvoSuite could be improved by revising the objective function. It also demonstrates that given the current formulation, the use of crossover has no benefits for the search as the most problematic landscape features are not the number of local optima but the presence of many plateaus.

Keywords Test data generation · genetic algorithms · fitness landscape characterisation

1 Introduction

The main aim of software testing is to detect as many faults as possible. Ideally, to gain sufficient confidence that the software system is reliable and no faults exist, an exhaustive testing technique should be carried out. However, software systems are becoming larger and more complex, which makes exhaustive search impracticable. Instead, testers resort to search-based methods, which use some criteria, such as branch or method coverage to measure the effectiveness of a test case, and optimise it (Aleti and Grunske, 2015; Ali et al., 2010a; Fraser and Arcuri, 2013; Kirkpatrick et al., 1983; Miller and Spooner, 1976).

A variety of search-based methods have successfully been applied to software testing, such as genetic algorithms (Fraser and Arcuri, 2013; Aleti and Grunske, 2015; Ali et al., 2010a), simulated annealing (Kirkpatrick et al., 1983) and local search (Miller and Spooner, 1976). Many approaches consider experimental studies to determine the success of the optimisation strategy based on a set of selected problem instances (Ali et al., 2010b). In some cases, the investigator implements several optimisers and compares the results to find the most suitable approach. However, this does not detect problems in the implementation details. More insightful conclusions can be drawn from the optimisation process, which emphasises the need to explore the relationship between key characteristics of optimisation problem instances and algorithm behaviour (Smith-Miles and Lopes, 2012).

Aldeida Aleti
Faculty of Information Technology, Monash University, Australia
E-mail: aldeida.aleti@monash.edu

Irene Moser
Faculty of Science, Engineering & Technology, Swinburne University of Technology, Australia
E-mail: imoser@swin.edu.au

Lars Grunske
Institute of Software Technology, University of Stuttgart, Germany
E-mail: lars.grunske@informatik.uni-stuttgart.de

Using the concept of fitness landscape, it is possible to study the structure of a search space and analyse features that make problems hard to solve (Gheorghita et al., 2013a,b). The fitness landscapes of different optimisation problems contain features that may affect the suitability of an optimisation algorithm. Examples of features are the number, distribution and size of the optima, the location of the global optimum, and plateaus. Fitness landscape characterisation also allows the study of the dynamics of the evolution of solutions, the comparative effectiveness of search methods and the ability of the algorithms to find good solutions to a given problem.

In particular, the fitness landscape of structural testing contains many plateaus (McMinn, 2004) due to known problems involving flag and enumeration variables, unstructured control flows, and state behaviour. Other unknown reasons may affect the success of search-based methods in finding test data for structural testing. Ideally, one should learn from the specific fitness landscape characteristics to understand what makes certain test cases hard to generate. An understanding of the fitness landscape features, and the relationship between features and success of optimisation methods can improve the design of algorithms that may be used in the future to generate test suites with search-based methods. In addition, details gained from the fitness landscape can also be used at run time to tune the parameters of the algorithm to get the best possible performance.

In this paper, we investigate the fitness landscape of software tests generated with a structural testing technique for various open source programs and libraries. We use fitness landscape characterisation metrics, which learn features of the fitness landscape from the information acquired during the optimisation process, such as the sequence of fitnesses of the best solutions during the iterations. Fitness landscape characterisation provides insights on what makes test data generation difficult for crossover and mutation, the two main operators of a genetic algorithm. Specifically, we use the *Change Rate* (CR), the *Delta Change* (DC), the *Negative Slope Coefficient* (NSC) (Vanneschi et al., 2006) and the *Population Information Content* (PIC) (an adaptation of information content (Vassilev et al., 2000)) as fitness landscape characterisation metrics. In the experimental study, we use 19 open source libraries and programs and generate test suites with evolutionary algorithms. The EvoSuite tool (Fraser and Arcuri, 2013) was used to generate test suites. We analyse the properties of the resulting test suites by measuring the relationship between problem features, fitness landscape characterisation metrics and algorithm performance.

2 Background and Related Work

For large software systems, the manual generation of test inputs is a costly and difficult task. This has motivated the automation of test data generation with various techniques. Exhaustive methods, such as the complete enumeration of test inputs is usually infeasible for large programs, whereas random methods may ignore features of software that are not exercised by only chance. The main difficulties with test data generation are the size and complexity of the software, and the fact that often this problem is undecidable (McMinn, 2004).

A wide range of approaches exist for automated test case generation. A number of approaches surveyed by (Utting et al., 2012) and Hierons et al. (2009) derive test cases from models. Other approaches produce test cases experimentally based on input source code (Ali et al., 2010b). The assessment of the quality of test suites also varies, with many approaches using coverage criteria (Ali et al., 2010b). To deal with the size of the software, many test data generation techniques apply search methods, such as genetic algorithms (Fraser and Arcuri, 2013; Aleti and Grunske, 2015; Ali et al., 2010a; Kalboussi et al., 2013), simulated annealing (Kirkpatrick et al., 1983; Matinnejad et al., 2015), memetic algorithm (Fraser et al., 2015) , ant colony optimisation (Mao et al., 2015) and local search (Miller and Spooner, 1976). *Genetic algorithms* (GAs) are the most frequently used optimisation technique for test data generation (Ali et al., 2010a), applied for the first time to generate structural test data by Xanthakis et al. (1992). There are three main areas in test data generation where search methods have proven to be successful:

- Coverage optimisation in white-box/structural testing;
- Verification of non-functional properties, such as worst case performance of a code segment;
- Disproving grey-box operation properties, such as the simulation of error conditions in a piece of software.

White-box testing, also known as structural testing generates tests using the internal structure of the software system. Instead of manually coding entire test suites, a developer can automatically generate test inputs that systematically test a program for different coverage criteria. Many approaches in white-box

testing use the *Control Flow Graph* (CFG) of the software system. The CFG is a directed graph, where nodes represent statements, and edges model the transfer of control from one node to another. Nodes that correspond to a decision statement, such as 'if' or 'while' statements, are known as *branching nodes*, and outgoing edges from these nodes are *branches*. A *branch predicate* is the condition upon which a branch is taken.

The set of n *input variables* to a program is the input vector $I = (i_1, i_2, ..., i_n)$, where $1 \leq i \leq k$ is the domain of input variable i_j. The *domain* of a program is the cross product of the domains of all input variables. The aim of search-based testing is to find a *program input* in this k-dimensional space that optimises a certain criterion. Clearly, this is a large combinatorial optimisation problem.

A sequence of neighbouring nodes in a CFG is a *path*. A path is considered feasible if there exists a program input for which the path is traversed. Symbolic execution is a white-box testing technique that solves path constraints to generate test data (Williams et al., 2005; Clarke, 1976; Boyer et al., 1975). These methods do not execute a program. Instead, they execute the process of assigning expressions to program variables while a path is followed through the code structure. Dynamic symbolic execution addresses problems in symbolic execution by incorporating concrete executions. The success of this method has been shown in many industrial and academical tools for different programming languages (Godefroid et al., 2005; Sen et al., 2005; Tillmann and De Halleux, 2008). However, challenges with infinite path and complex constraints have to be acknowledged. As an alternative to using CFG, Vivanti et al. (2013) have also provided an approach to use *Data Flow Graphs*(DFG) for automatic test case generation.

The literature in structural test data generation has focused mainly on generating inputs for specific paths, or program structures such as branches and statements. The main approaches can be classified based on the fitness function optimised as follows:

- *Coverage:* These approaches are based on the covered program structure (Roper, 1997), and the aim is to obtain full program coverage (Watkins, 1995). In the work by Watkins (1995), the fitness of solutions that follow already covered paths is penalised with the inverse of the number of times the path has been previously executed. This approach requires fewer tests to achieve path coverage compared to random testing. Optimisation approaches that use coverage as a fitness function suffer from the lack of guidance towards unexplored structures due to the small sample of the input domain. It is usually the case that full coverage is not achieved for large software systems (McMinn, 2004).
- *Structure:* The objective function used in structure-based approaches focuses either on branch-distance (Xanthakis et al., 1992; Michael et al., 2001; Jones et al., 1996), by exploiting information from branch predicates, control structures (Jones et al., 1996; Pargas et al., 1999), or both (Wegener et al., 2001). Using only control information during the optimisation process creates fitness landscapes with plateaus that are hard to search, since there are no gradients leading to local/global optima. Combining control information with branch distance information may help prevent the formation of a plateaux in the fitness landscape (McMinn, 2004).

3 EvoSuite - An Automated Test Suite Generation Technique

In this paper, we consider white-box testing techniques, which require no specifications, although if specifications exist, they may be used in the generation of test cases. EvoSuite (Fraser and Arcuri, 2013) is a prominent automated white-box testing tool that uses genetic algorithms. EvoSuite generates JUnit test suites, which are optimised to cover as many branches as possible. Test suites are a set of test cases that contain assertions that describe the behaviour of the software system. A software test consists of two main components, an input to the executable program and a definition of the expected outcome.

To automate this process, EvoSuite uses a genetic algorithm that evolves a population of randomly initialised test suites using mutation and crossover. Let t denote a test case, composed of a sequence of statements $t = \langle s_1, s_2, ..., s_l \rangle$ of length l. A statement s_i can be a constructor, a field, a primitive, a method or an assignment. A solution is defined as a collection of $T = \{t_1, t_2, ..., t_n\}$ of test cases. An optimal solution T^* is a test suite that covers all possible branches.

The fitness of a solution is based on the *code coverage* criterion, where the aim is to cover as many control structures, such as if and $while$ statements, as possible, by evaluating the logical predicates that result in either true and false. Formally, the branch coverage metric is defined as

$$f(T) = |M| - |M_T| + \sum_{b \in B} d(b, T), \tag{1}$$

where $|M|$ is the total number of methods, $|M_T|$ is the number of executed methods in test suite T and B is the set of branches in the program. The branch distance $d(b, T)$ is evaluated as

$$d(b,T) = \begin{cases} 0 & \text{if the branch has been covered,} \\ \frac{(x_{tt}-x)+(x_{tf}-x)}{((x_{tt}-x)+(x_{tf}-x))+1} & \text{if the predicate has been executed at least twice,} \\ 1 & \text{otherwise.} \end{cases} \quad (2)$$

The distance $d(b, T)$ is 0 if at least one of the branches has been covered, and > 0 otherwise. x_{tt} represents the threshold of the true branch whereas x_{tf} is the threshold of the false branch. x is the actual value of the variable. If the variable has a value that evaluates to true, $(x_{tt} - x)$ will evaluate to zero and $(x_{tf} - x)$ to a nonzero value. The outcome is normalised to achieve a value between zero and unity. The genetic algorithm aims at minimising the function in Eq. 1. The length of the test suite is measured as the sum of the lengths of all test cases that are part of it, formally defined as

$$|T| = \sum_{t \in T} |t|. \quad (3)$$

An upper bound on the length of the test suite ensures that the algorithm does not produce unreasonably large test suites. The solution representation in the genetic algorithm is of a variable size. The crossover operator creates two new solutions T_1' and T_2' by combining test cases from two pre-existing test suites T_1 and T_2. A parameter α that is randomly selected in the range $[0, 1]$ indicates how the solutions are combined: one child T_1' receives $\lceil \alpha |T_1| \rceil$ test cases from T_1 and the remainder from T_2, while the second child receives the first $\lceil \alpha |T_2| \rceil$ test cases from the second parent and the remainder from the first parent. Most often, this will lead to parents and children having all different lengths. The parents are selected using a ranked-based procedure (Whitley, 1989), where solutions are ranked based on their fitness (Eq. 1). Solutions are then assigned a selection probability based on their rank. When there is a tie between solutions, shorter test suites are assigned better ranks.

The solutions produced by the crossover operator undergo the mutation operators. Test suites may be mutated by changing each test case with a probability $1/n$, with n denoting the number of test cases in a suite, which leads to one test case being changed on average per suite. The change can be the removal, insertion or change of a statement in the test case with equal probability. Each statement has a $1/l$ probability of undergoing one of these changes, with l denoting the number of statements in the test case. The change mutation of test cases is followed by another type of mutation which adds new test cases t_i with a probability of σ^i, leading to an exponentially decreasing probability of adding more test cases. The set of solutions created by crossover and mutation operators are added to the population of solutions kept by the genetic algorithm. Similar to the original studies with EvoSuite, an elitist strategy is used as a replacement procedure (Fraser and Arcuri, 2013).

4 Fitness Landscape Characterisation

The suitability of a search method for solving an optimisation problem instance depends on the structure of the fitness landscape of that instance. A fitness landscape in the context of combinatorial optimisation problems refers to (i) the search space S, composed of all possible solutions that are connected through (ii) the search operator, which assigns each solution $s \in S$ to a set of neighbours $N(s) \subset S$, and the fitness function $F : S \rightarrow \Re$. As the neighbourhood of a solution depends on the search operator, a given problem can have any number of fitness landscapes. The neighbourhoods can be very large, such as the ones arising from the crossover operator of a genetic algorithm, while a 2-opt operator of a permutation problem has a neighbourhood that is relatively limited in size.

The fitness landscape of an optimisation problem is composed of certain features that may affect the suitability of an optimisation algorithm. Examples of features are the number and distribution as well as the sizes of the optima, the location of the global optimum, and plateaus. A fitness landscape with a single optimum is easy to search with a deterministic hill climbing method. On the other extreme, the existence of many local optima and an isolated global optimum makes a fitness landscape rugged and hard to explore. A plateau symbolises the presence of neighbouring solutions with equal fitness, where the progress of a search algorithm potentially stagnates. Plateaus can be detrimental to search algorithms such as local search and steepest descent approaches, which depend on gradients in the fitness landscape

they can follow. Depending on the landscape the algorithm produces given its operators, the search may be harder or easier. Identifying the features that arise from the combination of problem and algorithm can help determine the suitability of an algorithm over another.

The autocorrelation function and the correlation length are two seminal metrics (Weinberger, 1990, 1991), which estimate the relationship between the quality of solutions of a random walk. Assuming a statistically isotropic landscape, where, on average, the fitness landscape has the same topological features in all directions, this measure can be used as an indicator of difficulty for certain problems (Merz and Freisleben, 2000). The correlation length measures the decay of the autocorrelation function and the ruggedness of the fitness landscape, where values closer to 1 indicate a stronger correlation and values toward zero indicate the absence of a correlation (Stadler, 1996). Angel and Zissimopoulos (1998) observed a strong connection between correlation length and the hardness of an optimisation problem for a local search algorithm, suggesting that the number of local minima depends on the relationship between the fitness of a solution and the fitness of its neighbours.

The *Cumulative Probability of Success* (CPS) is employed as a measure of how difficult a problem instance is for certain search strategies (Kinnear, 1994). The CPS index represents the probability that the search strategy finds a solution to a problem in a given number of iterations. It is estimated as the number of feasible solutions divided by the total number of individuals created in n iterations. In his evaluation of CPS, Kinnear (1994) applied the fitness autocorrelation analysis technique by Weinberger (1990). The fitness autocorrelation was computed for several individuals and for several time-steps, and the overall average of fitness autocorrelations was presented as a measure of the fitness landscape difficulty. The lower the autocorrelation, the harder the problem is for that particular search strategy. However, Kinnear (1994) notes that there is practically no correlation between the CPS index and the fitness autocorrelation. Fitness landscapes with very low fitness autocorrelation coefficients, which would be treated as difficult by Kinnear's approach (Kinnear, 1994), had a high cumulative probability of success, i.e. the search strategy was successful in finding feasible solutions.

4.1 Population Information Content

The information characteristics of fitness landscapes have been researched by (Vassilev et al., 2000). In simplified terms, the approach records the steps of a random walk in a search landscape as a time series of -1 (deteriorating fitness), 1 (improving fitness) and 0 (no change). The information content is calculated from this time series as shown in Eq. 4.

$$H = -\sum_{i \neq j} P_{ij} \log_6 P_{ij} \tag{4}$$

The equation is based on the notion of Shannon's entropy (Shannon and Weaver, 1963); the distribution of features over a system. It disregards the 'informationless' sequences where the state does not change. The lower H, the higher the information content as fewer and shorter contiguous strings of identical values are included.

The mapping proposed by Vassilev naturally lends itself for random walks. To perform a similar landscape analysis during a GA-based search, possible alternatives are to follow the fitness development of a single solution or that of the entire population. Following a single solution is problematic because of the presence of the crossover and selection operators. To avoid this problem, we chose to follow the fitness development of the entire population (population-based walks) and naming the metric *Population Information Content* (PIC). The selection operator generally ensures that the fitness of the population does not decrease, but stagnates at worst before finding another incumbent best. As a result, these population-based walks create a bias towards fitter regions in the search space, however Pitzer and Affenzeller (2012) argue that population-based walks can be used to investigate the local features of the fitness landscape and are an alternative to random walks. To measure the population information content, we record the best fitness at each iteration $f*_i$, $i < n$ where n is the number of iterations. The sequence of best fitnesses $\{f_i^*\}_{i=0}^{n}$ is converted into a binary string $\{b_i^t\}_{i=0}^{n-1}$, where

$$b_i = \begin{cases} 0 & \text{if } f_i^* - f_{i-1}^* = 0 \\ 1 & \text{otherwise.} \end{cases} \tag{5}$$

Eq. 5 assigns 0 to b_i if there is no difference between the best fitness at iteration i (f_i^*) and the best fitness in the previous iteration (f_i^{t-1}). If in one iteration a GA does not find a better solution than the currently known optimum, but then finds a better solution in the next iteration, and an even better solution than the new incumbent best in the following iteration, while failing to produce an improvement of the incumbent best solution in the fourth iteration, a sequence of 0110 will be created. The sequence is recorded over the entire GA search.

The resulting binary string $\{b_i\}_{i=0}^{n-1}$ is divided into overlapping blocks of two bits. The example string 0110 yields three sub-strings: {01}, {11} and {10}. The probability of encountering a particular sub-string is given by the number of times the sub-string is found divided by the total number of sub-strings as shown in Eq. 6.

$$P_{\{b_i b_{i-1}\}} = \frac{|\{b_i b_{i-1}\}|}{n-1} \tag{6}$$

The average amount of information contained in the binary string $\{b_i\}_{i=0}^{n-1}$ is calculated as shown in Eq. 7.

$$H = -\sum_{i=1, b_i \neq b_{i-1}}^{n-1} P_{\{b_i b_{i-1}\}} log_2 P_{\{b_i b_{i-1}\}} \tag{7}$$

In a binary string, only two pairs with nonequal members $\{b_i b_{i-1}\}$ exist: {01} and {10}, with $P_{\{01\}}$ the probability of {01} sequences, hence Eq. 7 can be rewritten as $H = -P_{\{01\}} log_2 P_{\{01\}} - P_{\{10\}} log_2 P_{\{10\}}$. To deal with the indeterminate form $0 \times -\infty$, we apply L'Hôpital's rule. Observe that $x \log x = \frac{\log x}{1/x}$. Taking the limits, we obtain

$$\lim_{x \to 0} x \times \log x = \lim_{x \to 0} \frac{\log x}{1/x} = \lim_{x \to 0} \frac{1/x}{-1/x^2} = \lim_{x \to 0} -x = 0 \tag{8}$$

If a string has no changes, the H value approaches zero. The lower the value of H, the more irregular the fitness development. Using this measure on the fitness development of the incumbent best solution is based on the assumption that GAs that undergo improvements interspersed with non-improving iterations are more likely to explore different basins of attraction, while continued improvements suggest following a single gradient.

4.2 Negative Slope Coefficient

One of the most prominent fitness landscape characterisation metrics designed to estimate the degree of difficulty a problem poses to evolutionary algorithms is the *Negative Slope Coefficient* (NSC) (Vanneschi et al., 2006). It is based on the fitness difference between parent and offspring solutions. Given n parent solutions s_i, it uses the genetic operators (mutation and crossover) to sample the neighbourhood of each s_i. The fitness of each parent is viewed against the fitnesses of created solutions in a scatter plot. The scatter plot is know as the *fitness cloud*, which illustrates the correlation between current fitness and fitness after an operator is applied to the solution. To quantify the information contained in the scatter plot, the authors proposed the NSC (Vanneschi et al., 2006).

The fitness of the parents as the abscissas of the scatter plot is partitioned into m segments $\{I_1, I_2, ..., I_m\}$ of equal size, with the ordinates denoted as $\{J_1, J_2, ..., J_m\}$. The number of segments m is not fixed, and is decided based on the algorithm proposed by Vanneschi et al. (2006). The method starts by partitioning the fitness cloud into two segments with the same number of points. Next, the segment with the largest number of points is partitioned again into two sub-segments with an equal number of points. This iterative process goes on until either one of the segments contains less points than a threshold (in our case 50 points), or a segments is smaller than a predefined size (5% of the total length of the original segment). This method was shown to be reliable (Vanneschi et al., 2006).

Next, the averages of the abscissa values in each segment $\{M_1, M_2, ..., M_M\}$ and the averages of the ordinate values $\{N_1, N_2, ..., N_m\}$ are calculated. Each point (M_i, N_i) is connected to the point (M_{i+1}, N_{i+1}), creating a set of segments. For each segment connecting two points (M_i, N_i) and (M_{i+1}, N_{i+1}), the slope P_i is calculated as follows:

$$P_i = \frac{N_{i+1} - N_i}{M_{i+1} - M_i} \tag{9}$$

Finally, the negative slope coefficient is calculated as

$$\text{NSC} = \sum_{i=0}^{m-1} \min(P_i, 0). \tag{10}$$

The NSC considers only negative values, which means that if solutions produce better offspring, the NSC is zero and the fitness landscape is considered easy for a GA, whereas if the reverse is the case, the landscape is difficult. This is intuitive as the selection operator preserves better solutions as parents for the next generation. If better parents actually produce worse offspring, the GA is unlikely to be successful in finding optimal results. The magnitude of the NSC value quantifies the difficulty of the problem.

4.3 Change Rate

To measure the change rate, we record the best fitness at each iteration $f*_i$, $i < n$ where n is the number of iterations. The sequence of best fitnesses $\{f_i^*\}_{i=0}^{n}$ is converted to a binary string $\{b_i\}_{i=0}^{n-1}$ using Eq. 5. The value of b_i is 0 if there is no change in the best fitness at iteration i, and 1 otherwise. The change rate is calculated as:

$$\text{CR} = \frac{\sum_i^n b_i}{n}. \tag{11}$$

In words, the change rate is the number of iterations where an improvement was observed over the total number of iterations.

5 Experiments

The purpose of the experiments is to investigate the effectiveness of crossover and mutation operators in solving the test case generation problem. A genetic algorithm with only mutation (GA_M), and a genetic algorithm with both mutation and crossover (GA_MC) were implemented in EvoSuite (Fraser and Arcuri, 2013) and used to generate test suites for a set of open source libraries and programs. The objective function used considers both method and branch coverage (Eq. 1).

We analyse the properties of the resulting test suites by measuring the relationship between problem features and algorithm performance. We employ the fitness landscape characterisation metrics to shed light on the behaviour of the search strategies.

5.1 Experimental Settings

The crossover and mutation operators and their rates are probably the most prominent parameters in genetic algorithms (Bäck et al., 2000). The parameter values for the GA are based on recommendation from Fraser and Arcuri (2013), many of which are 'best practice'. The crossover rate is set to 0.75, the rate of inserting a new test case is 0.1, whereas the rate for inserting a new statement is 0.5. A fixed number of computations was used as a stopping criterion, equal to 1,000,000 function evaluations. The population size was set to 80. To handle the stochastic nature of the GA, each experiment was repeated 30 times with different seeds for the random number generator.

Branch coverage (Eq. 1) is used as an indicator of the performance of the optimisation algorithm. Branch coverage evaluates whether Boolean expressions tested in control structures (if-statement and while-statement) evaluate to either true and false. The number of branches and their ratio to the number of methods in a problem instance affects this metric, and is an indication of problem difficulty. Hence, we have selected a wide range of problems with different numbers of branches and branch/method ratios.

5.2 Problem Instances

To analyse the performance of the optimisation schemes, we selected 19 open source libraries and programs depicted in Table 1. To avoid any bias towards selecting problem instances that are in favour of the proposed method, the instances were selected uniformly at random from SourceForge, and can be downloaded from the websites listed in the third column. They vary in the lines of code, number of methods, fields, and classes. The second column in Table 1 shows the version or latest update of the programs used in the experiments.

Table 1: The set of open source libraries and programs.

Project name	LOC	M	F	C	**Version**	**Link**
TulliBee	1,929	246	374	19	2012-09-23	`sourceforge.net/projects/tullibee`
Amazon AWS 4 Java	21,961,264	522	175	45	V1.0.1	`a4j.sourceforge.net`
Remote Invocation Fr.	1,212	169	79	24	V1.0	`densebrain.com/rif/index.html`
DSA Chat Program	2,807	292	303	33	V1.2.0	`dsachat.sourceforge.net`
GreenCow	3	2	0	1	2010-04-10	`greencow.sourceforge.net`
J2EE PetStore	1,011	437	150	58	2013-03-08	`petsoar.sourceforge.net`
Jdbacl	8,926	1,385	1,207	165	Revision 711	`jdbacl.sourceforge.net`
Omjstate	164	59	16	13	2009-09-06	`omjstate.sourceforge.net`
Beanbin	1,895	484	127	79	V1.0	`beanbin.sourceforge.net`
Joomla Template	1,215	200	81	22	V2.0	`templatedetails.sourceforge.net`
Inspirento	1,404	392	112	41	2013-03-08	`sourceforge.net/projects/inspirento/`
Method Cohesion An.	7,936	794	384	47	2005-11-21	`jmca.sourceforge.net`
Nekomud	807	37	22	8	Revision 16	`nekomud.sourceforge.net`
Geo-google	2,213	1,234	572	108	Revision 113	`geo-google.sourceforge.net`
Byuic	1,123	72	34	9	Revision 1	`wiki.brilaps.com/wikka.php?wakka=byuic`
SaxPath	958	305	80	12	V1.1.6	`saxpath.org`
JNI-InChI	647	207	52	14	Revision 279	`jni-inchi.sourceforge.net`
Java Wiki Bot Fr.	2,816	655	305	84	V3.0.0	`jwbf.sourceforge.net`
Lilith	33,581	7,230	2,529	762	2014-05-20	`sourceforge.net/projects/lilith`

*F=fields, M=Methods, C = Classes

5.3 Results

In test data generation with EvoSuite, many classes are impossible to cover by test cases EvoSuite is able to generate (e.g. if there are environmental dependencies that EvoSuite cannot fulfil). In our experimental evaluation, 13% of the classes fell into this category, with search showing no improvement. These cases were not considered in the analysis of the results.

The difficulty an instance poses for a particular algorithm is typically measured by comparing the quality achieved after a certain number of iterations compared to other algorithms (Xin et al., 2009). For the purpose of these experiments, the fitness function described in Eq. 1 was used to assess the quality of the solutions, and the suitability of the two optimisation schemes: a genetic algorithm with only a mutation operator (GA_M), and a genetic algorithm with both mutation and crossover (GA_{MC}). The fitness improvement shown in Fig. 1 is measured as the difference between the best fitness in the first iteration and the best fitness in the final iteration, averaged over 30 trials.

The crossover operator combines test cases from different test suites, mixing the 'genetic' information from the two parents. The new solutions may look very different from the parents; a test suite that takes half of the test cases from each parent shares only 50% of its genetic composition with them. The mutation operator, on the other hand, introduces new information by changing a test case by either adding, altering or removing statements, or adding a new test case in a test suite. The resulting solution does not differ greatly from the parent solution, however, the improvement in fitness may be quite large. For instance, consider the hypothetical case of a program with 3 of 4 branches covered by a test suite.

Adding a single new statement as a test input which executes the uncovered branch increases the branch coverage from 75% to 100%; i.e. around 33% improvement on the original solution.

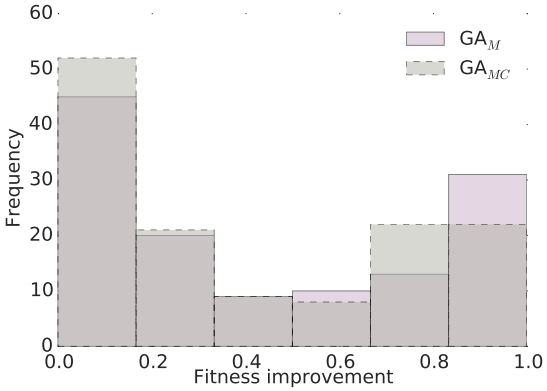
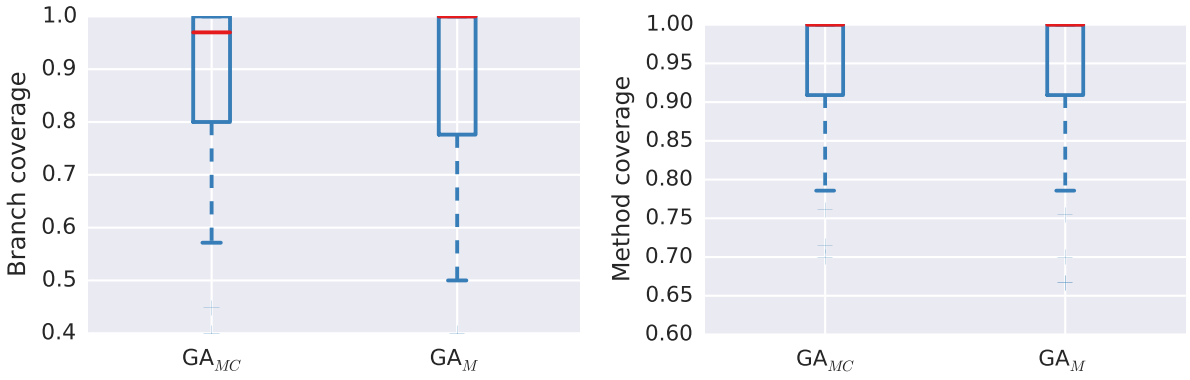


Fig. 1: The histograms of fitness improvement for genetic algorithm with only a mutation operator (GA_M), and a genetic algorithm with both mutation and crossover (GA_{MC}). The dark colour represents the common parts between both algorithms. The values are normalised over the 19 projects.

Fig. 1 shows that GA_M produces larger fitness improvements compared to GA_{MC}. In optimisation, crossover is often useful as a means of making large jumps in the search space when mutation steps cannot escape the attraction of a local optimum. Entrapment in local optima is a phenomenon in combinatorial optimisation where certain choices are incompatible with other, inherently beneficial choices in that they cause these choices to decrease the overall fitness of a solution. In terms of test suite generation, entrapment in a local optimum is only conceivable if an existing test case conflicts with a desirable test case to the effect that the fitness of the suite deteriorates if both test cases are present, rendering the incremental improvement of the solution impossible - adding the desirable test case worsens the quality, and the solution is likely to be removed during selection.

In the current problems, adding another test case that does not improve the coverage metrics causes a plateau in the landscape. The fitness stays the same, since there is no penalty for the length of the suite. Hence, mutations are able to improve the fitness incrementally, and there is no benefit in using crossover.



(a) Branch coverage.

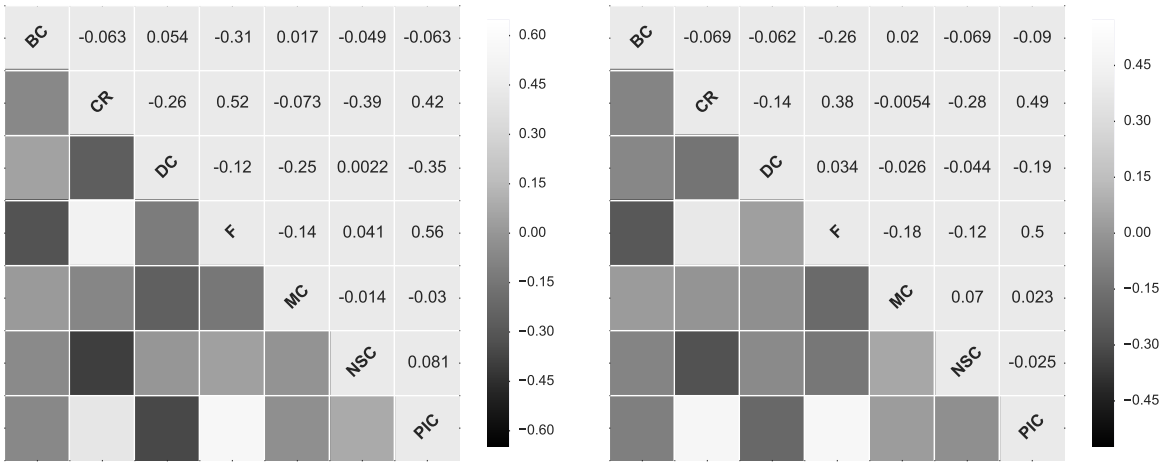
(b) Method coverage.

Fig. 2: Boxplots of branch and method coverage for all problem instances and the two optimisation schemes.

However, the difference in the performance of the two implementations investigated in this study is not very large, especially when compared in terms of method coverage (calculated as the number of methods

covered, divided by the number of total methods), as shown in Fig. 2. Nevertheless, GA_M achieves better branch coverage (Fig. 2a).

To investigate the behaviour of the two optimisation schemes, we analysed the additional information provided by the metrics applied during the search: the PIC, described in Eq. 7, the negative slope coefficient, defined in Eq. 10, and the change rate, given in Eq. 11. To investigate the benefit of the metrics, we correlated them with the final fitness and the method and branch coverage criteria. The solution fitness (Eq. 1) is designed to incorporate both the branch and method coverage criteria. Since the fitness function combines both criteria, it is meaningful to investigate how the optimisation result and the metrics correlate with the intended goals as well as the formulation of the goals. The correlation coefficients are depicted in Fig. 3.



(a) GA_{MC} (b) GA_M

Fig. 3: Correlations between fitness (F) and fitness landscape characterisation metrics Branch Coverage (BC), Change Rate (CR), Method Coverage (MC), Delta Change (DC), Negative Slope Coefficient (NSC) and Population Information Content (PIC).

The medium positive correlation between solution fitness (F) and PIC indicates that given the current problem, it seems to be detrimental for the outcome if the GA improvements alternate with non-improving iterations. High PIC values indicate that the population has many improving iterations which are flanked by iterations that do not improve on the incumbent best solution. The positive correlation with the fitness indicates that the fitness deteriorates with the number of improving moves.

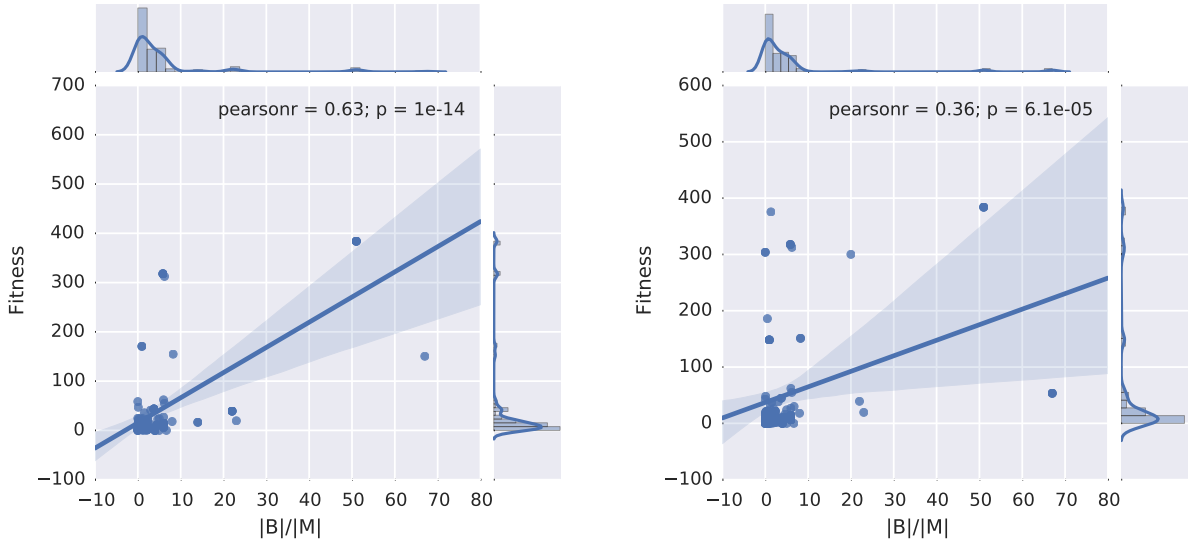
This counterintuitive observation is confirmed by the change rate, which records the number of improvements over the iterations. In the case of GA_{MC}, the correlation with fitness is medium and almost of equal strength as the correlation between PIC and fitness. In the case of GA_M, the correlation between fitness and CR is lower than that of fitness and PIC, suggesting that successive improvements are less detrimental to quality than improvements interspersed with periods of stagnation. Fig. 1 illustrates that the majority of the improvements in the test suite generation problem are quite small in scale. A few very good moves improve the fitness by almost half. The medium positive correlation between PIC and CR shows that when the improvements are more numerous, many of them are flanked by non-improving iterations. The correlation is similar for GA_{MC} and GA_M.

To verify the notion that larger improvements are more beneficial in this problem than numerous small ones, we recorded the average differences made in each improvement and divided it by the number of changes made over the iterations. Interestingly, the correlation between DC on the one hand and PIC/CR on the other are very weak in the case of GA_M but almost medium with PIC when considering GA_{MC}.

When crossover is used, larger improvements flanked by stagnation appears to be the predominant way of finding better quality solutions.

While DC hardly correlates with the fitness, it has a weak but not negligible correlation with MC (but not BC) in the case of GA_{MC}, which suggests that the large improvements made by the crossover benefit the method coverage more than the branch coverage.

The negative slope coefficient (NSC) hardly correlates with the fitness at all. Since NSC only records negative fitness developments, the best possible NSC is zero. The results so far have suggested that the landscape formed by the problem and the GA implementations has few improvements and many plateaus. In such a landscape, the NSC is not able to make significant distinctions between the instances.



(a) GA_{MC} (b) GA_{M}

Fig. 4: Correlations between properties of the problem and fitness achieved by the search. |B|/|M| is defined as the proportion of branches to methods.

A somewhat disconcerting observation is the fact that the correlation between the fitness function and the properties it is intended to quantify, the method and branch coverages, is rather weak. The correlation between method coverage and fitness is negligible, whereas there exists at least a weak correlation between fitness and branch coverage, which is slightly higher if the crossover operator is used. It is expected that crossover leads to more deteriorations in fitness, whereas mutation alone is likely to maintain equal fitness between iterations. In case the fitness deteriorates, the fitness function is in a better position to guide the search than when plateaus are encountered.

Fig. 4 suggests that there is a strong correlation between the ratio of branches to methods when crossover is used. When there are few methods per branch, more branches have to be covered to achieve a good fitness. Since the fitness is to be minimised, the stronger correlation with GA_{MC} indicates that such problems are harder to solve when crossover is used.

5.4 Threats to validity

This section discusses the threats to internal, external and construct validity that could affect the results obtained from the experimental evaluation.

Threats to *internal validity* result from the experimental setup as described in Section 5.1. This setup is in line with the previous studies in the area of search-based software testing using EvoSuite. We did not perform any parameter tuning before the experiments and selected standard parameters. Our choice not to perform any parameter tuning is influenced by the experimental evaluation of Arcuri and Fraser (2011,

2013), which specifically focusses on search-based software testing with EvoSuite. The conclusion of the experimental evaluation on parameter tuning is that the parameters have an effect on the performance of the genetic algorithm, however, the default values from the literature are sufficiently good to run meaningful experiments.
Threats to *external validity* relate to the generalizability of the results. The main threat is the relatively low number of 19 example systems that are used to perform the experiments on. However, to avoid any bias we randomly selected these programs. To improve the generalizability, the study should be extended to additional example programs. To further reduce the threats to external validity also closed source programs in an industrial setting similar to Matinnejad et al. (2015) should be used to investigate the search space for the automatic test case generation problem.
Threats to *construct validity* can be identified in two metric categories, namely the metrics that measure the quality of the generated test suites and metrics that provide information about the fitness landscape. In the first category, our study shares the same weaknesses as other approaches in the area of search-based software testing (Ali et al., 2010b), since we re-used the standard fitness function taken from Fraser and Arcuri (2013). The basic question is, if metrics like the code and branch coverage are good indicators to evaluate the quality of a test suite.

The second category, is concerned with metrics that evaluate characteristics of the fitness landscape. Since this is the main focus of this study, we took special care and selected multiple metrics to better understand the problem. Consequently, based on this diversity of metrics we reduce the potential threats to construct validity. The selection of the metrics, namely the change rate, delta change, population information content, negative slope coefficient and the number of improvements during the evolution progress, is consistent with the state of the art in the area of fitness landscape characterisation (Pitzer and Affenzeller, 2012; Smith et al., 2002). For the individual metrics, we used the Population Information Content (PIC) as a GA-specific alternative for the Information Content (IC). The PIC metric is influenced by the decision made during the search process, as high quality solutions survive from one iteration to the next. As a result, the sampling process no longer constitutes a random walk as in information content calculations. Instead the PIC uses a population-based walk that favours fitter regions of the fitness landscape. Pitzer and Affenzeller (2012) acknowledge that based on the selective sampling of population-based walks, the results of the PIC metric cannot be generalized to the entire fitness landscape, however PIC can still be used as an evolvability metric, which is exactly intended by our study since we aim to measure the chances of the population to improve.

6 Conclusion

In this article, we performed a descriptive study that analyses the fitness landscape of search-based software testing problems. This study used 19 programs and the EvoSuite tool (Fraser and Arcuri, 2013) to shed some light on the relationship between problem features, fitness landscape characterisation metrics and the specific algorithm performance. The main outcomes of this paper are:

1. Due to the nature of the objective function formulation given in eq. 1 the search space is rather poor in gradients to local optima, which renders the use of crossover operators as a means of escaping local optima rather useless. This is contraire to the usual behavior of crossover operators. Our observation is especially supported by the values of the NSC that suggest that we have a search space that is unimodal or has very few modes and large benches of equal fitness.
2. The search tends to achieve better results if it makes improvements contiguously, rather than when improvements are flanked by stagnation. This confirms the assumption that better optima are found at the end of longer gradients.
3. The correlation between the fitness and the change rate indicates that fewer improvements over the iterations lead to better quality solutions.
4. The ultimate goal of the optimisation problem is to optimise branch and method coverage. Both feature as factors in the objective function, which, however, shows little correlation with the method coverage and only a weak correlation with the branch coverage. This suggests that the reformulation of the objective function might have a great potential for improving the optimisation process and possibly the quality of the outcome.

In the future, the investigation of the fitness landscapes created by other formulations of coverage criteria is a priority. For large software systems, the mapping of the test inputs to the objective function

is computationally expensive, and in most cases infeasible. In flat landscapes of many plateaus, the search has little guidance and many iterations are wasted on non-improving changes. One of the priorities is to reformulate the fitness function to avoid this and reapply landscape characterisation metrics to ensure the search progresses as efficiently as possible. In future work, we will investigate a stopping criterion based on statistical significance tests to decide when the sampling should stop.

Acknowledgements

We would like to thank Gordon Fraser and his team who developed EvoSuite for making the source code available. We wish to acknowledge Monash University for the use of their Nimrod software in this work. The Nimrod project has been funded by the Australian Research Council and a number of Australian Government agencies, and was initially developed by the Distributed Systems Technology.

Funding

This research was supported under Australian Research Council's Discovery Projects funding scheme (project number DE140100017).

Conflict of Interest

The authors declare that they have no conflict of interest.

Ethical Statement

This research does not involve human participants and animals.

References

Aleti, A. and Grunske, L. (2015). Test data generation with a kalman filter-based adaptive genetic algorithm. *Journal of Systems and Software*, 103:343–352.

Ali, S., Briand, L., Hemmati, H., and Panesar-Walawege, R. (2010a). A systematic review of the application and empirical investigation of search-based test case generation. *Software Engineering, IEEE Transactions on*, 36(6):742–762.

Ali, S., Briand, L. C., Hemmati, H., and Panesar-Walawege, R. K. (2010b). A systematic review of the application and empirical investigation of search-based test case generation. *IEEE Trans. Software Eng.*, 36(6):742–762.

Angel, E. and Zissimopoulos, V. (1998). Autocorrelation coefficient for the graph bipartitioning problem. *Theoretical Computer Science*, 191:229–243.

Arcuri, A. and Fraser, G. (2011). On parameter tuning in search based software engineering. In *SSBSE*, pages 33–47.

Arcuri, A. and Fraser, G. (2013). Parameter tuning or default values? An empirical investigation in search-based software engineering. *Empirical Software Engineering*, 18(3):594–623.

Bäck, T., Eiben, A. E., and van der Vaart, N. A. L. (2000). An empirical study on GAs without parameters. In *Parallel Problem Solving from Nature – PPSN VI (6th PPSN'2000)*, volume 1917 of *Lecture Notes in Computer Science (LNCS)*, pages 315–324. Springer-Verlag (New York).

Boyer, R. S., Elspas, B., and Levitt, K. N. (1975). Select - a formal system for testing and debugging programs by symbolic execution. *ACM SigPlan Notices*, 10(6):234–245.

Clarke, L. A. (1976). A system to generate test data and symbolically execute programs. *Software Engineering, IEEE Transactions on*, (3):215–222.

Fraser, G. and Arcuri, A. (2013). Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291.

Fraser, G., Arcuri, A., and McMinn, P. (2015). A memetic algorithm for whole test suite generation. *Journal of Systems and Software*, 103:311–327.

Gheorghita, M., Moser, I., and Aleti, A. (2013a). Characterising fitness landscapes using predictive local search. In *Proceedings of the 15th Annual Conference Companion on Genetic and Evolutionary Computation*, pages 67–68. ACM.

Gheorghita, M., Moser, I., and Aleti, A. (2013b). Designing and characterising fitness landscapes with various operators. In *Evolutionary Computation (CEC), 2013 IEEE Congress on*, pages 2766–2772.

Godefroid, P., Klarlund, N., and Sen, K. (2005). Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223. ACM.

Hierons, R. M., Bogdanov, K., Bowen, J. P., Cleaveland, R., Derrick, J., Dick, J., Gheorghe, M., Harman, M., Kapoor, K., Krause, P. J., Lüttgen, G., Simons, A. J. H., Vilkomir, S. A., Woodward, M. R., and Zedan, H. (2009). Using formal specifications to support testing. *ACM Comput. Surv.*, 41(2).

Jones, B. F., Sthamer, H.-H., and Eyres, D. E. (1996). Automatic structural testing using genetic algorithms. *Software Engineering Journal*, 11(5):299–306.

Kalboussi, S., Bechikh, S., Kessentini, M., and Said, L. B. (2013). Preference-based many-objective evolutionary testing generates harder test cases for autonomous agents. In Ruhe, G. and Zhang, Y., editors, *Search Based Software Engineering - 5th International Symposium, SSBSE 2013*, volume 8084 of *Lecture Notes in Computer Science*, pages 245–250. Springer.

Kinnear, K. E. (1994). Fitness landscapes and difficulty in genetic programming. In *IEEE Conference on Evolutionary Computation*, volume 1, pages 142 – 147. IEEE.

Kirkpatrick, S., Vecchi, M., et al. (1983). Optimization by simmulated annealing. *Science*, 220(4598):671–680.

Mao, C., Xiao, L., Yu, X., and Chen, J. (2015). Adapting ant colony optimization to generate test data for software structural testing. *Swarm and Evolutionary Computation*, 20:23–36.

Matinnejad, R., Nejati, S., Briand, L. C., Bruckmann, T., and Poull, C. (2015). Search-based automated testing of continuous controllers: Framework, tool support, and case studies. *Information & Software Technology*, 57:705–722.

McMinn, P. (2004). Search-based software test data generation: a survey. *Softw. Test, Verif. Reliab*, 14(2):105–156.

Merz, P. and Freisleben, B. (2000). Fitness landscape analysis and memetic algorithms for the quadratic assignment problem. *IEEE Evolutionary Computation*, 4(4):337–352.

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

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

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

Pitzer, E. and Affenzeller, M. (2012). A comprehensive survey on fitness landscape analysis. In Fodor, J. C., Klempous, R., and Araujo, C. P. S., editors, *Recent Advances in Intelligent Engineering Systems*, volume 378 of *Studies in Computational Intelligence*, pages 161–191. Springer.

Roper, M. (1997). Computer aided software testing using genetic algorithms. *10th International Quality Week*.

Sen, K., Marinov, D., and Agha, G. (2005). Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA. ACM.

Shannon, C. and Weaver, W. (1963). *The Mathematical Theory of Communication*. Number pt. 11 in Illini books. University of Illinois Press.

Smith, T., Husbands, P., Layzell, P. J., and O'Shea, M. (2002). Fitness landscapes and evolvability. *Evolutionary Computation*, 10(1):1–34.

Smith-Miles, K. and Lopes, L. (2012). Measuring instance difficulty for combinatorial optimization problems. *Computers and Operations Research*, 39(5):875 – 889.

Stadler, P. (1996). Landscapes and their correlation functions. *Journal of Mathematical Chemistry*, 20:1–45.

Tillmann, N. and De Halleux, J. (2008). Pex: White box test generation for .net. In *Proceedings of the 2Nd International Conference on Tests and Proofs*, TAP'08, pages 134–153. Springer-Verlag.

Utting, M., Pretschner, A., and Legeard, B. (2012). A taxonomy of model-based testing approaches. *Softw. Test., Verif. Reliab.*, 22(5):297–312.

Vanneschi, L., Tomassini, M., Collard, P., and Vérel, S. (2006). Negative slope coefficient: A measure to characterize genetic programming fitness landscapes. In *Genetic Programming*, volume 3905 of *Lecture Notes in Computer Science*, pages 178–189. Springer Berlin Heidelberg.

Vassilev, V. K., Fogarty, T. C., and Miller, J. F. (2000). Information characteristics and the structure of landscapes. *Evol. Comput.*, 8(1):31–60.

Vivanti, M., Mis, A., Gorla, A., and Fraser, G. (2013). Search-based data-flow test generation. In *IEEE 24th International Symposium on Software Reliability Engineering, ISSRE 2013*, pages 370–379. IEEE Computer Society.

Watkins, A. L. (1995). The automatic generation of test data using genetic algorithms. In *Proceedings of the 4th Software Quality Conference*, volume 2, pages 300–309.

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

Weinberger, E. (1990). Correlated and uncorrelated fitness landscapes and how to tell the difference. *Biological Cybernetics*, 63:325–336.

Weinberger, E. D. (1991). Local properties of kauffman's N-k model: A tunably rugged enegy landscape. *Physical Review A*, 44(10):6399–6413.

Whitley, D. (1989). The GENITOR algorithm and selection pressure: Why rank-based allocation. In Schaffer, J. D., editor, *Proc. of the Third Int. Conf. on Genetic Algorithms*, pages 116–121, San Mateo, CA. Morgan Kaufmann.

Williams, N., Marre, B., Mouy, P., and Roger, M. (2005). Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis. In Cin, M. D., Kaâniche, M., and Pataricza, A., editors, *Dependable Computing - EDCC-5, 5th European Dependable Computing Conference, Budapest, Hungary, April 20-22, 2005, Proceedings*, volume 3463 of *Lecture Notes in Computer Science*, pages 281–292. Springer.

Xanthakis, S., Ellis, C., Skourlas, C., Le Gall, A., Katsikas, S., and Karapoulios, K. (1992). Application of genetic algorithms to software testing. In *Proceedings of the 5th International Conference on Software Engineering and its Applications*, pages 625–636.

Xin, B., Chen, J., and Pan, F. (2009). Problem difficulty analysis for particle swarm optimization: deception and modality. In *Proceedings of the first ACM/SIGEVO Summit on Genetic and Evolutionary Computation*, pages 623–630. ACM.