Metaheuristic Optimisation and Mutation-Driven Test Data Generation

Matthew Patrick

Abstract Metaheuristic optimisation techniques can be used in combination with mutation analysis to generate test data that is effective at finding faults and reduces the human effort involved in software testing. This chapter describes and evaluates various different metaheuristic techniques and considers their underlying properties in relation to test data generation. This represents the first attempt to bring together, compare and review ideas and research related to mutation analysis and metaheuristic optimisation. The intention is that by considering these application areas together, we can appreciate and understand important aspects of their strengths and weaknesses. This will allow us to make suggestions with regards to the ways in which they may be used together for maximum effectiveness and efficiency.

Keywords Metaheuristic optimisation · Mutation analysis · Test data generation · Search based software engineering · Fitness function · Hill climbing · Evolutionary optimisation · Swarm intelligence

1 Introduction

Metaheuristic techniques are abstract procedures that can be used to find or generate lower-level heuristics [1]. Although metaheuristic optimisation is not guaranteed to identify the global optimum solution to a problem [2], it is typically able to find a solution that is sufficiently good enough to be practically useful. Metaheuristic optimisation is particularly effective compared to simple deterministic heuristics when the information available is incomplete, imperfect or limited in some way [1]. The key advantage that metaheuristic optimisation has over other techniques is that it is not necessary to specify how to produce an effective test suite in advance [3].

M. Patrick (✉)
Department of Plant Sciences, University of Cambridge, Downing Street, Cambridge CB2 3EA, UK
e-mail: mtp33@cam.ac.uk

W. Pedrycz et al. (eds.), *Computational Intelligence and Quantitative Software Engineering*, Studies in Computational Intelligence 617,
DOI 10.1007/978-3-319-25964-2_5

Instead, metaheuristic optimisation uses a fitness function to automatically search for optimal solutions from those that are available [4]. It evaluates how good (fit) each potential solution is and takes advantages of patterns in the fitness landscape in order to seek out and identify optimal solutions to complex and challenging problems.

Metaheuristic optimisation can be applied to the task of test data generation by representing the set of test cases being optimised as a search space for the optimisation technique to explore [3]. Most test adequacy criteria can be encoded directly as fitness functions so that every time a testing goal is achieved, the value returned by the fitness function is slightly higher. For example, in order to generate a test suite that exercises all the branches in the program under test, a fitness function can be constructed so that it counts the number of branches [3] that have been covered so far and assesses how close a test input comes to executing each uncovered branch.

This chapter investigates the ways in which different forms of metaheuristic optimisation have been applied to generate test suites that perform well under mutation analysis. Mutation analysis is a stringent and powerful technique for evaluating the ability of a test suite to find faults [5]. It generates a large number of program variants (known as mutants), each of which features a small syntactic change to the logical and arithmetic constructs of the program code. Mutants are designed to be based on faults programmers are likely to make, so that a test suite which detects most of the artificial mutants is expected to perform well against actual faults [6].

It is difficult to represent the results of mutation analysis as a fitness function, as the procedure is more complex than other testing criteria [7]. Not only is it necessary for the execution of some test case to reach point of mutation, but it must also be able to cause a difference at the point of mutation and propagate that difference to the output. A number of different fitness functions are described in this chapter ranging from simple techniques that count the proportion of mutants detected by the test suite [7–9] through to advanced methodologies that encourage individual test cases to reach, effect and propagate a difference to the output [10–12].

A wide variety of metaheuristic optimisation techniques are explored. They range from simple procedures such as the alternating variable method [12, 13] through to nature-inspired algorithms inspired by biological models of population genetics [7, 11] and coordination within self-organising ant colonies [14, 15]. The techniques have been divided into three broad categories: hill climbing techniques search locally for optimal solutions by increasing or decreasing the current values by a certain amount; evolutionary optimisation techniques evolve a population of candidate solutions by selecting, adapting and recombining existing candidates; and finally swarm intelligence techniques, which optimise a diverse range of solutions that can adapt and respond rapidly to changes in the environment as and when necessary.

Testing is an important part of the software development process because, if left undetected, faults can have serious harmful effects [16]. Mutation Analysis may be used to evaluate the fault-finding abilities of test cases and metaheuristic

optimisation can apply this information to generate highly effective test suites [5]. There a wide range of ways in which mutation analysis and metaheuristic optimisation can be combined and it is difficult to know which of these techniques to apply. This chapter compares the similarities and differences between the techniques that have been developed so far and attempts to identify some important properties. It is not possible to identify one technique that is always the best, as they are highly dependent on the particular application. However, it is hoped this chapter will serve as a starting point to guide the practitioner in the right direction.

2 Test Data Generation

Software testing is the process of exercising software with a sample of possible inputs, chosen so as to demonstrate its correctness in a convincing manner [25]. Testing is a key part of the development lifecycle because it helps to ensure that programs function as intended. Test data may be generated according to the specification (black-box) or internal structure (white-box) of the software [16]. It is typically too expensive in computation and human effort to apply black-box or white-box techniques to test software exhaustively [16]. Programs have too many paths and potential input values to test them all. As a result, black-box testing is often performed using randomly chosen inputs and white-box testing is considered adequate once a certain set of structural components are covered by the test suite.

Inadequate testing may result in software products that are unsatisfactory or unsafe. For example, during the first Gulf war, a rounding error in a Patriot surface-to-air missile battery led to it failing to identify an incoming Iraqi Scud missile [17]. As a result, 28 American soldiers were killed and many more were injured. In 2012, an undisclosed fault in a high frequency trading program caused a financial services firm (Knight Capital Group) to lose $440 million in 30 min [18]. The firm lost 75 % of the value of its stock in two days and was sold to another trading company four months later. Testing is crucial for detecting failures and mistakes in software.

Developers make various kinds of mistakes, ranging from incorrectly interpreting the specification through to underestimating the usage requirements or just plain typographic mistakes [19]. Developer mistakes are known as faults. More broadly, a fault is defined as an incorrect step, process or data definition within a program [20]. Faults lead to errors in software behaviour. Testing aims to find as many of the faults in a program as possible by executing it with a variety of inputs and conditions so as to reveal errors [16]. Each set of inputs and conditions used in testing is known as a test case and a collection of test cases is called a test suite. Successful test data generation finds faults in the program under test with as few test cases as possible.

Software is difficult to test because it is intangible, unique and highly specialised to a particular purpose [21]. It is estimated that between 30 and 90 % of the labour resources required to produce a working program are spent on software testing [19].

For example, Microsoft employ approximate one test engineer for every developer [22]. Yet, despite this investment many faults are often missed. The Java Compatibility Kit [23] is an extensive test suite developed for the Java Development Kit (JDK), yet there are still thousands of additional JDK bug reports in Sun's bug database [24]. Most programs have too many paths to show that they are all correct [25]. Testing is very expensive and, partly as a result, it is often incomplete.

Test data generation techniques save time and money, as well as improve the standard of testing by creating test suites automatically according to some adequacy criterion [26]. Zhu et al. [27] describe three categories of criteria: Structural testing emphasises the need to exercise particular components in the program code (statements, branches, paths etc.); Error-based testing ensures the input domain is covered thoroughly (e.g. by partition testing); and Fault-based testing (i.e. mutation analysis) aims to detect a range of artificially introduced mistakes in the software.

3 Mutation Analysis

Mutation Analysis is a fault-based testing technique based around the idea that small syntactic changes can be used to simulate actual faults. The concept was first introduced in 1971 by Richard Lipton [28]. Since then, there have been over 400 research papers and at least 36 software tools have been developed [5, 29]. Mutation analysis is considered superior to other testing criteria because it measures a test suite's ability to find faults (of course this depends on how representative the mutants are).

Mutation analysis is supported by the *competent programmer hypothesis* (experienced programmers produce programs that are either correct or very close to being correct) [30] and the *coupling effect hypothesis* (test suites capable of detecting all the simple faults in a program can also detect most of the more complex ones) [31]. Developers may understand how the program should behave and make a small mistake in its implementation, or have a slight misconception about the intended behaviour and carry it through to the implementation. In either case, small syntactic changes are claimed to be sufficient to represent most faults [6].

A *mutant* is a copy of the original program that has had a small syntactic change (known as a *mutation*) made to its logical and arithmetic constructs. Mutations are typically applied one at a time. For example, in Fig. 1, the greater-than inequality of line 1 has been replaced with a greater-than-or-equal-to inequality. A mutant is said to be *killed* by input values that cause it to output a different result to the original program. The mutant in Fig. 1 is killed when the value of 'a' is equal to 10 and the value of 'b' is not equal to zero. Mutation analysis evaluates test suites according to the number of mutants they kill. A test suite is considered to be effective for the program under test if it kills a large proportion of the mutants that are produced.

The proportion of mutants killed by the test suite is known as its mutation score (see Eq. 1). This value may be used to indicate weaknesses, since if the test suite

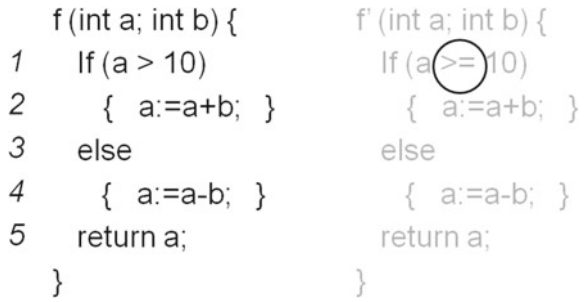

Fig. 1 A simple syntactic mutation

fails to kill some of the mutants, it is also likely to miss actual faults. Some mutants cannot be killed, since they function equivalently to the original program for every possible input; they are typically removed from the calculation, so that mutation score is correctly scaled between 0 and 1. Mutation scores are a more useful measurement for test data generation than the actual number of faults detected because finding many faults in the program under test may indicate good test data or poor software. In this way, Mutation analysis functions as an independent adequacy criterion that can be used to provide confidence in the quality of software.

$$\text{Mutation score} = \frac{\text{number of mutants killed}}{\text{number of non-equivalent mutants}} \quad (1)$$

Mutation analysis can be applied iteratively to help improve the quality and efficiency of a test suite (see Fig. 2). The presence of mutants that are not killed by

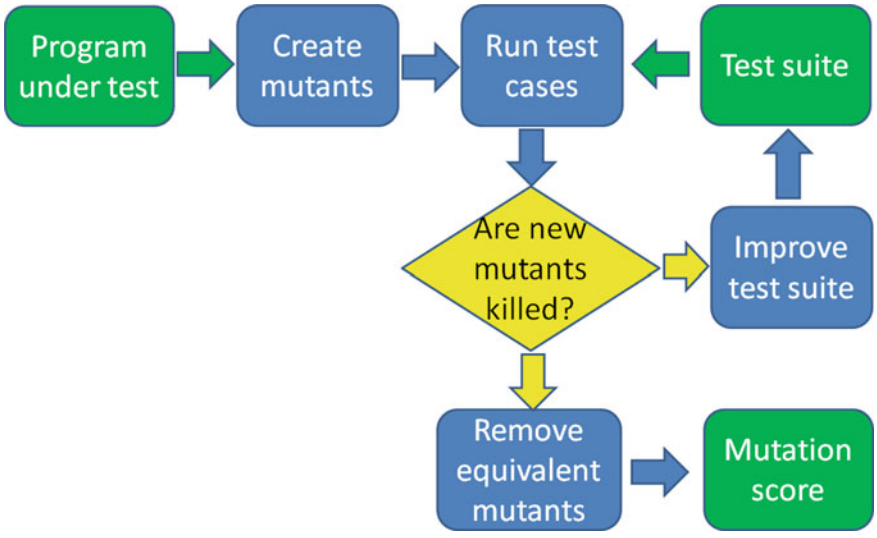

Fig. 2 The process of mutation analysis, adapted from [32]

the existing test suite indicate behaviours of the program under test that are not adequately represented. Test cases may be added or removed in an attempt to kill the remaining mutants as efficiently as possible. Mutation analysis is the reapplied and the test suite is improved until it is able to kill all (or most of) the non-equivalent mutants. The intention is that by generating test data to improve the test suite against a set of mutants, its ability to detect actual faults will also be improved.

4 Metaheuristic Optimisation

Metaheuristic optimisation has been used on a wide variety of problems, from scheduling and planning through to data mining and machine intelligence [2]. It explores candidate solutions to a problem efficiently and identifies optimal or near-optimal solutions. Metaheuristic techniques are not problem-specific [4]. They make few assumptions about the problem being solved, so can be applied 'out of the box' as generalised tools for optimisation. Metaheuristics range from straightforward search strategies through to advanced methodologies inspired by nature. They can be used to solve complex optimisation problems [4] and have been shown to be effective on problems that involve uncertain, stochastic or dynamic information.

Metaheuristics guide the search procedure through a fitness landscape of candidate solutions (see Fig. 3). For example, each point in the landscape may represent the suitability (or fitness) of a particular test suite. The landscape features peaks and troughs with various heights, gradients and spatial arrangements, as well as other areas where fitness is approximately the same [2]. The aim of metaheuristic optimisation is to guide the search towards the highest possible value in the landscape. In our example, this would correspond to a highly effective test suite.

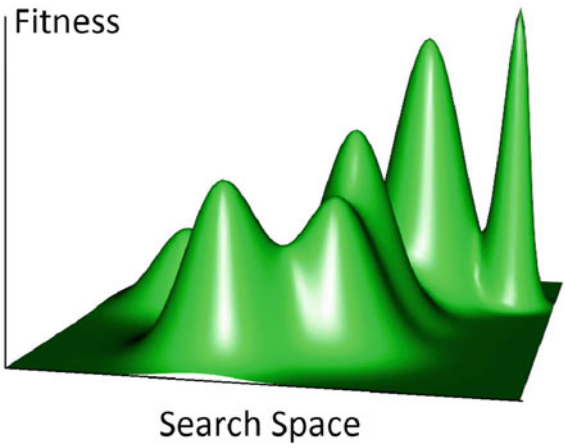

Fig. 3 A metaheuristic fitness landscape

Metaheuristics guide the search using a simple set of rules that describe what to do when particular fitness values are encountered. Typically these rules are stochastic, so that the decisions made and the solutions found are dependent on a set of random variables [4]. Fitness values are not usually calculated in advance, but evaluated as and when needed by the optimisation technique. For example, each test suite can be applied to a set of mutants to generate a mutation score. Metaheuristic optimisation does not guarantee that a globally optimal solution will be found, but it typically finds good solutions with less computational effort than other techniques.

Rather than specifying how to produce an effective test suite, metaheuristic optimisation searches the fitness landscape iteratively, taking advantage of patterns of suitability as it goes along [2]. Figure 4 describes the general processes involved in metaheuristic optimisation. First, one (or a set of) initial candidates is chosen (typically at random). If a candidate meets the fitness termination criteria, the candidate solution is saved and the optimisation process is stopped. Otherwise, further optimisation is performed to create new candidates to evaluate. The process continues until a candidate is found that meets the fitness criteria.

Metaheuristic techniques have to find a balance between intensifying and diversifying their search. Intensification exploits information from previous good solutions to fine tune parameters on a local scale [2]. Diversification explores new regions of the landscape on a global scale and finds solutions different from those found before. Rather than fixing the rates at which diversification and intensification are used, metaheuristic techniques typically alternate between the two approaches dynamically [33]. This allows flexibility to the patterns of fitness they encounter.

A decision must also be made regarding the number of candidate solutions to maintain concurrently. Single solution approaches maintain one candidate at a time

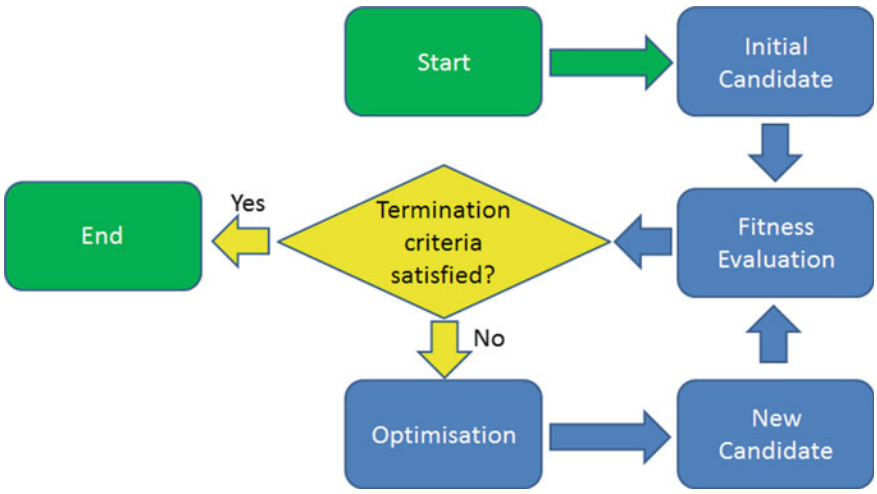

Fig. 4 The process of metaheuristic optimisation

[33], adapting and enhancing its parameter values to form a continuous search trajectory through the fitness landscape. By contrast, population-based approaches maintain multiple candidates [33] and adapt their parameter values simultaneously, so as to evolve a set of points in the fitness landscape. Single solution approaches are typically biased towards intensification [1]. They are effective at fine-tuning parameter values through local search. Population-based approaches typically focus on diversification [1]. They are better suited towards exploring the entire landscape.

5 Using Metaheuristic Optimisation to Kill Mutants

Metaheuristic optimisation is an efficient way to generate and select test suites for mutation analysis. A wide variety of techniques have been used, ranging from hill climbing through to evolutionary optimisation and swarm intelligence. There are also a number of ways to describe and evaluate the fitness landscape. It may be constructed so as to optimise input values for individual test cases or for the entire test suite. This section explores the various metaheuristic techniques and fitness functions that have been used in mutation analysis for test data generation.

There is a danger of introducing new techniques just because they are based upon a new biological metaphor or perform optimisation in a slightly different way, rather than because they are more effective than any other technique. Sörensen [34] argues this could lead metaheuristic research away from scientific rigour. We therefore focus on the general principles, advantages and disadvantages of each technique.

5.1 Fitness Functions Based on Mutation Analysis

The choice of fitness function is important for successful metaheuristic optimisation. Good fitness functions have two key properties: they must be able to differentiate effectively between desirable and undesirable candidate solutions—without this, optimisation may converge to a poor solution, or not converge at all; they must also be inexpensive to calculate—since the fitness function is used extensively during optimisation, if it is too expensive, the optimisation process will become unfeasible.

One option is to evaluate candidate test suites according to their mutation score (see Eq. 1). The higher the mutation score, the fitter the test suite is considered to be. Ghiduk [7] and Mishra et al. [8] apply mutation score as a fitness function in their genetic algorithms for test data generation. Baudry et al. [9] take a similar approach, but with a bacteriological algorithm. Mutation score is a simple and direct fitness function, but it can also be very expensive. Every time a test case is added or changed, it must be run against all of the mutants to find out which ones are killed.

Other fitness functions exist that are cheaper than mutation analysis, but still guide optimisation towards mutant killing test cases. Independent path coverage [35] uses McCabe's Cyclomatic Complexity metric to identify a set of 'basis paths' that can be combined to describe all the paths through a program, then counts how many are covered by the test suite. Mala and Mohan [36] apply independent path coverage as part of their fitness function. It is more stringent than branch coverage and (unlike full path coverage) it is computationally feasible.

$$\text{Independent Path Coverage} = \frac{\text{number of basis paths covered}}{\text{total number of basis paths}} \tag{2}$$

The techniques described so far are suitable for optimising an entire test suite at once, but sometimes it is more efficient to use other intermediate criteria that guide the optimisation of individual test cases to kill each mutant. With mutation score, it is only possible to know whether a particular mutant is killed. Intermediate criteria may be used to evaluate which of the branches leading up to the mutant are exercised, the size of difference that the mutant introduces in the program state and the depth of its propagation through to the output [3]. Each one of these criteria can be considered individually for a more incremental measure of test suite fitness.

The approach level and branch distance metrics (see Fig. 5) can be used to describe how close a test case is towards causing the program execution to reach a point of mutation [3]. The branch containing the mutated statement is given an approach level of zero, and then it is incremented by one for every branch that could prevent the mutation being reached. This means branches with the highest approach level are farthest from the point of mutation. Branch distance measures the difference between the evaluated and required value at the first branch condition where

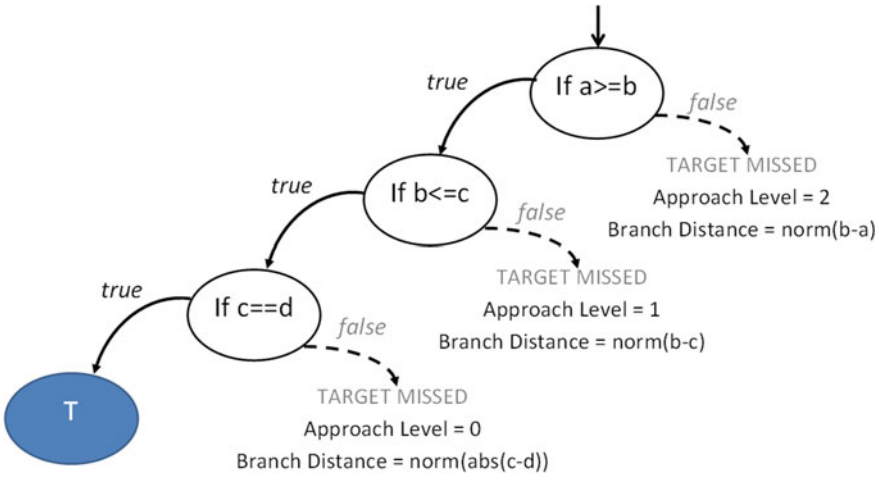

Fig. 5 Example of approach level and branch distance

execution diverts from the intended path [3]. Together, approach level and branch distance can be used to guide test cases towards reaching each point of mutation.

Another metric is needed to describe how far the test data is away from causing a difference at the point of mutation. Papadakis and Malevris [12] measure this using a metric called mutation distance, which is based on the previously described metric for branch distance. For example, consider the original expression $a > b$. If it is mutated to $a \geq b$, the mutation distance is $abs(a - b)$, if it is mutated to $a < b$ it is infinite and if it is mutated to $a \leq b$ it is zero. When combined, the approach level, branch distance and mutation distance satisfy a condition known as weak mutation.

For strong mutation to be achieved, the effects of each mutant must be propagated to the output. Bottaci [10] measures propagation in terms of the number of unequal state pairs that occur between a mutant and the original program once the mutation point is reached. It can be difficult to synchronise the sequence of state pairs when a mutant causes the path to diverge. This is because path divergence does not guarantee state divergence and the program may return back to the original path and state at some later point. It is also prohibitively expensive to keep track of and compare all the program states that result from test data generation. This is why Ayari et al. [15] only implement Bottaci's metrics for reaching and causing a difference at the point of mutation in their optimisation technique for test data generation.

Fraser and Zeller [11] measure the potential for a test case to propagate the effect of a mutant in terms of its impact on statement coverage and function return values. This alleviates the previous problems regarding synchronisation because it does not matter in which order the statements are executed. As well as considering how the impact of a mutant is affected by the choice of test case, we can also consider how the execution of each node in the program affects the mutant's impact. Certain nodes are more likely to propagate the effect of a mutant to the output than others. For every node following the point of mutation, Papadakis et al. [12] count the number of mutants that are killed when that node is executed. The more mutants that are killed, the greater the impact that particular node is considered to have.

Clearly there is a trade-off between the expressive capability and cost of the criteria used to evaluate fitness. Simple criteria such as branch coverage are computationally inexpensive, but produce test suites that are inefficient at killing mutants. Smaller test suites may be produced that achieve a higher mutation score by incorporating propagation into the fitness criteria, but this is typically more expensive. Kaur and Goyal [14] improve the efficiency of their generated test suites by maximising the ratio of mutants killed to execution time (see Eq. 3). However, it is also important to consider the human effort involved in evaluating and understanding the outputs produced by a test suite. One way to do this is to specialise the tests so that they kill specific mutants. Patrick et al. [45] apply the fitness function in Eq. 4 to target each test case at a different group of mutants. Ultimately, human effort is more costly than computational expense and test data generation techniques are more attractive if they reduce the time required to evaluate the test cases.

$$\text{Test case quality} = \frac{\text{number of mutants killed}}{\text{test case execution time}} \tag{3}$$

$$\begin{gathered}\text{Test case specialisation} = \sum_{m \in M} \sum_{s \in S} \frac{(\bar{K}_m - \bar{K})^2}{(K_{s,m} - \bar{K}_m)^2} \\ \bar{K}_m = \left(\sum_{s \in S} K_{s,m} \right) / |S| \qquad \bar{K} = \left(\sum_{m \in M} K_{s,m} \right) / |M|\end{gathered} \tag{4}$$

(S is the set of test suites, M the set of mutants, $K_{s,m}$ is the number of times s kills m, $\bar{K}_m$ is the average number of times m is killed and $\bar{K}$ is the average number of times any mutant is killed).

5.2 Hill Climbing

Hill climbing is a simple, yet powerful technique for metaheuristic optimisation. It is considered to be a form of local search [37] because it only explores candidates in the neighbourhood of the current solution. Hill climbing is not guaranteed to find the best possible solution across the entire landscape, but it does quickly find a solution that is locally optimum [37]. In many cases, this is sufficient. Other, more advanced techniques may give better results under certain circumstances, but hill climbing often performs equally well [38] and the algorithm is easier to understand [37]. As such, hill climbing is typically used as a starting point for optimisation research and a benchmark with which to compare other more complicated algorithms [38]. Hill climbing is simple to implement, easy to understand and yet surprisingly effective.

Hill climbing starts with an initial candidate solution (typically selected at random), then updates its parameter values through a series of iterations. At each step, one of the parameter values is changed to a new value in the neighbourhood of the current solution. Hill climbing takes advantage of local gradients in fitness by selecting neighbouring candidates if they improve the fitness evaluation. The candidate's values are repeatedly adjusted until no further improvement in fitness is obtained.

A decision must be made as to the neighbourhood of candidate solutions to evaluate. Once the neighbours are evaluated, hill climbing moves to the neighbour with the highest fitness and a new neighbourhood is created. The simplest strategy is to adjust the value of each parameter systematically (e.g. from left to right), then select the first such move that improves the fitness evaluation [38]. By contrast, steepest ascent hill climbing evaluates the fitness of all possible moves in the neighbourhood, then selects the one that provides the greatest improvement [38]. The steepest ascent strategy takes more time to find the best possible move at each step, but it typically requires fewer steps to reach an optimal solution than simple hill climbing.

The solution found by both the above strategies is only guaranteed to be locally optimal. To increase the chance of finding the global optimum, it may be better not to evaluate every possible move. Stochastic hill climbing selects neighbours to evaluate at each step sparsely according to a probability distribution. In an experiment with combinatorial optimisation, stochastic hill climbing found an optimum faster than both simple and steepest-ascent hill climbing [39]. It was also shown to perform better than two Genetic Algorithms, with populations of 128 and 1024 respectively [39]. This confirms the findings of Harman and McMinn [40] that even though hill climbing is straightforward to implement and easy to understand, it is still effective at test data generation compared to other more complex optimisation techniques.

Hill climbing is often applied to test data generation, using the Alternating Variable Method (AVM), a technique first introduced by Korel [13]. AVM simplifies the process of optimising a test case by adjusting each input parameter in isolation from the rest. Hill climbing is applied to the input parameters one at a time in turn. Whilst hill climbing is being applied to one of the input parameters, the values for all other parameters remain fixed. If adjusting the value for the first variable does not improve the fitness evaluation, the algorithm tries adjusting the next variable, and so on, until every parameter value in the test case has been adjusted.

Adjustments to parameter values are typically made using a combination of exploratory and pattern moves. Exploratory moves change the value of a parameter by a small amount. They are used to determine a suitable direction for the hill climb. If one of the exploratory moves increases the fitness evaluation, then the value that gave the improvement will be used as the new candidate solution. A pattern search is made in the direction of improvement, applying increasingly larger changes to the chosen variable as long as a better solution is found. If none of the neighbours have a higher fitness than the current solution, AVM continues to perform exploratory searches on the other parameters, until either a better neighbour has been found or all the parameters have been unsuccessfully explored.

Papadakis and Malevris [12] applied AVM to kill mutants from 8 small Java programs. Test inputs were optimised, according to a number of different control flow criteria; first so that they reach each point of mutation, then so that they kill the mutants located at each point. The results showed AVM to be more effective than random testing, but only by a small amount. There was little or no difference in effectiveness with fewer than 4500 fitness evaluations [12]. With 50,000 evaluations, AVM killed on average 22 % more mutants than random testing. However, AVM was significantly more effective for two programs (variants of the Triangle program). Arcuri [41] performed a full run-time analysis of AVM on the triangle program. The program performs a simple calculation to determine whether a triangle is a valid equilateral, isosceles or scalene. Hill climbing and the Alternating Variable Method may perform less well may on non-numerical and more complex programs.

In particular, hill climbing performs poorly on fitness landscapes where there are a number of local (i.e. non-global) optima (see Fig. 6). Once optimisation reaches a

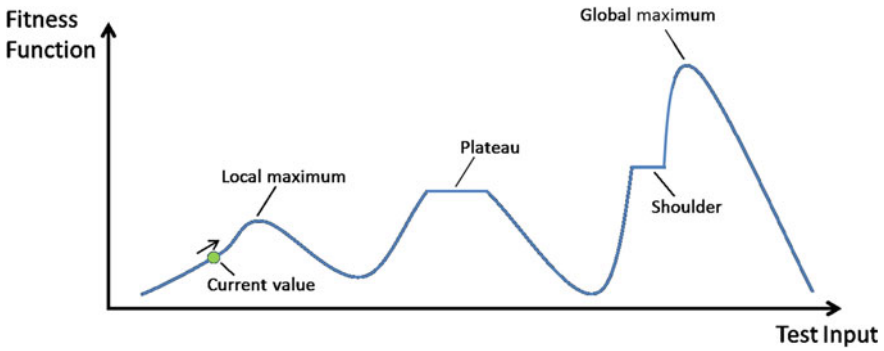

Fig. 6 A challenging landscape for hill climbing

local maximum fitness value, all the neighbouring solutions have lower fitness, so an exploratory search is likely to fail [37]. Similarly at a plateau, the neighbours all have the same fitness and there is no indication as to which direction for the hill climb to travel. The result is that hill climbing often becomes stuck at a local optimum, terminates prematurely, or (in the case of a plateaux) is left wandering aimlessly through the landscape. Success depends largely on the initial conditions, which are determined by the starting point for the hill climb [38]. For example, the global maximum in Fig. 6 has a plateau on one side, which forms a shoulder. This makes the global optimum much harder to reach from one direction than the other. One way to address this problem is to restart the hill climb from a number of different locations (usually chosen at random) to achieve a new maximum fitness value each time. The fitness values are compared and the highest one selected, so as to identify a local optimum that is closer to the global one.

Another option (where available) is to initialise hill climbing using pre-existing test cases. Most test data generation techniques start from scratch, assuming that no test cases already exist [42]. Yet, this is seldom the case. Throughout the various stages of a project, people gather test data for a a variety of different purposes: Requirements engineers capture and explain the ways in which the software is intended to behave with use-cases; software designers expand upon this information using interaction diagrams to describe the flow of information through the system; and programmers use simple test cases to check their code is functioning as they work on it. In addition, test case may be available from testing that was performed on a previous version of the program (these are known as regression tests). Even if this data has been lost, or was not created, it is still possible to generated initial tests used some automated tool, such as Dynamic Symbolic Execution [42]. Overall, there is typically no need to start testing a program from scratch.

Yoo and Harman [42] applied this idea to generate test cases for methods from two real world Java libraries: *binarySearch* is an implementation of the binary search algorithm from a scientific computing library developed at CERN; whereas *read_number* is a numerical parser taken from an Internet event notification library.

They started with a small initial test suite for each program, manually constructed so that it exercises each branch in the code, and allowed their testing tool to search for new test cases by making simple changes to the existing tests. Yoo and Harman's technique achieved mutation scores that were 3.1 and 22.2 % higher for *binarySearch* and *read_number* respectively, when compared to another test data generation tool (Iguana) that uses the more popular alternating variable method.

Pre-existing test cases can be exploited to reduce the time and effort involved in test data generation and produce more effective tests [42]. These test cases are a suitable starting point for hill climbing, since they are likely to be closer to a global optimum than test cases chosen at random. Although testing cannot be used to prove that a program is correct, some confidence may be provided in the success of a large number of test cases. Techniques, such as hill climbing, that apply a local search to existing test data, can be used inexpensively to generate further test cases for evaluation. This makes it less likely for the test cases to be over-specialised to the mutants they were created for, which is useful in ensuring repairs to a program do not allow test cases to pass without fixing the underlying problems [42].

In summary, hill climbing has been shown to be an effective technique for generating test data to kill mutants. It is particularly efficient at fine tuning test cases, but depending on the starting conditions, it is prone to becoming stuck in a local optimum. For this reason, hill climbing is suitable for being used with multiple restarts or combined with some other, more global, optimisation techniques.

5.3 Evolutionary Optimisation

Evolutionary optimisation techniques are inspired by the process of evolution in nature [43]. Candidate solutions compete for the rights to survive and reproduce. Yet, only the fittest candidates are allowed to proceed to the next generation. This leads, over a number of generations, to individuals that are highly adapted to their environment. Evolutionary optimisation techniques are able to find the global optimum even when the fitness landscape is large and complex [44]. For example, they perform well when the fitness function is discontinuous, noisy, changes over time, or has many local optima [43]. This makes evolutionary optimisation techniques suitable for the task of generating test data for large, complex programs that have many different input parameters or types and an intricate system of control and data flow.

As with stochastic hill climbing, evolutionary optimisation applies a randomised process of trial and error to seek for optimal solutions [43]. However, in contrast to hill climbing, evolutionary optimisation is non-local and it maintains a population of multiple candidates at the same time. Non-local optimisation searches a wider range of values in the fitness landscape and is less likely to get stuck in a local optimum. Population-based techniques are intrinsically parallel in nature [44]. It is not necessary to restart the optimisation process when a member of the population

fails to reach an optimum, as useful information is preserved in the other candidates.

Since the population of an evolutionary optimisation technique is fixed in size and candidates are selected to be replaced at random, some highly fit individuals are inevitably lost from the population [43]. Although it seems this could cause problems, it actually improves the optimisation. Losing highly fit (and potentially over-trained) candidates allows the search to descend into valleys in order to reach different (and potentially higher) optima [43]. This makes evolutionary optimisation more likely to find values that are closer to the global optimum than hill climbing.

Algorithm 1 presents an overview of the algorithm used by evolutionary optimisation. The initial population is typically chosen at random. Then as long as the termination condition has not been reached, the candidate solutions are evaluated and the fittest individuals are selected to continue into the next generation, after first being adapted and/or recombined according to the particular evolutionary technique.

Algorithm 1 Algorithm for Evolutionary Optimisation

1: Generate the initial population (typically at random)
2: **repeat**
3: Evaluate fitness values for each candidate solution
4: Candidate solutions compete to continue into the next generation
5: Select the fittest candidates (allowing for some randomisation)
6: *Allow the fittest selected candidate solutions to remain unchanged**
7: Adapt the selected candidates by changing some of their values at random
8: *Combine values from the selected candidates to create new candidates**
9: Form a new population from the various modifications of candidates in the previous one
10: **until** Some termination condition is reached (e.g. fitness above certain level)

* Steps 6 and 8 are not present in all evolutionary optimisation algorithms, they are used by elitist and recombinatorial techniques respectively

Ghiduk [7] represents a candidate solution of test cases using a binary vector of ones and zeroes. The length of this vector is set according to the number and precision of input values to the program under test. Initially the value of each bit is set at random, but they are adjusted throughout the optimisation process and selected according to how much they improve the mutation score of the resulting test suite [7]. Optimisation is terminated when the maximum number of generations (100) is reached or there are no further improvements in mutation score. In experiments with 13 Java programs, evolutionary optimisation achieved a mutation score of 81.8 %, compared to the 68.1 % achieved by random testing.

The encoding of candidate solutions is important as it has a significant impact on the optimisation process. Binary vector representations provide a large number of ways in which candidate solutions can be modified, but some changes are less productive than others and this can make it take longer to reach an optimum. For example, changing the most significant bit of a numerical value has a very different effect to changing the least significant bit. Fraser and Zeller [11] introduce an alternative test case representation using a sequence of method calls. The sequence

may be modified by the optimisation technique through the deletion, insertion or modification of program statements. In addition, Fraser and Zeller [11] use a more sophisticated fitness function, using control flow criteria in a similar way to Papadakis and Malevris [12]. The new technique was evaluated against manually devised test suites for two large Java libraries: Joda-Time and Commons-Math. Evolutionary optimisation produced fewer tests and killed more of the mutants from Joda-Time, but produced more tests and killed fewer mutants from Commons-Math.

There are two main driving forces in evolutionary optimisation: variation and selection [43]. At each generation, new candidate values are created by making random adaptations to the parameter values of the existing candidates. This ensures that the diversity of the population is maintained and allows new parameter values and combinations of values to be explored. Also at each generation, allowing for some randomisation in the selection process, the fittest candidates are generally selected and the weaker candidates are removed. This means that as optimisation progresses, there should be a trend towards increasingly fitter candidate solutions. The strengths of the existing candidates in the population are exploited, as their parameter values are carried through into the next generation.

The challenge in setting up an effective evolutionary optimisation technique is to balance the processes of exploration and exploitation so as to achieve an efficient trade-off between them [44]. If the technique concentrates on making the best use of the candidates that are currently available, it might not be able to reach the global optimum. Yet, if it spends time searching for other (potentially more effective) candidates, there is no guarantee that a fitter candidate will be found. Either way, the technique risks wasting time and resources that may be better applied in a different way. Mishra et al. [8] propose the use of an elitist GA to evolve test cases for each unit, whilst maintaining the test cases that have been shown to be particularly effective, in case they are effective on other units. However, it can be difficult to determine in advance for any given situation, which strategy will be the most effective, as this depends on both the program under test and the type of evolutionary algorithm that is used. In practice, it is useful to implement a small trial to compare a number of different options when starting a new project.

A further decision must be made as to whether to include recombination as part of the evolutionary optimisation technique. Recombination is used to combine parameter values from different candidates in the hope that the new combination of values will have the strengths of the previous candidates without any of their weaknesses [44]. In contrast to adaptation, which takes a single candidate solution and changes its parameter values to produce a new candidate, recombination takes two or more candidate solutions and selects parameter values from one or more of the previous candidates [43]. This allows a transfer of information in that candidates can benefit from what other candidates have learnt [44]. However, there is no guarantee that combining parameter values from different solutions will be effective—the values may only be effective in the context of the other parameter values in the original candidate. Recombination can be destructive as well as beneficial, so it is important to consider its role in an evolutionary optimisation technique carefully.

Without recombination, candidates explore the fitness landscape on their own, without interference from the other candidates, but with recombination they are able to take of things other candidates have discovered [44].

Harman and McMinn [40] claim that hill climbing outperforms evolutionary optimisation in its ability to generate effective test suites, as recombination is often disruptive to the optimisation process rather than helpful. Forrest et al. [39] claim that certain 'royal road' functions are particularly suitable for recombination because they allow different sections of the solution to be evolved individually then combined together to form the global optimum. However, it should be noted that since these examples are artificially constructed to be suitable for recombination, it is unclear how frequently these kinds of functions occur in practice. Harman and McMinn [40] claim that recombination should be avoided in most circumstances, so they prefer the use of the Alternating Variable Method for hill climbing.

Nevertheless, recombination does not necessary need to be included as part of an evolutionary optimisation technique and there are a number of ways in which local search can be combined with evolution so as to take advantage of the benefits of each approach. For example, Mala and Mohan [36] use a memetic algorithm to generate test suites. Memetic algorithms apply a local search at each generation to improve the fitness by exploring the immediate neighbourhood of the current candidate. This allows evolutionary optimisation to take advantage of local gradients in fitness as well as searching on a global scale. Mala and Mohan [36] evaluated 18 Java programs from industry and academia and found that they were able to achieve a similar mutation score to a genetic algorithm, but with fewer test cases. Other options include the CMA-ES algorithm, which uses a multivariate Gaussian distribution to describe the neighbourhood of the current best solution in a way that is a compromise between local and global search. Patrick et al. [45] applied a CMA-ES along with dynamic transformation of the program under test to identify effective subdomains of test input. In a study with 6 Java programs, they achieved a 160 % improvement in mutation score compared to random testing.

5.4 Swarm Intelligence

Swarm intelligence techniques are also inspired by a biological concept [46]. However, instead of being based on the inheritance of genetic information, they focus on the ways in which individuals cooperate by sharing information. For example, ants decide where to forage using networks of pheromone trails [47]. If ants encounter an obstacle (see Fig. 7), they look for ways around it at random. However, when some of the ants find a way around it, the other ants follow their pheromone trail to form a new route. There is a genetic element to the coordination of populations (e.g. polymorphism in ants), but the most significant factor in cooperation is self-organisation [46]. Self-organisation refers to the spontaneous way in which coordination arises at the global scale out of local interactions between organisms that are initially disorganised [49]. Individual organisms exhibit

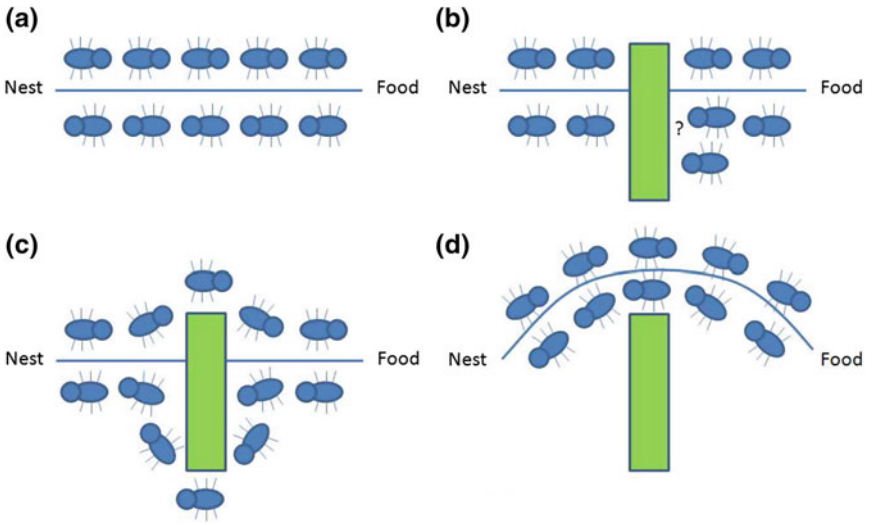

Fig. 7 Ants following pheromones to find their way around an obstacle

simple behaviour and when viewed in isolation, their actions appear noisy and random. However, when multiple organisms work together, complex collective behaviour emerges.

Self-organisation uses simple actions as seeds for random growth in order to create complex behaviour among the collective [48]. It is the way in which the simple actions of individuals interact that allows the collective to search for effective new solutions to a problem. Two forms of interaction exist: positive feedback and negative feedback [49]. Positive feedback reinforces actions that lead to a useful result. For example, when a foraging bee brings nectar back to the hive, it will choose from one of three options: If the bee has found a good source of nectar, it will dance to indicate its direction to other bees; if the source of nectar is mediocre, the bee will return back to the source but not dance; or, if the bee has found a poor source of nectar, it will abandon the source and follow the other bees [48].

Negative feedback acts as a counterbalance to positive feedback by dissuading individuals against making less useful actions [49]. For example, when a source of nectar is exhausted, or merely saturated from too many bees seeking after it at once, the bees will stop dancing so as to prevent other bees from travelling to it [48]. Even when a suitable source of nectar has been found, competition with other sources may prevent bees from travelling to it. Typically, the vigour with which a bee dances is in proportion to the quality of the source it has encountered, so that the majority of bees travel to the best source of nectar [48]. In this way, negative feedback helps to stabilise the behaviour of the collective. Together with positive feedback, negative feedback allows the colony to maintain an optimal supply of nectar to the hive.

Kaur and Goyal [14] use mutation analysis and an artificial bee colony to select and prioritise test cases for regression testing. They evaluate their approach on two C++ programs, for which test suites have been manually developed: a college system for managing course admissions (which has 35 test cases); and a Hotel Reservation system (which has 40 test cases). Kaur and Goyal [14] aim to select a subset of these test cases that form an optimised test suite. Two types of 'bees' are employed: scout bees apply a global search to explore potential candidate test suites and evaluate their fitness according to mutation analysis; by contrast, forager bees start at the fittest test suites that were observed by the scout bees and apply a local search to exploit the neighbourhood of each candidate. Test cases are selected such that they detect faults not detected by the test cases already selected. As a result, test suites can be ordered such that they kill more mutants in less time [14].

Ayari et al. [15] use ant colony optimisation to generate test suites that achieve a high mutation score. The technique starts with a global search performed by 'ants' who evaluate the fitness of random test cases according to how far away they are from killing a mutant. This distance is measured in terms of the number of critical decision nodes that are not traversed and the difference between the current and required value at the node where execution deviates from the path to the mutant [15]. Subsequent ants follow the pheromone trails left by previous ants and perform a local search to take advantage of previous fitness evaluations. Pheromone trails guide ants in constructing test cases, one parameter value at a time. At each step, the ant selects a value that was previously evaluated, or chooses a new value, in proportion to the fitness of the corresponding test cases. Ayari et al. [15] evaluate their approach using two programs written in Java: the Triangle program (as described before); and NextDate, which validates the date that is input and determines the date of the next day. Ant colony optimisation is able to kill more than twice as many mutants as a genetic algorithm and more than three times as many as hill climbing.

Artificial bee colony and ant colony optimisation require the fitness landscape to be described in a discrete way. This makes it difficult to use these algorithms for continuous optimisation tasks, such as generating test cases for programs with numerical inputs. Kaur and Goyal [14] try to get around this problem by generating test cases in advance and optimising the order in which they are executed. Whereas, Ayari et al. [15] discretise the fitness landscape into a limited number of values for each input parameter (spread evenly over the input domain) and then optimise the combination of those input values. These options both help to simplify the optimisation process, and it may be worthwhile to choose them for that reason alone. Yet other models of swarm intelligence (based on particle swarm optimisation) can be applied directly to continuous optimisation problems. Two of these techniques are reviewed below: artificial immune systems and bacteriologic algorithms.

May et al. [50] implement an artificial immune system to generate test suites that are efficient at killing mutants. Artificial immune systems optimise antibodies that are effective against specific antigens. In this case, each antibody represents a test

case and each antigen represents a mutant [50]. Test cases are optimised so that they kill at least one mutant not killed by any of the test cases stored in memory as antibodies. The collection of antibodies in memory at the end of the optimisation process are returned to the user as a test suite. The mechanism used to search for new test cases that are effective against the remaining mutants is known as clonal selection theory [50]. Clonal selection theory describes the way in which antigens activate specific antibodies according to their affinity. These antibodies then multiply in numbers by cloning and adapt to be even more effective against the antigen by a process of mutation and selection. May et al. [50] consider test cases that kill more mutants to have a higher affinity (or fitness). New test cases are mutated from the previous ones according to their affinity. High affinity antibodies generate more clones than low affinity ones, but mutate less (as they are closer to the desired solution). May et al. [50] show the artificial immune system is able to kill more mutants with fewer fitness evaluations than a genetic algorithm on four Java programs.

Baudry et al. [9] use a bacterial foraging algorithm to create an effective test suite for a C# parser. Bacteria locate food sources in their environment by sensing and following chemical gradients. They propel themselves along the gradients using long thin structures called flagella. Baudry et al. [9] interpret improvements in mutation score as gradients in food sources and model individual test cases as bacteria that are travelling towards them. Each movement of a bacterium is implemented with a small change to one of the input parameters. Test cases are selected according to their mutation score and the best test cases are allowed to remain within the new population [9]. By remembering which candidates achieve a high mutation score, it is not necessary to recalculate the mutation score for every individual in each generation. Baudry et al. [9] showed that their bacterial foraging algorithm achieved a higher mutation score than a genetic algorithm with fewer mutant executions.

Baudry et al.'s implementation differs from the classic version of a bacterial foraging algorithm [51], in that bacteria only have one mode of movement. In the original paper, Passino [51] described the way in which bacteria move by either swimming or tumbling. Initially, bacteria have no way of knowing which way to travel in order to reach a food source, so they move chaotically through their environment (tumbling). However, once a bacterium is able to detect a gradient, it starts to travel quickly towards the food source (swimming). This can be implemented by dynamically modifying the step size by which parameter values are adapted according to the strength and direction of the fitness gradient. This approach is similar to the multivariate Gaussian adaption used by CMA-ES and the combination of pattern and exploratory search moves in the alternating variable method.

5.5 Comparing Metaheuristic Techniques for Killing Mutants

Table 1 summarises the empirical studies that have been discussed in this chapter. Although it contains a significant proportion of the literature, the table is not intended to be definitive. There are other techniques that have been used to generate test data for mutation analysis which are not included. These techniques, such as adaptive random testing [54] and dynamic symbolic execution [52, 53], are outside the scope of this chapter. The studies that have been included were chosen to demonstrate are broad sample of techniques for metaheuristic optimisation.

Table 1 contains two studies on hill climbing (HC), four studies on evolutionary optimisation (EO) and four studies on swarm intelligence (SI). The studies were primarily conducted in Java, although C, C++, C# and Eiffel have also been used. Table 1 also lists the number of programs included in each study, as well as the total code length (in classes or in lines of code) and the number of mutants generated from the programs under test. Question marks indicate details that were not provided. The studies are arranged in order of the number of mutants (ranging from some unknown number up to 117,913). The latter study included 1416 classes and, although not directly specified, it probably includes the largest number of lines of code as well.

Empirical studies typically compare the performance of the technique they propose against a benchmark of random testing. Papadakis and Malevris [12] showed with 50,000 test cases that hill climbing killed on average 22 % more mutants than random testing [12]. Similarly, Ghiduk [7] showed with 1000 test cases that evolutionary optimisation achieved a mutation score of 81.8 %, compared to the 68.1 % achieved by random testing. Finally, Kaur and Goyal [14] showed that swarm intelligence selected 35 and 40 test cases respectively for two different programs in an order that killed more mutants in less time than random testing. We can therefore state with reasonable authority that metaheuristic optimisation techniques can be more efficient than random testing. Metaheuristic optimisation can be

Table 1 Summary of empirical studies

Study	Technique	Language	#Programs	Code length	#Mutants
Kaur and Goyal [14]	SI	C++	2	NA	NA
Mala and Mohan [36]	EO	Java/C++	18	305 (classes)	NA
Ayari et al. [15]	SI	Java	2	127 (LOC)	198
Yoo and Harman [42]	HC	C	4	NA	1267
Baudry et al. [9]	SI	Eiffel/C#	2	35 (classes)	1647
Ghiduk [7]	EO	Java	13	927 (LOC)	1772
Papadakis and Malevris [12]	HC	Java	8	395 (LOC)	2759
May et al. [21]	SI	Java	4	290 (LOC)	3958
Patrick et al. [45]	EO	Java	10	1945 (LOC)	8114
Fraser and Zeller [11]	EO	Java	10	1416 (classes)	117,913

NA not available

said to be an effective technique for automatically generating test suites to kill mutants.

Metaheuristic optimisation techniques are able to select test cases more efficiently than random testing because they generate test cases by taking into account the results of previous tests. This can be understood through the use of a simple example (see Fig. 8). Consider the problem of selecting test cases to reach two mutants (M1 and M2). Assume the three input variables (a, b and c) have integer input domains from 0 to 9 inclusive. The probability of a randomly selected test case reaching M1 is 0.1 * 0.9 * 0.1 = 0.009 and the probability of it reaching M2 is 0.1 * 0.1 * 0.2 = 0.002. Metaheuristic optimisation may measure the distance between the current and required value at each branch condition to target the correct branch.

If in our example the value of 'a' at the point of reaching the first branch condition is 6, the branch distance is measured as 1 (i.e. 6–5). The branch condition may thus be met by decreasing the value of 'a' by 1 (from 6 down to 5). Typically, the task of optimising input values to meet a particular branch condition is less trivial than this, since the value of a variable at a certain point in the program depends upon the branches and statements that were executed before. Nevertheless, metaheuristic optimisation techniques can use the information provided by metrics such as branch distance to guide the selection of test cases to reach and kill each mutant.

It is difficult to identify the most effective metaheuristic technique from the results of the empirical studies. The results cannot be directly compared because they use different test programs written in different program languages and with different settings for the techniques. Some studies compare multiple techniques, yet their conclusions are based upon experiments with only a small number of programs. Ayari claimed [15] ant colony optimisation was able to kill more than twice as many mutants as a genetic algorithm and more than three times as many as hill climbing. May [21] showed an artificial immune system to kill more mutants than a genetic algorithm with just 12.9 % the number of fitness evaluations. Similarly Baudry [9] showed a bacterial foraging algorithm to achieve 96 % mutation score in

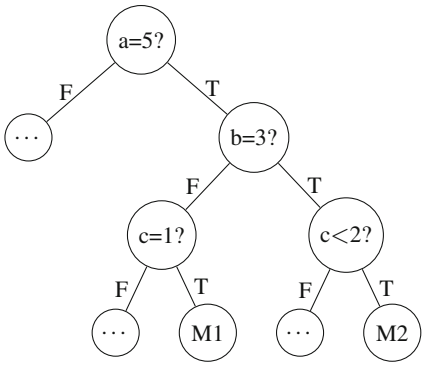

Fig. 8 A simple branch structure leading to two mutants

just 30 generations, compared to a genetic algorithm's 85 % after 200 generations. Yet in total, these three studies are based on experiments with just 8 programs. Certainly this is not enough to say swarm intelligence techniques are more effective in general. It is possible the researchers unwittingly chose programs for which it is more effective.

Each metaheuristic technique has advantages and disadvantages with respect to the program under test. For some programs, different parts of a test case can be evolved individually and then joined together by recombination in a genetic algorithm [39]. Yet for most programs, techniques such as hill climbing may be more efficient [40]. Similarly, there is no guarantee that techniques which reuse existing test cases (memetic algorithms and swarm intelligence) will be any more efficient than restarting the search. One suggestion is to evaluate a number of techniques in an attempt to determine empirically which one is the most effective for the program under test. However, in practice there is rarely enough time to fine-tune every technique to reach the full potential of its effectiveness. We need some general guidelines as to the techniques which might be effective for a particular program.

Hill climbing only stores one candidate solution at a time. This makes it difficult to take advantage of input values that were previously successful at killing mutants. Consider the example in Fig. 8. A test case may have been found to kill M1, but when targeting M2 the program could follow a different path. If this path includes a condition which prevents 'c' being less than two, this introduces a local optimum that hill climbing can only resolve by restarting the search. It may be possible to get around this problem by starting the search with input values from a test case that killed a nearby mutant. Yet, the steps taken from these values may still lead to a local optimum. The mutant might only be killed after a certain number of loop iterations or a particular branch is taken. Of course, hill climbing can be restarted as many times as is necessary, but in the worst case this degenerates to random testing.

Evolutionary optimisation techniques maintain a population of candidate solutions, which allows them to be more robust against local optima. Even if most of the candidates follow a path that prevents the goal being reached, a candidate which follows the right path can be used to create a new population and avoid having to restart the search. Techniques such as CMA-ES [45] use information about the rate at which fitness increases to maintain a population of candidates that is likely to be efficient at finding the global optimum solution. In addition, recombination can sometimes be applied to two locally optimum solutions to produce a candidate that is nearer to the global optimum. For example, when trying to kill M2, recombination can take the value of 'b' from the test case that fails at the 'c < 2' branch condition and the value of 'c' that kills M1. Since 'c = 1' is completely subsumed by the 'c < 2' condition, the new test case will kill M2. Evolutionary techniques are more likely to find the global optimum in program that as lots of local optima than hill climbing.

It seems counter-intuitive not to take advantage of local fitness gradients when available. Swarm intelligence combines the best of evolutionary optimisation and hill climbing in an approach that is population-based and involves local search.

They make it easier to kill M2, not only by remembering the test case which kills M1, but also a number of other tests that are found to be fit. Yet it is difficult to uniquely define these techniques, since evolutionary optimisation can also keep some of the fittest solutions in its population and variation may be applied locally as well as globally. Swarm intelligence has not yet been evaluated as thoroughly for test data generation as evolutionary optimisation. More work is required to determine what distinguishes these techniques, both in their definition and effectiveness.

6 Conclusions

Testing is a challenging, but vitally important part of the software development process. Time and effort spent on improving a test suite is worthwhile if it reduces the damage caused by faults and the work required to repair them later. Mutation Analysis may be used to evaluate the effectiveness of test cases and it provides indications as to how to improve them. The effectiveness of a test suite can be improved iteratively by combining mutation analysis with metaheuristic optimisation. Metaheuristic optimisation is highly suitable for this task because it makes few assumptions about the problem being solved. Rather than specifying how to produce an effective test suite, metaheuristic optimisation searches the fitness landscape iteratively and takes advantage of patterns of suitability as it goes along.

This chapter explored the application of three kinds of metaheuristic optimisation technique for test data generation. Hill climbing is efficient at fine tuning test cases, but depending on the starting conditions, it is prone to becoming stuck in a local optimum. Evolutionary optimisation is able to find the global optimum even when the fitness landscape is large, complex, noisy or discontinuous. However, swarm intelligence is more adaptable to change because it optimises a range of potential solutions and favours diversity over perfection. This means that when the software changes, or new unexpected faults are found, swarm intelligence is able to react more quickly to the new situation than other techniques. On the other hand, since swarm intelligence is based upon redundancy, it can be less efficient than other techniques and the candidate solutions it produces are not equally as effective. Each technique has its advantages and disadvantages, so it is necessary to choose between them according to purpose, or take a hybrid approach using multiple techniques.

Metaheuristic techniques are often applied as a ‘black box’ tool for optimisation. This has its obvious advantages, but it also brings with it some problems. First, the results can be difficult to understand. Why were these input values chosen? What is the purpose of this particular test case? Secondly, it is difficult to know what the best settings for a metaheuristic technique will be. How should we set the fitness function? Should we use recombination or elitism etc. and what should the termination criteria be? Finally, they can also take longer than deterministic techniques. We need to be prepared to leave metaheuristic optimisation techniques running in the background and allow enough time for them to produce an effective test suite.

The key to success when designing a metaheuristic technique for mutation analysis is not to focus too much on the metaphor that is being used, but on the decisions that cut across the different kinds of optimisation technique. A suitable balance must be found between intensifying and diversifying the search. A decision must also be made regarding the number of candidate solutions to maintain. A large population allows a broad range of options to be explored and is less likely to become stuck in a local optimum, but it tends to be less appropriate for fine-tuning the solution. The choice of fitness function is also very important. Good fitness functions must differentiate clearly between desirable and undesirable candidate solutions, but not be too expensive to calculate. It is not usually possible to make the correct decisions in advance, so it is important to compare and evaluate the different options empirically.

References

1. Talbi, E.-G.: Metaheuristics: From Design to Implementation. Wiley, Hoboken (2009)
2. Yang, X.-S.: Metaheuristic optimization: algorithm analysis and open problems. Lect. Notes Comp. Sci. **6630**, 21–32 (2011)
3. McMinn, P.: Search-based software test data generation: a survey. Softw. Test. Verif. Reliab. **14**, 105–156 (2004)
4. Bianchi, L., Dorigo, M., Gambardella, L.M., Gutjahr, W.J.: A Survey on metaheuristics for stochastic combinatorial optimization. Nat. Comput. **8**, 239–287 (2009)
5. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Trans. Softw. Eng. **37**, 649–678 (2011)
6. Offutt, A.J.: Investigations of the software testing coupling effect. ACM Trans. Softw. Eng. Methodol. **1**, 5–20 (1992)
7. Ghiduk, A.S.: Using evolutionary algorithms for higher-order mutation testing. Int. J. Comp. Sci. Issues **11**, 93–104 (2014)
8. Mishra, K.K., Tiwari, S., Kumar, A., Misra, A.K.: An approach for mutation testing using elitist genetic algorithm. In: Proceedings of IEEE International Conference Computer Science Information Technology 426–429 (2010)
9. Baudry, B., Fleurey, F., Jézéquel, J.-M., Le Traon, Y.: From genetic to bacteriological algorithms for mutation-based testing. Softw. Test. Verif. Reliab. **15**, 73–96 (2005)
10. Bottaci, L.: A genetic algorithm fitness function for mutation testing. In: Proceedings of International Works. Software Engineering Metaheuristic Innovative Algorithms, pp. 3–7 (2001)
11. Fraser, G., Zeller, A.: Mutation driven generation of unit tests and oracles. In: Proceedings of 19th International Symposium Software Testing Analysis, pp. 147–158 (2010)
12. Papadakis, M., Malevris, N.: Searching and generating test inputs for mutation testing. SpringerPlus **2** (2013)
13. Korel, B.: Automated software test data generation. IEEE Trans. Softw. Eng. **16**, 870–879 (1990)
14. Kaur, A., Goyal, S.: A bee colony optimization algorithm for fault coverage based regression test suite prioritization. Int. J. Adv. Sci. Tech. **29**, 17–30 (2011)
15. Ayari, K., Bouktif, S., Antoniol, G.: Automatic mutation test input data generation via ant colony. In: Proceedings of 9th Annual Conference Genetic Evolutionary Computation, pp. 1074–1081 (2007)
16. Myers, G.J., Sandler, C., Badgett, T.: The Art of Software Testing. Wiley, Hoboken (2012)

17. Blair, M., Obenski, S., Bridickas, P.: Patriot missile software problem. Technical report GAO/IMTEC-92-26, United States General Accounting Office (2002)
18. Heusser, M.: Software testing lessons learned from Knight Capital Fiasco. CIO Magazine (2012). http://www.cio.com/article/713628/Software_Testing_Lessons_Learned_From_Knight_Capital_Fiasco/. Cited 28 Sep 2014
19. Bezier, B.: Software Testing Techniques. Van Nostrand Reinhold, New York (1990)
20. Jay, F., Mayer, R.: IEEE standard glossary of software engineering terminology. Technical report 610.12-1990, IEEE (1990)
21. May, P.S.: Test data generation: two evolutionary approaches to mutation testing. Ph.D. thesis, Department of Computer Science, University of Kent, Canterbury, UK (2007)
22. Pacheco, C., Lahiri, S., Ball, T.: Finding errors in .NET with feedback-directed random testing. Technical report MSR-TR-2008-29, Microsoft Research (2008)
23. Sun Microsystems: Java Compatability Kit 6b User's Guide. Sun Microsystems (2012). http://openjdk.java.net/groups/conformance/docs/JCK6bUsersGuide/JCK6b_Users_Guide.pdf. Cited 28 Sep 2014
24. Oracle: Java Bug Database. Oracle (2012). http://bugs.sun.com. Cited 28 Sept 2014
25. Dijkstra, E.W.: Notes on structured programming. In: Dahl, O.J., Dijkstra, E.W., Hoare, C.A. R. (eds.) Structured Programming, pp. 1–82. Academic Press Ltd., London (1972)
26. Mahmood, S.: A systematic review of automated test data generation techniques. Masters thesis, School of Engineering., Institute of Technology Box, Ronneby, Sweden (2007)
27. Zhu, H., Hall, P.A.V., May, J.H.R.: Software unit test coverage and adequacy. ACM Comput. Surv. **29**, 366–427 (1997)
28. Lipton, R.: Fault diagnosis of computer programs. Technical report, School of Computer Science, Carnegie Mellon University (1971)
29. Jia, Y., Harman, M.: Java mutation testing repository. UCL (2014). http://crestweb.cs.ucl.ac.uk/resources/mutation_testing_repository/. Cited 28 Sept 2014
30. Budd, T.A.: Mutation analysis of program test data. Ph.D. thesis, Department of Computer Science, Yale University, New Haven, CT (1980)
31. DeMillo, R.A., Lipton, R.J., Sayward, F.G.: Hints on test data selection: help for the practicing programmer. Computer **11**, 34–41 (1978)
32. Offutt, A.J., Untch, R.H.: Mutation 2000: uniting the orthogonal. In: Wong, W.E. (ed.) Mutation Testing for the New Century, pp. 34–44. Kluwer Academic Publishers, Norwell (2001)
33. Blum, C., Roli, A.: Metaheuristics in combinatorial optimization: overview and conceptual comparison. ACM Comput. Surv. **35**, 268–308 (2003)
34. Sörensen, K.: Metaheuristics—the metaphor exposed. Int. Trans. Oper. Res. (in press)
35. Burnstein, I.: Practical Software Testing: A Process-Oriented Approach. Springer, New York (2003)
36. Mala, D.J., Mohan, V.: Quality improvement and optimization of test cases: a hybrid genetic algorithm based approach. ACM SIGSOFT Softw. Eng. Notes **35**, 1–14 (2010)
37. Burke, E.K., Kendall, G.: Search methodologies: introductory tutorials in optimization and decision support techniques. Springer, New York (2010)
38. Simon, D.: Evolutionary Optimization Algorithms. Wiley, Hoboken (2013)
39. Forrest, S., Mitchell, M.: Relative building-block fitness and the building-block hypothesis. In: Whitley, D. (ed.) Foundations of Genetic Algorithms 2, pp. 109–126. Morgan Kaufmann, San Mateo (1993)
40. Harman, M., McMinn, P.: A theoretical and empirical analysis of evolutionary testing and hill climbing for structural test data generation. In: Proceedings of 16th International Symposium Software Testing Analysis, pp. 73–83 (2007)
41. Arcuri, A.: Full theoretical runtime analysis of alternating variable method on the triangle classification problem. In: Proceedings 1st International Symposium Search Based Software Engineering, pp. 113–121 (2009)
42. Yoo, S., Harman, M.: Test data regeneration: generating new test data from existing test data. Softw. Test. Verif. Reliab. **22**, 171–201 (2012)

43. Eiben, A.E., Smith, J.E.: Introduction to Evolutionary Computing. Springer, New York (2003)
44. Koza, J.R.: Genetic Programming III: Darwinian Invention and Problem Solving. Morgan Kaufmann, San Francisco (1999)
45. Patrick, M., Alexander, R., Oriol, M., Clark J.A.: Selecting highly efficient sets of subdomains for mutation adequacy. In: Proceedings 20th Asia-Pacific Software Engineering Conference, pp. 91–98 (2013)
46. Bonabeau, E., Dorigo, M., Theraulaz, G.: Swarm Intelligence: From Natural to Artificial Systems. Oxford University Press, New York (1999)
47. Blum, C., Merkle, D.: Swarm Intelligence: Introduction and Applications. Springer, New York (2008)
48. Camazine, S., Deneubourg, J.L., Franks, N.R., Sneyd, J., Theraula, G., Bonabeau, E.: Self-organization in Biological Systems. Princeton University Press, Princeton (2003)
49. Serugendo, G.M., Gleizes, M.-P., Karageorgos, A.: Self-organising Software: From Natural to Artificial Adaptation. Springer, New York (2011)
50. May, P., Timmis, J., Mander, K.: Immune and evolutionary approaches to software mutation testing. Lect. Notes Comput. Sci. **4628**, 336–347 (2007)
51. Passino, K.M.: Biomimicry of bacterial foraging for distributed optimization and control. IEEE Control Syst. **22**, 52–67 (2002)
52. Papadakis, M., Malevris, N.: Automatic mutation test case generation via dynamic symbolic execution. In: Proceedings of 21st International Symposium Software Reliability Engineering, pp. 121–130 (2010)
53. Harman, M., Yue, J., Langdon, W.B.: Strong higher order mutation-based test data generation. In: Proceedings of 21st ACM SIGSOFT Symposium Foundations Software Engineering, pp. 212–222 (2011)
54. Chen, T.Y., Kuo, F.-C., Liu, H., Wong, W.E.: Code coverage of adaptive random testing. IEEE Control Syst. **62**, 226–237 (2013)

Author Biography

Matthew Patrick is a Research Associate in the Department of Plant Sciences, in the University of Cambridge. He is currently working on the characterisation and regeneration of heterogeneous host landscapes for epidemiological modelling, although his interests also include software testing and search-based software engineering. Matthew strives to work at the confluence between Biology and Software Engineering.