Search Based Software Engineering: Techniques, Taxonomy, Tutorial

Mark Harman[1], Phil McMinn[2], Jerffeson Teixeira de Souza[3], and Shin Yoo[1]

[1] University College London, UK
[2] University of Sheffield, UK
[3] State University of Ceará, Brazil

Abstract. The aim of Search Based Software Engineering (SBSE) research is to move software engineering problems from human-based search to machine-based search, using a variety of techniques from the metaheuristic search, operations research and evolutionary computation paradigms. The idea is to exploit humans' creativity and machines' tenacity and reliability, rather than requiring humans to perform the more tedious, error prone and thereby costly aspects of the engineering process. SBSE can also provide insights and decision support. This tutorial will present the reader with a step-by-step guide to the application of SBSE techniques to Software Engineering. It assumes neither previous knowledge nor experience with Search Based Optimisation. The intention is that the tutorial will cover sufficient material to allow the reader to become productive in successfully applying search based optimisation to a chosen Software Engineering problem of interest.

1 Introduction

Search Based Software Engineering (SBSE) is the name given to a body of work in which Search Based Optimisation is applied to Software Engineering. This approach to Software Engineering has proved to be very successful and generic. It has been a subfield of software engineering for ten years [45], the past five of which have been characterised by an explosion of interest and activity [48]. New application areas within Software Engineering continue to emerge and a body of empirical evidence has now accrued that demonstrates that the search based approach is definitely here to stay.

SBSE seeks to reformulate Software Engineering problems as 'search problems' [45, 48]. This is not to be confused with textual or hypertextual searching. Rather, for Search Based Software Engineering, a search problem is one in which optimal or near optimal solutions are sought in a search space of candidate solutions, guided by a fitness function that distinguishes between better and worse solutions. The term SBSE was coined by Harman and Jones [45] in 2001, which was the first paper to advocate Search Based Optimisation as a general approach to Software Engineering, though there were other authors who had previously applied search based optimisation to aspects of Software Engineering.

B. Meyer and M. Nordio (Eds.): LASER Summer School 2008-2010, LNCS 7007, pp. 1–59, 2012.

SBSE has been applied to many fields within the general area of Software Engineering, some of which are already sufficiently mature to warrant their own surveys. For example, there are surveys and overviews, covering SBSE for requirements [111], design [78] and testing [3, 4, 65], as well as general surveys of the whole field of SBSE [21, 36, 48].

This paper does not seek to duplicate these surveys, though some material is repeated from them (with permission), where it is relevant and appropriate. Rather, this paper aims to provide those unfamiliar with SBSE with a tutorial and practical guide. The aim is that, having read this paper, the reader will be able to begin to develop SBSE solutions to a chosen software engineering problem and will be able to collect and analyse the results of the application of SBSE algorithms.

By the end of the paper, the reader (who is not assumed to have any prior knowledge of SBSE) should be in a position to prepare their own paper on SBSE. The tutorial concludes with a simple step-by-step guide to developing the necessary formulation, implementation, experimentation and results required for the first SBSE paper. The paper is primarily aimed at those who have yet to tackle this first step in publishing results on SBSE. For those who have already published on SBSE, many sections can easily be skipped, though it is hoped that the sections on advanced topics, case studies and the SBSE taxonomy (Sections 7, 8 and 9) will prove useful, even for seasoned Search Based Software Engineers.

The paper contains extensive pointers to the literature and aims to be sufficiently comprehensive, complete and self-contained that the reader should be able to move from a position of no prior knowledge of SBSE to one in which he or she is able to start to get practical results with SBSE and to consider preparing a paper for publication on these results.

The field of SBSE continues to grow rapidly. Many exciting new results and challenges regularly appear. It is hoped that this tutorial will allow many more Software Engineering researchers to explore and experiment with SBSE. We hope to see this work submitted to (and to appear in) the growing number of conferences, workshops and special issue on SBSE as well as the general software engineering literature.

The rest of the paper is organised as follows. Section 2 briefly motivates the paper by setting out some of the characteristics of SBSE that have made it well-suited to a great many Software Engineering problems, making it very widely studied. Sections 3 and 4 describe the most commonly used algorithms in SBSE and the two key ingredients of representation and fitness function. Section 5 presents a simple worked example of the application of SBSE principles in Software Engineering, using Regression Testing as an exemplar. Section 6 presents an overview of techniques commonly used to understand, analyse and interpret results from SBSE. Section 7 describes some of the more advanced techniques that can be used in SBSE to go beyond the simple world of single objectives for which we seek only to find an optimal result. Section 8 presents four case studies of previous work in SBSE, giving examples of the kinds of results obtained. These cover a variety of topics and involve very different software engineering activities,

illustrating how generic and widely applicable SBSE is to a wide range of software engineering problem domains. Section 9 presents a taxonomy of problems so far investigated in SBSE research, mapping these onto the optimisation problems that have been formulated to address these problems. Section 10 describes the next steps a researcher should consider in order to conduct (and submit for publication) their first work on SBSE. Finally, Section 11 presents potential limitations of SBSE techniques and ways to overcome them.

2 Why SBSE?

As pointed out by Harman, Mansouri and Zhang [48] Software Engineering questions are often phrased in a language that simply cries out for an optimisation-based solution. For example, a Software Engineer may well find themselves asking questions like these [48]:

1. What is the smallest set of test cases that cover all branches in this program?
2. What is the best way to structure the architecture of this system?
3. What is the set of requirements that balances software development cost and customer satisfaction?
4. What is the best allocation of resources to this software development project?
5. What is the best sequence of refactoring steps to apply to this system?

All of these questions and many more like them, can (and have been) addressed by work on SBSE [48]. In this section we briefly review some of the motivations for SBSE to give a feeling for why it is that this approach to Software Engineering has generated so much interest and activity.

1. **Generality**
   As the many SBSE surveys reveal, SBSE is very widely applicable. As explained in Section 3, we can make progress with an instance of SBSE with only two definitions: a *representation* of the problem and a *fitness function* that captures the objective or objectives to be optimised. Of course, there are few Software Engineering problems for which there will be no representation, and the readily available representations are often ready to use 'out of the box' for SBSE. Think of a Software Engineering problem. If you have no way to represent it then you cannot get started with any approach, so problem representation is a common starting point for any solution approach, not merely for SBSE. It is also likely that there is a suitable fitness function with which one could start experimentation since many software engineering metrics are readily exploitable as fitness functions [42].
2. **Robustness**
   SBSE's optimisation algorithms are robust. Often the solutions required need only to lie within some specified tolerance. Those starting out with SBSE can easily become immersed in 'parameter tuning' to get the most performance from their SBSE approach. However, one observation that almost all those who experiment will find, is that the results obtained are often robust to

the choice of these parameters. That is, while it is true that a great deal of progress and improvement can be made through tuning, one may well find that all reasonable parameter choices comfortably outperform a purely random search. Therefore, if one is the first to use a search based approach, almost any reasonable (non extreme) choice of parameters may well support progress from the current 'state of the art'.

3. **Scalability Through Parallelism**
   Search based optimisation techniques are often referred to as being 'embarrassingly parallel' because of their potential for scalability through parallel execution of fitness computations. Several SBSE authors have demonstrated that this parallelism can be exploited in SBSE work to obtain scalability through distributed computation [12, 62, 69]. Recent work has also shown how General Purpose Graphical Processing devices (GPGPUs) can be used to achieve scale up factors of up to 20 compared to single CPU-based computation [110].
4. **Re-unification**
   SBSE can also create linkages and relationships between areas in Software Engineering that would otherwise appear to be completely unrelated. For instance, the problems of Requirements Engineering and Regression Testing would appear to be entirely unrelated topics; they have their own conferences and journals and researchers in one field seldom exchange ideas with those from the other.

   However, using SBSE, a clear relationship can be seen between these two problem domains [48]. That is, as *optimisation problems* they are remarkably similar as Figure 1 illustrates: Both involve selection and prioritisation problems that share a similar structure as search problems.
5. **Direct Fitness Computation**
   In engineering disciplines such as mechanical, chemical, electrical and electronic engineering, search based optimisation has been applied for many years. However, it has been argued that it is with Software Engineering, *more than any other engineering discipline*, that search based optimisation has the highest application potential [39]. This argument is based on the nature of software as a unique and very special engineering 'material', for which even the word 'engineering material' is a slight misnomer. After all, software is the only engineering material that can only be sensed by the mind and not through any of the five senses of sight, sounds, smell, taste and touch.

   In traditional engineering optimisation, the artefact to be optimised is often simulated precisely because it is of physical material, so building mock ups for fitness computation would be prohibitively slow and expensive. By contrast, software has no physical existence; it is purely a 'virtual engineering material'. As a result, the application of search based optimisation can often be completely direct; the search is performed directly on the engineering material itself, not a simulation of a model of the real material (as with traditional engineering optimisations).

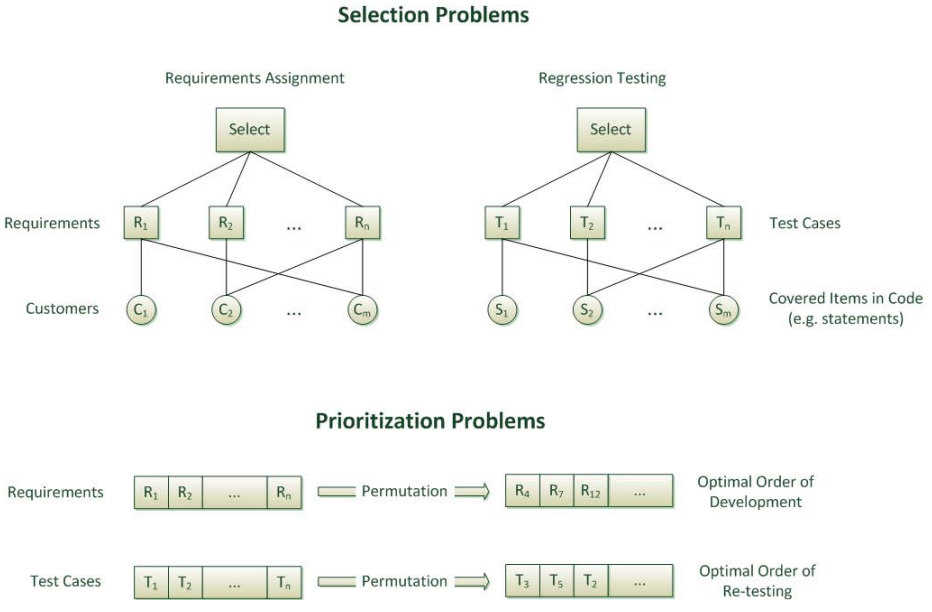


Fig. 1. Requirements Selection and Regression Testing: two different areas of Software Engineering that are Re-unified by SBSE (This example is taken from the recent survey [48]). The task of selecting requirements is closely related to the problem of selecting test cases for regression testing. We want test cases to cover code in order to achieve high fitness, whereas we want requirements to cover customer expectations. Furthermore, both regression test cases and requirements need to be prioritised. We seek to order requirements ensure that, should development be interrupted, then maximum benefit will have been achieved for the customer at the least cost. We seek to order test cases to ensure that, should testing be stopped, then maximum achievement of test objectives is achieved with minimum test effort.

3 Defining a Representation and Fitness Function

SBSE starts with only two key ingredients [36, 45]:

1. The choice of the representation of the problem.
2. The definition of the fitness function.

This simplicity makes SBSE attractive. With just these two simple ingredients the budding Search Based Software Engineer can implement search based optimisation algorithms and get results.

Typically, a software engineer will have a suitable representation for their problem. Many problems in software engineering also have software metrics associated with them that naturally form good initial candidates for fitness functions [42]. It may well be that a would-be Search Based Software Engineer will

have to hand, already, an implementation of some metric of interest. With a very little effort this can be turned into a fitness function and so the 'learning curve' and infrastructural investment required to get started with SBSE is among the lowest of any approach one is likely to encounter.

4 Commonly Used Algorithms

Random search is the simplest form of search algorithm that appears frequently in the software engineering literature. However, it does not utilise a fitness function, and is thus unguided, often failing to find globally optimal solutions (Figure 2). Higher quality solutions may be found with the aid of a fitness function, which supplies heuristic information regarding the areas of the search space which may yield better solutions and those which seem to be unfruitful to explore further. The simplest form of search algorithm using fitness information in the form of a fitness function is Hill Climbing. Hill Climbing selects a point from the search space at random. It then examines candidate solutions that are in the 'neighbourhood' of the original; i.e. solutions in the search space that are similar but differ in some aspect, or are close or some ordinal scale. If a neighbouring candidate solution is found of improved fitness, the search 'moves' to that new solution. It then explores the neighbourhood of that new candidate solution for better solutions, and so on, until the neighbourhood of the current candidate solution offers no further improvement. Such a solution is said to be *locally optimal*, and may not represent globally optimal solutions (as in Figure 3a), and so the search is often restarted in order to find even better solutions (as in Figure 3b). Hill Climbing may be restarted as many times as computing resources allow.

Pseudo-code for Hill Climbing can be seen in Figure 4. As can be seen, not only must the fitness function and the 'neighbourhood' be defined, but also the type of 'ascent strategy'. Types of ascent strategy include 'steepest ascent', where all neighbours are evaluated, with the ascending move made to the neighbour offering the greatest improvement in fitness. A 'random' or 'first' ascent strategy, on the other hand, involves the evaluation of neighbouring candidate solutions at random, and the first neighbour to offer an improvement selected for the move.

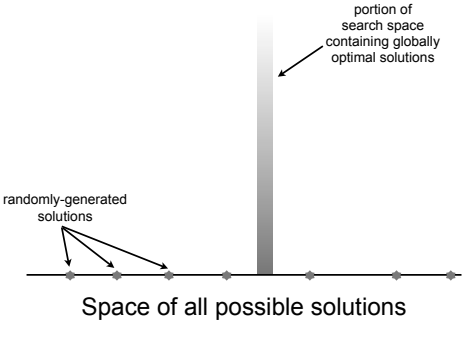
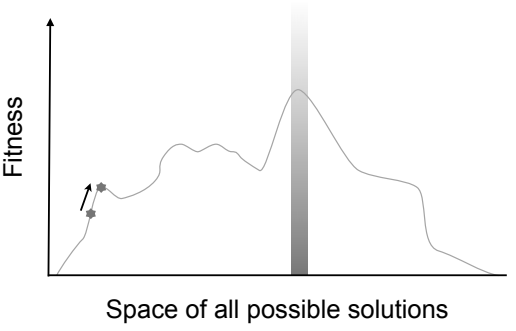
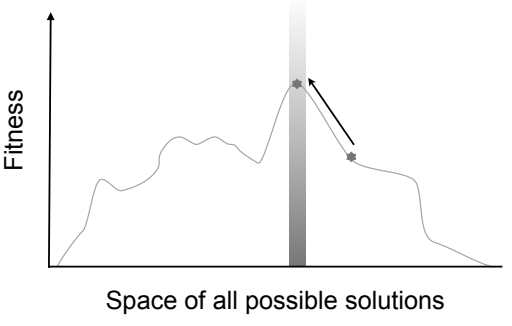


Fig. 2. Random search may fail to find optimal solutions occupying a small proportion of the overall search space (adapted from McMinn [66])



(a) A climb to a local optimum



(b) A restart resulting in a climb to the global optimum

Fig. 3. Hill Climbing seeks to improve a single solution, initially selected at random, by iteratively exploring its neighbourhood (adapted from McMinn [66])

Select a starting solution $s \in S$
Repeat
    Select $s' \in N(s)$ such that $fit(s') > fit(s)$ according to ascent strategy
    $s \leftarrow s'$
Until $fit(s) \geq fit(s'), \forall s' \in N(s)$

Fig. 4. High level description of a hill climbing algorithm, for a problem with solution space S; neighbourhood structure N; and fit, the fitness function to be maximised (adapted from McMinn [65])

Simulated Annealing (Figure 5), first proposed by Kirkpatrick et al. [56], is similar to Hill Climbing in that it too attempts to improve one solution. However, Simulated Annealing attempts to escape local optima without the need to continually restart the search. It does this by temporarily accepting candidate solutions of poorer fitness, depending on the value of a variable known as the *temperature*. Initially the temperature is high, and free movement is allowed through the search space, with poorer neighbouring solutions representing

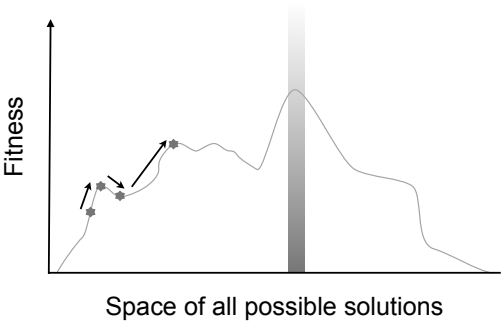


Fig. 5. Simulated Annealing also seeks to improve a single solution, but moves may be made to points in the search space of poorer fitness (adapted from McMinn [66])

potential moves along with better neighbouring solutions. As the search progresses, however, the temperature reduces, making moves to poorer solutions more and more unlikely. Eventually, *freezing point* is reached, and from this point on the search behaves identically to Hill Climbing. Pseudo-code for the Simulated Annealing algorithm can be seen in Figure 6. The probability of acceptance p of an inferior solution is calculated as $p = e^{-\frac{\delta}{t}}$, where δ is the difference in fitness value between the current solution and the neighbouring inferior solution being considered, and t is the current value of the temperature control parameter.

```
Select a starting solution s ∈ S
Select an initial temperature t > 0
Repeat
    it ← 0
    Repeat
        Select s' ∈ N(s) at random
        Δe ← fit(s) − fit(s')
        If Δe < 0
            s ← s'
        Else
            Generate random number r, 0 ≤ r < 1
            If r < e^(−δ/t) Then s ← s'
        End If
        it ← it + 1
    Until it = num_solns
    Decrease t according to cooling schedule
Until Stopping Condition Reached
```

Fig. 6. High level description of a simulated annealing algorithm, for a problem with solution space S; neighbourhood structure N; *num_solns*, the number of solutions to consider at each temperature level t; and fit, the fitness function to be maximised (adapted from McMinn [65])

'Simulated Annealing' is named so because it was inspired by the physical process of annealing; the cooling of a material in a heat bath. When a solid material is heated past its melting point and then cooled back into its solid state, the structural properties of the final material will vary depending on the rate of cooling.

Hill Climbing and Simulated Annealing are said to be *local searches*, because they operate with reference to one candidate solution at any one time, choosing 'moves' based on the neighbourhood of that candidate solution. Genetic Algorithms, on the other hand, are said to be *global searches*, sampling many points in the search space at once (Figure 7), offering more robustness to local optima. The set of candidate solutions currently under consideration is referred to as the current *population*, with each successive population considered referred to as a *generation*. Genetic Algorithms are inspired by Darwinian Evolution, in keeping with this analogy, each candidate solution is represented as a vector of components referred to as *individuals* or *chromosomes*. Typically, a Genetic Algorithm uses a binary representation, i.e. candidate solutions are encoded as strings of 1s and 0s; yet more natural representations to the problem may also be used, for example a list of floating point values.

The main loop of a Genetic Algorithm can be seen in Figure 8. The first generation is made up of randomly selected chromosomes, although the population may also be 'seeded' with selected individuals representing some domain information about the problem, which may increase the chances of the search converging on a set of highly-fit candidate solutions. Each individual in the population is then evaluated for fitness.

On the basis of fitness evaluation, certain individuals are selected to go forward to the following stages of crossover, mutation and reinsertion into the next generation. Usually selection is biased towards the fitter individuals, however the possibility of selecting weak solutions is not removed so that the search does not converge early on a set of locally optimal solutions. The very first Genetic Algorithm, proposed by Holland[1], used 'fitness-proportionate' selection, where the expected number of times an individual is selected for reproduction is proportionate to the individual's fitness in comparison with the rest of the population. However, fitness-proportionate selection has been criticised because highly-fit individuals appearing early in the progression of the search tend to dominate the selection process, leading the search to converge prematurely on one sub-area of the search space. Linear ranking [100] and tournament selection [23] have been proposed to circumvent these problems, involving algorithms where individuals are selected using relative rather than absolute fitness comparisons.

In the crossover stage, elements of each individual are recombined to form two offspring individuals. Different choices of crossover operator are available, including 'one-point' crossover, which splices two parents at a randomly-chosen position in the string to form two offspring. For example, two strings '`111`' and '`000`' may be spliced at position 2 to form two children '`100`' and '`011`'. Other

[1] This was introduced by Holland [54], though Turing had also briefly mentioned the idea of evolution as a computational metaphor [94].

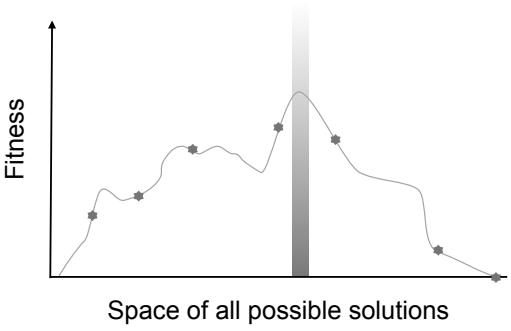


Fig. 7. Genetic Algorithms are global searches, taking account of several points in the search space at once (adapted from McMinn [66])

Randomly generate or seed initial population P
Repeat
    Evaluate fitness of each individual in P
    Select parents from P according to selection mechanism
    Recombine parents to form new offspring
    Construct new population P' from parents and offspring
    Mutate P'
    $P \leftarrow P'$
Until Stopping Condition Reached

Fig. 8. High level description of a Genetic Algorithm, adapted from McMinn [65]

operators may recombine using multiple crossover points, while 'uniform' crossover treats every position as a potential crossover point.

Subsequently, elements of the newly-created chromosomes are mutated at random, with the aim of diversifying the search into new areas of the search space. For GAs operating on binary representation, mutation usually involves randomly flipping bits of the chromosome. Finally, the next generation of the population is chosen in the 'reinsertion' phase, and the new individuals are evaluated for fitness. The GA continues in this loop until it finds a solution known to be globally optimal, or the resources allocated to it (typically a time limit or a certain budget of fitness evaluations) are exhausted. Whitley's tutorial papers [101, 102] offer a further excellent introductory material for getting starting with Genetic Algorithms in Search Based Software Engineering.

5 Getting the First Result: A Simple Example for Regression Testing

This section presents an application of a search-based approach to the Test Case Prioritisation (TCP) in regression testing, illustrating the steps that are

necessary to obtain the first set of results. This makes concrete the concepts of representation, fitness function and search based algorithm (and their operators) introduced in the previous sections. First, let us clarify what we mean by TCP.

Regression testing is a testing activity that is performed to gain confidence that the recent modifications to the System Under Test (SUT), e.g. bug patches or new features, did not interfere with existing functionalities [108]. The simplest way to ensure this is to execute all available tests; this is often called *retest-all* method. However, as the software evolves, the test suite grows too, eventually making it prohibitively expensive to adopt the retest-all approach. Many techniques have been developed to deal with the cost of regression testing.

Test Case Prioritisation represents a group of techniques that particularly deal with the permutations of tests in regression test suites [28, 108]. The assumption behind these techniques is that, because of the limited resources, it may not be possible to execute the entire regression test suite. The intuition behind Test Case Prioritisation techniques is that more important tests should be executed earlier. In the context of regression testing, the 'important' tests are the ones that detect regression faults. That is, the aim of Test Case Prioritisation is to maximise *earlier fault detection rate*. More formally, it is defined as follows:

Definition 1. ***Test Case Prioritisation Problem***

Given: *A test suite, T, the set of permutations of T, PT, and a function from PT to real numbers, $f : PT \rightarrow \mathbb{R}$.*

Problem: *To find $T' \in PT$ such that $(\forall T'')(T'' \in PT)(T'' \neq T')[f(T') \geq f(T'')]$.*

Ideally, the function f should be a mapping from tests to their fault detection capability. However, whether a test detects some faults or not is only known *after* its execution. In practice, a function f that is a surrogate to the fault detection capability of tests is used. Structural coverage is one of the most popular choices: the permutation of tests that achieves structural coverage as early as possible is thought to maximise the chance of early fault detection.

5.1 Representation

At its core, TCP as a search problem is an optimisation in a permutation space similar to the Travelling Salesman Problem (TSP), for which many advanced representation schemes have been developed. Here we will focus on the most basic form of representation. The set of all possible candidate solutions is the set of all possible permutations of tests in the regression test suite. If the regression test suite contains n tests, the representation takes the form of a vector with n elements. For example, Figure 9 shows one possible candidate solution for TCP with size n, i.e. with a regression test suite that contains 6 tests, $\{t_0, \ldots, t_5\}$.

Depending on the choice of the search algorithm, the next step is either to define the neighbouring solutions of a given solution (local search) or to define the genetic operators (genetic algorithm).

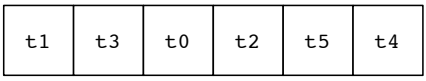


Fig. 9. One possible candidate solution for TCP with a regression test suite with 6 tests, $\{t_0, \ldots, t_5\}$

Neighbouring Solutions. Unless the characteristics of the search landscape is known, it is recommended that the neighbouring solutions of a given solution for a local search algorithm is generated by making the smallest possible changes to the given solution. This allows the human engineer to observe and understand the features of the search landscape.

It is also important to define the neighbouring solutions in a way that produces a manageable number of neighbours. For example, if the set of neighbouring solutions for TCP of size n is defined as the set of all permutations that can be generated by swapping two tests, there would be $n(n-1)$ neighbouring solutions. However, if we only consider swapping adjacent tests, there would be $n-1$. If the fitness evaluation is expensive, i.e. takes non-trivial time, controlling the size of the neighbourhood may affect the efficiency of the search algorithm significantly.

Genetic Operators. The following is a set of simple genetic operators that can be defined over permutation-based representations.

- **Selection:** Selection operators tend to be relatively independent of the choice of representation. It is more closely related to the design of the fitness function. One widely used approach that is also recommended as the first step is n-way tournament selection. First, randomly sample n solutions from the population. Out of this sample, pick the fittest individual solution. Repeat once again to select a pair of solutions for reproduction.
- **Crossover:** Unlike selection operators, crossover operators are directly linked to the structure of the representation of solutions. Here, we use the crossover operator following Antoniol et al. [6] to generate, from parent solutions p_1 and p_2, the offspring solutions o_1 and o_2:
  1. Pick a random number k $(1 \leq k \leq n)$
  2. The first k elements of p_1 become the first k elements of o_1.
  3. The last $n - k$ elements of o_1 are the sequence of $n - k$ elements that remain when the k elements selected from p_1 are taken from p_2, as illustrated in Figure 10.
  4. o_2 is generated similarly, composed of the first $n - k$ elements of p_2 and the remaining k elements of p_1.
- **Mutation:** Similarly to defining the neighbouring solutions for local search algorithms, it is recommended that, initially, mutation operators are defined to introduce relatively small changes to individual solutions. For example, we can swap the position of two randomly selected tests.

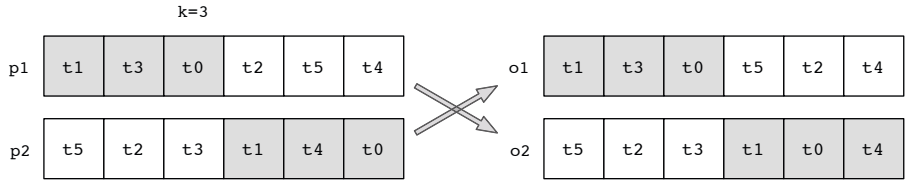


Fig. 10. Illustration of crossover operator for permutation-based representations following Antoniol et al.

5.2 Fitness Function

The recommended first step to design the fitness function is to look for an existing metric that measures the quality we are optimising for. If one exists, it often provides not only a quick and easy way to evaluate the search-based approach to the problem but also a channel to compare the results to other existing techniques.

The metric that is widely used to evaluate the effectiveness of TCP techniques is Average Percentage of Faults Detected (APFD) [28]. Higher APFD values mean that faults are detected earlier in testing. Suppose that, as the testing progresses, we plot the percentage of detected faults against the number of tests executed so far: intuitively, APFD would be the area behind the plot.

However, calculation of APFD requires the knowledge of which tests detected which faults. As explained in Section 5, the use of this knowledge defies the purpose of the prioritisation because fault detection information is not available until all tests are executed. This forces us to turn to the widely used surrogate, structural coverage. For example, Average Percentage of Blocks Covered (APBC) is calculated in a similar way to APFD but, instead of percentage of detected faults, percentage of blocks covered so far is used. In regression testing scenarios, the coverage information of tests are often available from the previous iteration of testing. While the recent modification that we are testing against might have made the existing coverage information imprecise, it is often *good enough* to provide guidance for prioritisation, especially when regression testing is performed reasonably frequently.

5.3 Putting It All Together

The representation of solutions and the fitness function are the only problem-specific components in the overall architecture of SBSE approach in Figure 11. It is recommended that these problem specific components are clearly separated from the search algorithm itself: the separation not only makes it easier to reuse the search algorithms (that are problem independent) but also helps testing and debugging of the overall approach (repeatedly used implementations of search algorithms can provide higher assurance).

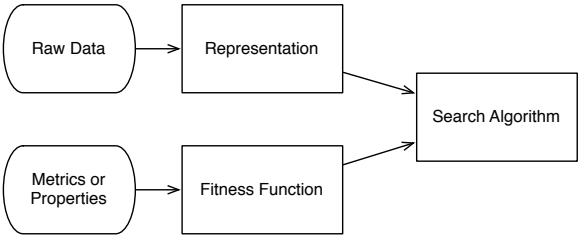


Fig. 11. Overall Architecture of SBSE Approach

6 Understanding Your Results

6.1 Fair Comparison

Due to the stochastic nature of optimisation algorithms, searches must be repeated several times in order to mitigate against the effects of random variation. In the literature, experiments are typically repeated 30-50 times.

When comparing two algorithms, the best fitness values obtained by the searches concerned are an obvious indicator to how well the optimisation process performed. However, in order to ensure a fair comparison, it is important to establish the amount of effort expended by each search algorithm, to find those solutions. This effort is commonly measured by logging the *number of fitness evaluations* that were performed. For example, it could be that an algorithm found a solution with a better fitness value, but did so because it was afforded a higher number of trials in which to obtain it. Or, there could be trade-offs, for example search A may find a solution of good fitness early in the search, but fail to improve it, yet search B can find solutions of slightly better fitness, but requiring many more fitness evaluations in which to discover it. When a certain level of fitness is obtained by more than one search algorithm, the *average number of fitness evaluations* over the different runs of the experiments by each algorithm is used to measure the cost of the algorithm in obtaining that fitness, or to put it another way, its relative *efficiency*.

For some types of problem, e.g. test data generation, there is a specific goal that must be attained by the search; for example the discovery of test data to execute a particular branch. In such cases, merely 'good' solutions of high fitness are not enough - a solution with a certain very high fitness value must be obtained, or the goal of the search will not be attained. In such cases, best fitness is no longer as an important measure as the *success rate*, a percentage reflecting the number of times the goal of the search was achieved over the repetitions of the experiment. The success rate gives an idea of how *effective* the search was at achieving its aim.

6.2 Elementary Statistical Analysis

The last section introduced some *descriptive* statistics for use in Search Based Software Engineering experiments, but also *inferential* statistics may be applied

to discern whether one set of experiments are significantly different in some aspect from another.

Suppose there are two approaches to address a problem in SBSE, each of which involves the application of some search based algorithm to a set of problem instances. We collect results for the application of both algorithms, A and B, and we notice that, over a series of runs of our experiment, Algorithm A tends to perform better than Algorithm B. The performance we have in mind, may take many forms. It may be that Algorithm A is faster than B, or that after a certain number of fitness evaluations it has achieved a higher fitness value, or a higher average fitness. Alternatively, there may be some other measurement that we can make about which we notice a difference in performance that we believe is worth reporting.

In such cases, SBSE researchers tend to rely on inferential statistics as a means of addressing the inherently stochastic nature of search based algorithms. That is, we may notice that the mean fitness achieved by Algorithm A is higher than that of Algorithm B after 10 executions of each, but how can we be *sure* that this is not merely an observation arrived at by *chance*? It is to answer precisely these kinds of question that statistical hypothesis testing is used in the experimental sciences, and SBSE is no exception.

A complete explanation of the issues and techniques that can be used in applying inferential statistics in SBSE is beyond the scope of this tutorial. However, there has been a recent paper on the topic of statistical testing of randomised algorithms by Arcuri and Briand [8], which provides more detail. In this section we provide an overview of some of the key points of concern.

The typical scenario with which we are concerned is one in which we want to explore the likelihood that our experiments found that Algorithm A outperforms Algorithm B purely by chance. Usually we wish to be in a position to make a claim that we have evidence that suggests that Algorithm A is better than Algorithm B. For example, as a sanity check, we may wish to show that our SBSE technique comfortably outperforms a random search. But what do we mean by 'comfortably outperforms'?

In order to investigate this kind of question we set a threshold on the degree of chance that we find acceptable. Typically, in the experimental sciences, this level is chosen to be either 1% or 5%. That is, we will have either a less than 1 in 100 or a less than 5 in 100 chance of believing that Algorithm A outperforms Algorithm B based on a set of executions when in fact it does not. This is the chance of making a so-called 'Type I' error. It would lead to us concluding that some Algorithm A was better than Algorithm B when, in fact, it was not.

If we choose a threshold for error of 5% then we have a 95% confidence level in our conclusion based on our sample of the population of all possible executions of the algorithm. That is, we are '95% sure that we can claim that Algorithm A really is better than Algorithm B'. Unpacking this claim a little, what we find is that there is a population involved. This is the population of all possible runs of the algorithm in question. For each run we may get different behaviour due

to the stochastic nature of the algorithm and so we are not in a position to say exactly what the value obtained will be. Rather, we can give a range of values.

However, it is almost always impractical to perform all possible runs and so we have to sample. Our '95% confidence claim' is that we are 95% confident that the evidence provided by our sample allows us to infer a conclusion about the algorithm's performance on the whole population. This is why this branch of statistics is referred to as 'inferential statistics'; we *infer* properties of a whole population based on a sample.

Unfortunately a great deal of 'ritualistic' behaviour has grown up around the experimental sciences, in part, resulting for an inadequate understanding of the underlying statistics. One aspect of this ritual is found in the choice of a suitable confidence level. If we are comparing some new SBSE approach to the state of the art, then we are asking a question as to whether the new approach is worthy of consideration. In such a situation we may be happy with a 1 in 10 chance of a Type I error (and could set the confidence level, accordingly, to be 90%). The consequences of considering a move from the status quo may not be so great.

However, if we are considering whether to use a potently fatal drug on a patient who may otherwise survive we might want a much higher confidence that the drug would, indeed, improve the health of the patent over the status quo (no treatment). For this reason it is important to think about what level of confidence is suitable for the problem in hand.

The statistical test we perform will result in a p-value. The p-value is the chance that a Type I error has occurred. That is, we notice that a sample of runs produces a higher mean result for a measurement of interest for Algorithm A than for Algorithm B. We wish to reject the so-called 'null hypothesis'; the hypothesis that the population of all executions of Algorithm A is no different to that of Algorithm B. To do this we perform an inferential statistical test. If all the assumptions of the test are met and the sample of runs we have is unbiased then the p-value we obtain indicates the chance that the populations of runs of Algorithm A and Algorithm B are identical given the evidence we have from the sample. For instance a p-value equal to or lower than 0.05 indicates that we have satisfied the traditional (and somewhat ritualistic) 95% confidence level test. More precisely, the chance of committing a Type I error is p.

This raises the question of how large a sample we should choose. The sample size is related to the statistical power of our experiment. If we have too small a sample then we may obtain high p-values and incorrectly conclude that there is no significant difference between the two algorithms we are considering. This is a so-called Type II error; we incorrectly accept the null hypothesis when it is, in fact, false. In our case it would mean that we would incorrectly believe Algorithm A to be no better than Algorithm B. More precisely, we would conclude, *correctly*, that we have no evidence to claim that Algorithm A is significantly better than Algorithm B at the chosen conference level. However, had we chosen a larger sample, we may have had just such evidence. In general, all else being equal, the larger the sample we choose the less likely we are to commit a Type II error. This is why researchers prefer larger sample sizes where this is feasible.

There is another element of ritual for which some weariness is appropriate: the choice of a suitable statistical test. One of the most commonalty performed tests in work on search based algorithms in general (though not necessarily SBSE in particular) is the well-known t test. Almost all statistical packages support it and it is often available at the touch of a button. Unfortunately, the t test makes assumptions about the distribution of the data. These assumptions may not be borne out in practice thereby increasing the chance of a Type I error. In some senses a type I error is worse than a Type II error, because it may lead to the publication of false claims, whereas a Type I error will most likely lead to researcher disappointment at the lack of evidence to support publishable results.

To address this potential problem with parametric inferential statistics SBSE researchers often use nonparametric statistical tests. Non-parametric tests make fewer assumptions about the distribution of the data. As such, these tests are weaker (they have less power) and may lead to the false acceptance of the null hypothesis for the same sample size (a Type II error), when used in place of a more powerful parametric test that is able to reject the null hypothesis. However, since the parametric tests make assumptions about the distribution, should these assumptions prove to be false, then the rejection of the null hypothesis by a parametric test may be an artefact of the false assumptions; a form of Type I error.

It is important to remember that all inferential statistical techniques are founded on probability theory. To the traditional computer scientist, particularly those raised on an intellectual diet consisting exclusively of formal methods and discrete mathematics, this reliance on probability may be as unsettling as quantum mechanics was to the traditional world of physics. However, as engineers, the reliance on a confidence level is little more than an acceptance of a certain 'tolerance' and is quite natural and acceptable.

This appreciation of the probability-theoretic foundations of inferential statistics rather than a merely ritualistic application of 'prescribed tests' is important if the researcher is to avoid mistakes. For example, armed with a non parametric test and a confidence internal of 95% the researcher may embark on a misguided 'fishing expedition' to find a variant of Algorithm A that outperforms Algorithm B. Suppose 5 independent variants of Algorithm A are experimented with and, on each occasion, a comparison is made with Algorithm B using an inferential statistical test. If variant 3 produces a p-value of 0.05, while the others do not it would be a mistake to conclude that *at the 95% confidence level* Algorithm A (variant 3) is better than Algorithm B.

Rather, we would have to find that Algorithm A variant 3 had a p-value lower than 0.05/5; by repeating the same test 5 times, we raise the confidence required for each test from 0.05 to 0.01 to retain the same overall confidence. This is known as a 'Bonferroni correction'. To see why it is necessary, suppose we have 20 variants of Algorithms A. What would be the expected likelihood that one of these would, *by chance*, have a p-value equal or lower than 0.05 in a world where none of the variants is, in fact, any different from Algorithm B? If we repeat a statistical test sufficiently many times without a correction to the

confidence level, then we are increasingly likely to commit a Type I error. This situation is amusingly captured by an `xkcd` cartoon [73].

Sometimes, we find ourselves comparing, not vales of measurements, but the success rates of searches. Comparison of success rates using inferential statistics requires a categorical approach, since a search goal is either fulfilled or not. For this Fisher's Exact test is a useful statistical measure. This is another non-parametric test. For investigative of correlations, researchers use Spearman and Pearson correlation analysis. These tests can be useful to explore the degree to which increases in one factor are correlated to another, but it is important to understand that correlations does not, of course, entail causality.

7 More Advanced Techniques

Much has been achieved in SBSE using only a single fitness function, a simple representation of the problem and a simple search technique (such as hill climbing). It is recommended that, as a first exploration of SBSE, the first experiments should concern a single fitness function, a simple representation and a simple search technique. However, once results have been obtained and the approach is believed to have potential, for example, it is found to outperform random search, then it is natural to turn one's attention to more advanced techniques and problem characterisations.

This section considers four exciting ways in which the initial set of results can be developed, using more advanced techniques that may better model the real world scenario and may also help to extend the range and type of results obtained and the applicability of the overall SBSE approach for the Software Engineering problem in hand.

7.1 Multiple Objectives

Though excellent results can be obtained with a single objective, many real world Software Engineering problems are multiple objective problems. The objectives that have to be optimised are often in competition with one another and may be contradictory; we may find ourselves trying to balance the different optimisation objectives of several different goals.

One approach to handle such scenarios is the use of Pareto optimal SBSE, in which several optimisation objectives are combined, but without needing to decide which take precedence over the others. This approach is described in more detail elsewhere [48] and was first proposed as the 'best' way to handle multiple objectives for all SBSE problems by Harman in 2007 [36]. Since then, there has been a rapid uptake of Pareto optimal SBSE to requirements [27, 31, 84, 90, 113], planning [5, 98], design [17, 88, 95], coding [9, 99], testing [33, 35, 47, 76, 90, 96, 107], and refactoring [52].

Suppose a problem is to be solved that has n fitness functions, $f_1, \ldots, f_n$ that take some vector of parameters $\overline{x}$. Pareto optimality combines a set of measurements, f_i, into a single ordinal scale metric, F, as follows:

$$F(\overline{x_1}) > F(\overline{x_2})$$
$$\Leftrightarrow$$
$$\forall i.f_i(\overline{x_1}) \geq f_i(\overline{x_2}) \quad \wedge \quad \exists i.f_i(\overline{x_1}) > f_i(\overline{x_2})$$

Under Pareto optimality, one solution is better than another if it is better according to at least one of the individual fitness functions and no worse according to all of the others. Under the Pareto interpretation of combined fitness, no overall fitness improvement occurs no matter how much almost all of the fitness functions improve, should they do so at the slightest expense of any one of their number. The use of Pareto optimality is an alternative to simply aggregating fitness using a weighted sum of the n fitness functions.

When searching for solutions to a problem using Pareto optimality, the search yields a set of solutions that are non–dominated. That is, each member of the non-dominated set is no worse than any of the others in the set, but also cannot be said to be better. Any set of non–dominated solutions forms a Pareto front.

Consider Figure 12, which depicts the computation of Pareto optimality for two imaginary fitness functions (Objective 1 and Objective 2). The longer the search algorithm is run the better the approximation becomes to the real Pareto front. In the figure, points $S1$, $S2$ and $S3$ lie on the Pareto front, while $S4$ and $S5$ are dominated.

Pareto optimality has many advantages. Should a single solution be required, then coefficients can be re-introduced in order to distinguish among the non–dominated set at the current Pareto front. However, by refusing to conflate the individual fitness functions into a single aggregate, the search may consider solutions that may be overlooked by search guided by aggregate fitness. The approximation of the Pareto front is also a useful analysis tool in itself. For example, it may contain 'knee points', where a small change in one fitness is accompanied by a large change in another. These knee points denote interesting parts of the solution space that warrant closer investigation.

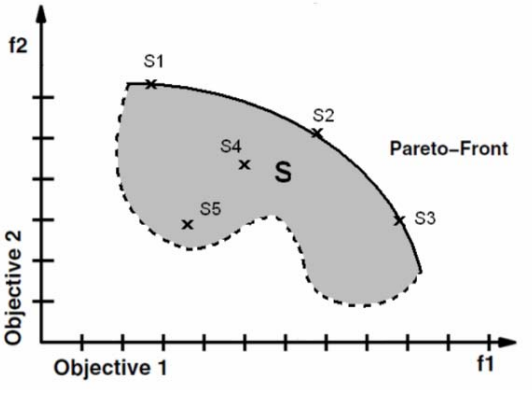


Fig. 12. Pareto Optimality and Pareto Fronts (taken from the survey by Harman et al. [48])

7.2 Co Evolution

In Co–Evolutionary Computation, two or more populations of solutions evolve simultaneously with the fitness of each depending upon the current population of the other. Adamopoulos et al. [2] were the first to suggest the application of co-evolution to an SBSE problem, using it to evolve sets of mutants and sets of test cases, where the test cases act as predators and the mutants as their prey. Arcuri and Yao [10] use co-evolution to evolve programs and their test data from specifications using co-evolution.

Arcuri and Yao [11] also developed a co-evolutionary model of bug fixing, in which one population essentially seeks out patches that are able to pass test cases, while test cases can be produced from an oracle in an attempt to find the shortcomings of a current population of proposed patches. In this way the patch is the prey, while the test cases, once again, act as predators. The approach assumes the existence of a specification to act the oracle.

Many aspects of Software Engineering problems lend themselves to a co-evolutionary model of optimisation because software systems are complex and rich in potential populations that could be productively co-evolved (using both competitive and co-operative co-evolution). For example: components, agents, stakeholder behaviour models, designs, cognitive models, requirements, test cases, use cases and management plans are all important aspects of software systems for which optimisation is an important concern. Though all of these may not occur in the same system, they are all the subject of change. If a suitable fitness function be found, the SBSE can be used to co-evolve solutions.

Where two such populations are already being evolved in isolation using SBSE, but participate in the same overall software system, it would seem a logical 'next step', to seek to evolve these populations together; the fitness of one is likely to have an impact on the fitness of another, so evolution in isolation may not be capable of locating the best solutions.

7.3 SBSE as Decision Support

SBSE has been most widely used to find solutions to complex and demanding software engineering problems, such as sets of test data that meet test adequacy goals or sequences of transformations that refactor a program or modularisation boundaries that best balance the trade off between cohesion and coupling. However, in many other situations it is not the actual solutions found that are the most interesting nor the most important aspects of SBSE.

Rather, the value of the approach lies in the insight that is gained through the analysis inherent in the automated search process and the way in which its results capture properties of the structure of software engineering solutions. SBSE can be applied to situations in which the human will decide on the solution to be adopted, but the search process can provide insight to help guide the decision maker.

This insight agenda, in which SBSE is used to gain insights and to provide decision support to the software engineering decision maker has found natural

resonance and applicability when used in the early aspects of the software engineering lifecycle, where the decisions made can have far–reaching implications.

For instance, addressing the need for negotiation and mediation in requirements engineering decision making, Finkelstein et al. [31] explored the use of different notions of fairness to explore the space of requirements assignments that can be said to be fair according to multiple definitions of 'fairness'. Saliu and Ruhe [84] used a Pareto optimal approach to explore the balance of concerns between requirements at different levels of abstraction, while Zhang et al, showed how SBSE could be used to explore the tradeoff among the different stakeholders in requirements assignment problems [112].

Many of the values used to define a problem for optimisation come from estimates. This is particularly the case in the early stages of the software engineering lifecycle, where the values available necessarily come from the estimates made by decision makers. In these situations it is not optimal solutions that the decision maker requires, so much as guidance on which of the estimates are most likely to affect the solutions. Ren et al. [46] used this observation to define an SBSE approach to requirements sensitivity analysis, in which the gaol is to identify the requirements and budgets for which the managers' estimates of requirement cost and value have most impact. For these *sensitive* requirements and budgets, more care is required. In this way SBSE has been used as a way to provide *sensitivity analysis*, rather than necessarily providing a proposed set of requirement assignments.

Similarly, in project planning, the manager bases his or her decisions on estimates of work package duration and these estimates are notoriously unreliable. Antoniol et al. [5] used this observation to explore the trade off between the completion time of a software project plan and the risk over overruns due to misestimation. This was a Pareto efficient, bi–objective approach, in which the two objectives were the completion time and the risk (measured in terms of overrun due to misestimation). Using their approach, Antoniol et al., demonstrated that a decision maker could identify safe budgets for which completion times could be more assured.

Though most of the work on decision support through SBSE has been conducted at the early stages of the lifecycle, there are still opportunities for using SBSE to gain insight at later stages in the lifecycle. For example, White et al. [99] used a bi-objective Pareto optimal approach to explore the trade off between power consumption and functionality, demonstrating that it was possible to find knee points on the Pareto front for which a small loss of functionality could result in a high degree of improved power efficiency.

As can be seen from these examples, SBSE is not merely a research programme in which one seeks to 'solve' software engineering problems; it is a rich source of insight and decision support. This is a research agenda for SBSE that Harman has developed through a series of keynotes and invited papers, suggesting SBSE as a source of additional insight and an approach to decision support for predictive modelling [38], cognitive aspects of program understanding [37], multiple objective regression testing [40] and program transformation and refactoring [41].

7.4 Augmenting with Other Non SBSE Techniques

Often it is beneficial to augment search algorithms with other techniques, such as clustering or static analysis of source code. There is no hard rules for augmentation: different non-SBSE techniques can be considered appropriate depending on the context and challenge that are unique to the given software engineering problem. This section illustrates how some widely used non-SBSE techniques can help the SBSE approach.

Clustering. Clustering is a process that partitions objects into different subsets so that objects in each group share common properties. The clustering criterion determines which properties are used to measure the commonality. It is often an effective way to reduce the size of the problem and, therefore, the size of the search space: objects in the same cluster can be replaced by a single representative object from the cluster, resulting in reduced problem size. It has been successfully applied when the human is in the loop [109].

Static Analysis. For search-based test data generation approaches, it is common that the fitness evaluation involves the program source code. Various static analysis techniques can improve the effectiveness and the efficiency of code-related SBSE techniques. Program slicing has been successfully used to reduce the search space for automated test data generation [43]. Program transformation techniques have been applied so that search-based test data generation techniques can cope with flag variables [15].

Hybridisation. While hybridising different search algorithms are certainly possible, hybridisation with non-SBSE techniques can also be beneficial. Greedy approximation has been used to *inject* solutions into MOEA so that MOEA can reach the region close to the true Pareto front much faster [107]. Some of more sophisticated forms of hybridisation use non-SBSE techniques as part of fitness evaluation [105].

8 Case Studies

This section introduces four case studies to provide the reader with a range of examples of SBSE application in software engineering. The case studies are chosen to represent a wide range of topics, illustrating the way in which SBSE is highly applicable to Software Engineering problem; with just a suitable representation, fitness function and a choice of algorithm it is possible to apply SBSE to the full spectrum of SBSE activities and problems and to obtain interesting and potentially valuable results. The case studies cover early lifecycle activities such as effort estimation and requirements assignment through test case generation to regression testing, exemplifying the breadth of applications to which SBSE has already been put.

8.1 Case Study: Multi-objective Test Suite Minimisation

Let us consider another class of regression testing techniques that is different from Test Case Prioritisation studied in Section 5: test suite minimisation. Prioritisation techniques aim to generate an ideal test execution order; minimisation techniques aim to reduce the size of the regression test suite when the regression test suite of an existing software system grows to such an extent that it may no longer be feasible to execute the entire test suite [80]. In order to reduce the size of the test suite, any *redundant* test cases in the test suite need to be identified and removed.

Regression Testing requires optimisation because of the problem posed by large data sets. That is, organisations with good testing policies quickly accrue large pools of test data. For example, one of the regression test suites studied in this paper is also used for a smoke-test by IBM for one of its middleware products and takes over 4 hours if executed in its entirety. However, a typical smoke-test can be allocated only 1 hour maximum, forcing the engineer either to select a set of test cases from the available pool or to prioritise the order in which the test cases are considered.

The cost of this selection or prioritisation may not be amortised if the engineer wants to apply the process with every iteration in order to reflect the most recent test history or to use the whole test suite more evenly. However, without optimisation, the engineer will simply run out of time to complete the task. As a result, the engineer may have failed to execute the most optimal set of test cases when time runs out, reducing fault detection capabilities and thereby harming the effectiveness of the smoke test.

One widely accepted criterion for redundancy is defined in relation to the coverage achieved by test cases [16, 20, 53, 74, 81]. If the test coverage achieved by test case t_1 is a subset of the test coverage achieved by test case t_2, it can be said that the execution of t_1 is redundant as long as t_2 is also executed. The aim of test suite minimisation is to obtain the smallest subset of test cases that are not redundant with respect to a set of test requirements. More formally, test suite minimisation problem can be defined as follows [108]:

Definition 2. ***Test Suite Minimisation Problem***

Given: *A test suite of n tests, T, a set of m test goals $\{r_1, \ldots, r_m\}$, that must be satisfied to provide the desired 'adequate' testing of the program, and subsets of T, T_is, one associated with each of the r_is such that any one of the test cases t_j belonging to T_i can be used to achieve requirement r_i.*

Problem: *Find a representative set, T', of test cases from T that satisfies all r_is.*

The testing criterion is satisfied when every test-case requirement in $\{r_1, \ldots, r_m\}$ is satisfied. A test-case requirement, r_i, is satisfied by any test case, t_j, that belongs to T_i, a subset of T. Therefore, the representative set of test cases is the

hitting set of T_is. Furthermore, in order to maximise the effect of minimisation, T' should be the minimal hitting set of T_is. The minimal hitting-set problem is an NP-complete problem as is the dual problem of the minimal set cover problem [34].

The NP-hardness of the problem encouraged the use of heuristics and meta-heuristics. The greedy approach [74] as well as other heuristics for minimal hitting set and set cover problem [20, 53] have been applied to test suite minimisation but these approaches were not cost-cognisant and only dealt with a single objective (test coverage). With the single-objective problem formulation, the solution to the test suite minimisation problem is one subset of test cases that maximises the test coverage with minimum redundancy.

Later, the problem was reformulated as a multi-objective optimisation problem [106]. Since the greedy algorithm does not cope with multiple objectives very well, Multi-Objective Evolutionary Algorithms (MOEAs) have been applied to the multi-objective formulation of the test suite minimisation [63, 106]. The case study presents the multi-objective formulation of test suite minimisation introduced by Yoo and Harman [106].

Representation. Test suite minimisation is at its core a set-cover problem; the main decision is whether to include a specific test into the minimised subset or not. Therefore, we use the binary string representation. For a test suite with n tests, $\{t_1, \ldots, t_n\}$, the representation is a binary string of length n: the i-th digit is 1 if t_i is to be included in the subset and 0 otherwise. Binary tournament selection, single-point crossover and single bit-flip mutation genetic operators were used for MOEAs.

Fitness Function. Three different objectives were considered: structural coverage, fault history coverage and execution cost. Structural coverage of a given candidate solution is simply the structural coverage achieved collectively by all the tests that are selected by the candidate solution (i.e. their corresponding bits are set to 1). This information is often available from the previous iteration of regression testing. This objective is to be maximised.

Fault history coverage is included to compliment structural coverage metric because achieving coverage may not always increase fault detection capability. We collect all known previous faults and calculate *fault coverage* for each candidate solution by counting how many of the previous faults could have been detected by the candidate solution. The underlying assumption is that a test that has detected faults in the past may have a higher chance of detecting faults in the new version. This objective is to be maximised.

The final objective is execution cost. Without considering the cost, the simplest way to maximise the other two objectives is to select the entire test suite. By trying to optimise for the cost, it is possible to obtain the trade-off between structural/fault history coverage and the cost of achieving them. The execution cost of each test is measured using a widely-used profiling tool called `valgrind`.

Algorithm. A well known MOEA by Deb et al. [24], NSGA-II, was used for the case study. Pareto optimality is used in the process of selecting individuals. This leads to the problem of selecting one individual out of a non-dominated pair. NSGA-II uses the concept of crowding distance to make this decision; crowding distance measures how far away an individual is from the rest of the population. NSGA-II tries to achieve a wider Pareto frontier by selecting individuals that are far from the others. NSGA-II is based on elitism; it performs the non-dominated sorting in each generation in order to preserve the individuals on the current Pareto frontier into the next generation.

The widely used single-objective approximation for set cover problem is greedy algorithm. The only way to deal with the chosen three objectives is to take the weighted sum of each coverage metric per time, i.e.:

Results. Figure 13 shows the results for the three objective test suite minimisation for a test suite of a program called `space`, which is taken from Software Infrastructure Repository (SIR). The 3D plots display the solutions produced by the weighted-sum additional greedy algorithm (depicted by + symbols connected with a line), and the reference Pareto front (depicted by × symbols). The reference Pareto front contains all non-dominated solutions from the combined results of weighted-sum greedy approach and NSGA-II approach. While the weighted-sum greedy approach produces solutions that are not dominated, it can be seen that NSGA-II produces a much richer set of solutions that explore wider area of the trade-off surface.

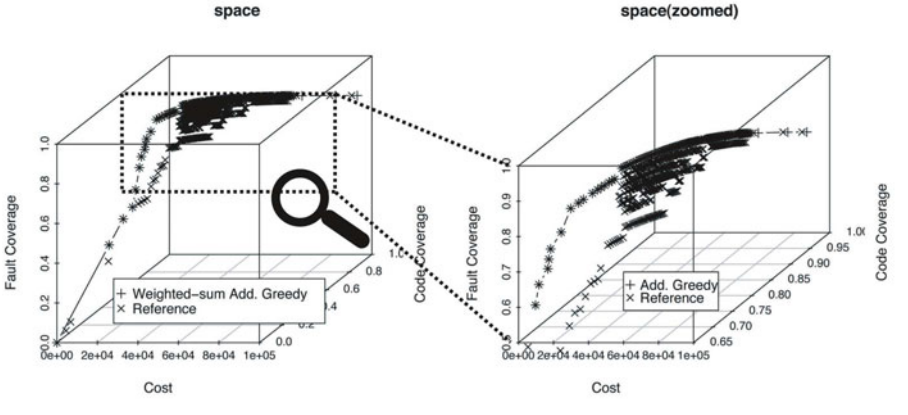


Fig. 13. A plot of 3-dimensional Pareto-front from multi-objective test suite minimisation for program `space` from European Space Agency, taken from Yoo and Harman [106]

8.2 Case Study: Requirements Analysis

Selecting a set of software requirements for the release of the next version of a software system is a demanding decision procedure. The problem of choosing the

optimal set of requirements to include in the next release of a software system has become known as the Next Release Problem (NRP) [13, 113] and the activity of planning for requirement inclusion and exclusion has become known as release planning [82, 84].

The NRP deals with the selecting a subset of requirements based on their desirability (e.g. the expected revenue) while subject to constraints such as a limited budget [13]. The original formulation of NRP by Bagnall et al. [13] considered maximising the customer satisfaction (by inclusion of their demanded requirements in the next version) while not exceeding the company's budget.

More formally, let $C = \{c_1, \ldots, c_m\}$ be the set of m customers whose requirements are to be considered for the next release. The set of n possible software requirements is denoted by $R = \{r_1, \ldots, r_n\}$. It is assumed that all requirements are independent, i.e. no requirement depends on others[2]. Finally, let $cost = [cost_1, \ldots, cost_n]$ be the cost vector for the requirements in R: $cost_i$ is the associate cost to fulfil the requirement r_i.

We also assume that each customer has a degree of importance for the company. The set of relative weights associated with each customer $c_j (1 \leq j \leq m)$ is denoted by $W = \{w_1, \ldots, w_m\}$, where $w_j \in [0, 1]$ and $\sum_{j=1}^{m} w_j = 1$. Finally, it is assumed that all requirements are not equally important for a given customer. The level of satisfaction for a given customer depends on the requirements that are satisfied in the next release of the software. Each customer $c_j (1 \leq j \leq m)$ assigns a value to requirement $r_i (1 \leq i \leq n)$ denoted by $value(r_i, c_j)$ where $value(r_i, c_j) > 0$ if customer c_j gets the requirement r_i and 0 otherwise.

Based on above, the overall *score*, or importance of a given requirement $r_i (1 \leq i \leq n)$, can be calculated as $score_i = \sum_{j=1}^{m} w_j \cdot value(r_i, c_j)$. The score of a given requirement is represented as its overall *value* to the organisation.

The aim of the Multi-Objective NRP (MONRP) is to investigate the trade-off between the score and cost of requirements. Let $score = [score_1, \ldots, score_n]$ be the score vector calculated as above. Let $x = [x_1, \ldots, x_n] \in \{0, 1\}^n$ a solution vector, i.e. a binary string identifying a subset of R. Then MONRP is defined as follows:

Definition 3. ***Given:*** *The cost vector,* $cost = [cost_1, \ldots, cost_n]$ *and the score vector (calculated from the customer weights and customer-assigned value of requirements)* $score = [score_1, \ldots, score_n]$.

Problem: *Maximise* $\sum_{i=1}^{n} score_i \cdot x_i$ *while minimising* $\sum_{i=1}^{n} cost_i \cdot x_i$.

Representation. Similar to the test suite minimisation problem in Section 8.1, the candidate solution for NRP should denote whether each requirement will be *selected*, i.e. implemented in the next release. For a set of n requirements, $\{r_1, \ldots, r_n\}$, a candidate solution can be represented with a binary string of length n: the i-th digit is 1 if r_i is to be included in the subset and 0 otherwise.

[2] Bagnall et al. [13] describe a method to remove dependences in this context by computing the transitive closure of the dependency graph and regarding each requirement and all its prerequisites as a new single requirement.

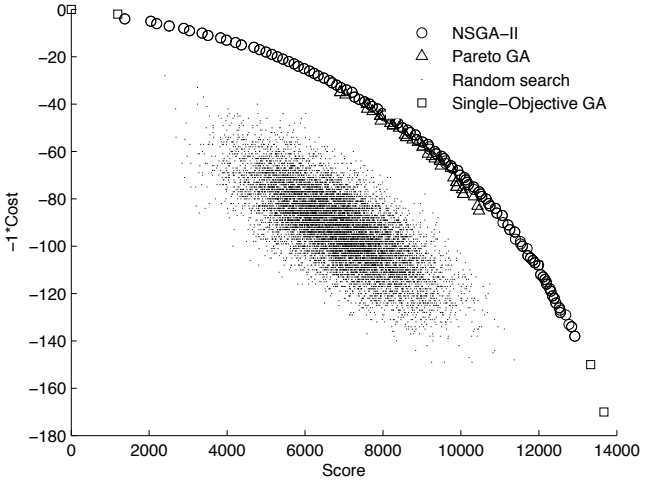
Fitness Function. The *cost* and *profit* function can be directly used as fitness functions for each objectives for MOEAs: *cost* should be minimised while *profit* should be maximised.

Algorithm. The case study compares three different evolutionary algorithms to random search: NSGA-II, Pareto-GA and a single-objective GA. Pareto-GA is a variation of a generic single-objective GA that uses Pareto-optimality only for the selection process. The single-objective GA is used to deal with the multi-objective formulation of NRP by adopting different sets of weights with the weighted-sum approach. When using weighted-sum approach for two objective functions, f_1 and f_2, the overall fitness F of a candidate solution x is calculated as follows:

$$F(x) = w \cdot f_1(x) + (1 - w) \cdot f_2(x)$$

Depending on the value of the weight, w, the optimisation will target different regions on the Pareto front. The case study considered 9 different weight values ranging from 0.1 to 0.9 with step size of 0.1 to achieve wider Pareto fronts.

Results. Figure 14 shows results for an artificial instance of NRP with 40 requirements and 15 customers. Random search produces normally distributed solutions, whereas the weighted-sum, single-objective GA tends to produce solutions at the extremes of the Pareto front. Pareto-GA does produce some solutions that dominate most of the randomly generated solutions, but it is clear



(**a**) 15 customers; 40 requirements

Fig. 14. Plot of results for NRP from different algorithms taken from Zhang and Harman [113]

that the Pareto front is dominantly produced by NSGA-II. Solutions generated by NSGA-II form the widest Pareto front that represents the trade-off between the cost and the expected profit (score).

8.3 Case Study: Structural Test Data Generation

Since the costs of manual software testing are extremely high, the software engineering community has devoted a lot of attention to researching methods of automating the process. The problem of generating structural test data, i.e. test data that attempts to execute all of a program's paths, program statements or true and false decisions, is one area that has attracted a lot of interest, particularly with respect to branch coverage; motivated by the prevalence of its variants in software testing standards.

To motivate the use of Search Based Software Engineering in this context, the program of Figure 15 will be studied, with the aim of generating a test suite that covers each of its individual branches. It is a program for evaluating a Chemical Abstracts Service (CAS) registry number assigned to chemicals. Each number is a string of digits separated by hyphens, with the final digit serving as a check digit. The routine takes a pointer to the first character of the string, processes it, and returns zero if the number is valid. An error code is returned in the case the number is not valid.

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

Given a target structure t in p, the problem is to find an input vector $\boldsymbol{I} \in D$ such that t is executed.

Representation. Defining a representation for structural test data generation simply involves a method of encoding the input vector to a program. This is straightforward for program units such as functions involving primitive types such as integers, reals or characters, as the input vector can be manipulated directly by the search algorithm or trivially encoded into a binary format. However, programs involving arrays or dynamic data structures require more careful handling. In order to avoid a multi-length encoding, the size and shape of the data structure may need to be fixed. However research has been undertaken to remove this restriction [60]. For the CAS check routine, the representation is a sequence of integer values in the range 0-255, fixed to a length of 15. In this case, the whole range of the `char` type is used. For primitive types with large domains, however, the tester may wish to restrict the domains to sensible limits or a legal range of values.

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

Fig. 15. C routine for validating CAS registry numbers of chemical substances (e.g. '7732-18-5', the CAS number of water), taken from McMinn [66]

Fitness Function. In this case study, each branch is taken as the focus of a separate test data search, using the fitness function defined by Wegener et al. [97]. Fitness is computed according to the function $fit(t, \boldsymbol{i}) \rightarrow \mathbb{R}$, that takes a structural target t and individual input $\boldsymbol{i}$, and returns a real number that scores how 'close' the input was to executing the required branch. This assessment is based on a) the path taken by the input, and b) the values of variables in predicates at critical points along the path.

The path taken by the input is assessed and used to derive the value of a metric known as the 'approach level'. The approach level is essentially a count of the target's control dependencies that were not executed by the path. For structured programs, the approach level reflects the number of unpenetrated levels of nesting levels surrounding the target. Suppose, for example, a string is required for the execution of the true branch from line 19, i.e. where the string corresponds to a valid registry number. A diagram charting the computation of fitness can be seen in Figure 16. The approach level will be 2 if no invalid characters are found in the string, but there are too few digits in the string to form a valid CAS number, and the false branch is taken at line 17. If instead the string has too many digits, the true branch is taken at node 17, but the target is then missed because the false branch was taken at node 18, and the approach level is 1. When the checksum calculation is reached at line 19, the approach level is zero.

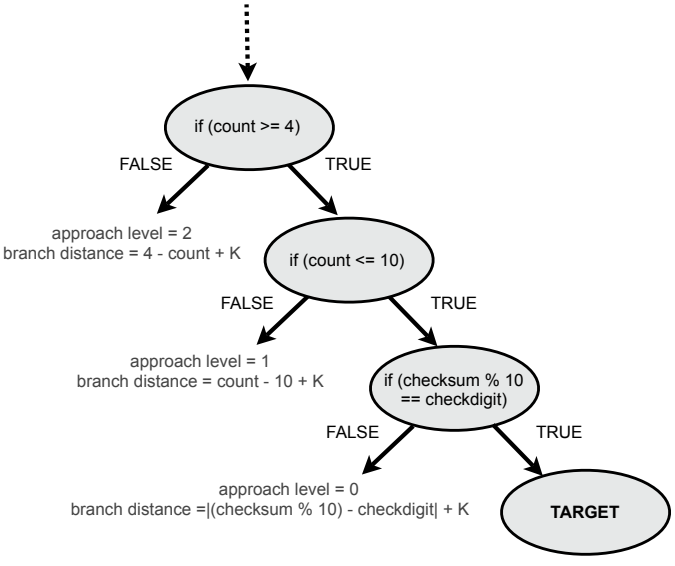


Fig. 16. Fitness function computation for execution of the true branch from line 19 of the CAS registry number check program of Figure 15, taken from McMinn [66]

When execution of a test case diverges from the target branch, the second component, the *branch distance*, expresses how close an input came to satisfying the condition of the predicate at which control flow for the test case went 'wrong'; that is, how close the input was to descending to the next approach level. For example, suppose execution takes the false branch at node 17 in Figure 15, but the true branch needs to be executed. Here, the branch distance is computed using the formula $4-count+K$, where K is a constant added when the undesired, alternate branch is taken. The closer `count` is being greater than 4, the 'closer' the desired true branch is to being taken. A different branch distance formula is applied depending on the type of relational predicate. In the case of `y >= x`, and the `>=` relational operator, the formula is $x - y + K$. For a full list of branch distance formulae for different relational predicate types, see Tracey et al. [93].

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

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

Since the maximum branch distance is generally not known, the standard approach to normalisation cannot be applied [7]; instead the following formula is used:

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

Algorithm. A popular Genetic Algorithm for Search Based Structural Test Data Generation is that of Wegener et al. [97], which we will refer to as the 'Wegener GA' hereinafter. The GA uses a population size of 300 individuals, divided across 6 subpopulations, initially made up of 50 individuals each. It uses a linear ranking method [100] as part of the selection phase. Linear ranking sorts the population into fitness order as assigns new ranked fitness values such that the best individual is awarded a ranked fitness of Z, the median individual a value of 1 and the worst individual a value of $Z - 2$. The Wegener GA uses a value of $Z = 1.7$. Stochastic universal sampling [14] is then used as a selection method, whereby individuals are selected with a probability proportionate to its ranked fitness value. The selection method therefore favours fitter individuals, but the use of ranked fitness values rather than direct values helps prevent super-fit individuals from being selected as many times as they would have been normally, which may go on to dominate the next generation and cause the search to converge prematurely.

The Wegener GA uses a special type of mutation that is well-suited for test data generation problems involving real values. The mutation operator is derived from the Breeder GA [71]. Mutation is applied with a probability p_m of $1/len$, where len is the length of the input vector. The mutation operator applies a different mutation step size, 10^{-pop}, depending on the subpopulation $pop, 1 \leq pop \leq 6$. A mutation range r is defined for each input parameter by the product of pop and the domain size of the parameter. The 'mutated' value of an input parameter can thus be computed as $v' = v \pm r \cdot \delta$. Addition or subtraction is chosen with an equal probability. The value of δ is defined to be $\sum_{y=0}^{15} \alpha_y \cdot 2^{-y}$, where each α_y is 1 with a probability of 1/16 else 0. If a mutated value is outside the allowed bounds of a variable, its value is set to either the minimum or maximum value for that variable. Discrete recombination [71] is used as a crossover operator. Discrete recombination is similar to uniform crossover. However with uniform crossover, genes (input values) are guaranteed to appear in one of the offspring. With discrete recombination offspring individuals receive 'genes' (*i.e.* input variable values) from either parent with an equal probability. Thus a particular gene may be copied into both children, one of the children or neither child.

The Wegener GA, uses an elitist reinsertion strategy, with the top 10% of a current generation retained and used in the next, with the remaining 90% discarded and replaced by the best offspring.

Finally the Wegener GA incorporates competition and migration between each of its subpopulations. A progress value, $prog$, is computed for each population at the end of a generation. This value is obtained using the formula $0.9 \cdot prog + 0.1 \cdot rank$. The average fitness $rank$ for a population is obtained by linearly ranking its individuals as well as the populations amongst themselves (again with $Z = 1.7$). After every 4 generations, the populations are ranked according to their progress value and a new slice of the overall population is computed for each, with weaker subpopulations transferring individuals to stronger ones. However, no subpopulation can lose its last 5 individuals, preventing it from

dying out. Finally, a general migration of individuals takes place after every 20^{th} generation, where subpopulations randomly exchange 10% of their individuals with one another.

Results. Results with the CAS check example can be found in Table 1. The search for test data was repeated 50 times with both the GA outlined in the last section and random search. Random search simply constructs an input by constructing a string where each of the 15 characters is selected at random. Both algorithms were given a budget of 100,000 inputs to generate and execute the program with, in order to find test data for a particular branch. Each branch is denoted as LT|F, where L is the line number of the branch, while T and F denote which of the true or false branches is being referred to.

The performance of the GA is compared with random search. In the table, the average number of test data evaluations (fitness function evaluations for the GA) is reported, unless the branch was not covered over the 50 runs with a 100% success rate, in which case the success rate is reported instead. As can be seen from the table, random search is very effective, covering all the branches of the example except 1 (branch 18F). The GA, on the other hand, achieves 100% coverage. Statistical tests were also performed, as outlined in Section 6.2. For the majority of branches, there was no statistical difference in performance between the two searches. However, for 18F and 19T, the GA was significantly

Table 1. Results with the CAS registry number checking program of Figure 15. The average number of test data evaluations over 50 runs is reported for the branch and algorithm if the success rate of finding test data to execute the branch was 100%, else the success rate is reported instead. A figure appears in bold for the GA if its performance was significantly better than random search.

Branch	**Search**	
	Random	Genetic Algorithm
4T	1	1
4F	1	1
7T	2	2
7F	1	1
8T	2	2
8F	8	8
10T	8	8
10F	2	2
17T	465	329
17F	1	1
18T	465	329
18F	0%	**3,230**
19T	4,270	**802**
19F	519	363

better. Random search never covered 18F, and requires 5 times as much effort (test data evaluations) in order to cover 19T.

8.4 Case Study: Cost Estimation for Project Planning

Software effort estimation is an important activity performed in the planning phase of a software project. Its importance can be easily realised by the fact that the effort estimation will drive the planning of basically all remaining activities in the software development. Given such significance, many approaches have been proposed in the attempt to find effective techniques for software estimation. Nevertheless, as a result of the high complexity of this activity, the search for efficient and effective estimation models is still underway. An interesting example on the application of a search based approach - genetic programming, in this case - to tackle the software estimation problem can be found in [25].

In this application, the software estimation problem is modeled as a search problem, considering as search space the set of cost predictive functions which will have their predictive capability evaluated based on some particular measure. A search algorithm would then seek functions which maximise this evaluation measure.

Therefore, the Software Cost Estimation problem can be defined, as in [25], as follows:

Definition 4. ***Software Cost Estimation Problem***

Given: *Well-formed equations, which can be used to produce cost predictions.*

Problem: *Find the equation with best predictive capability, calculated by measures such as mean squared error or correlation coefficient.*

Representation. For this problem, solutions are represented as trees, expressing well-formed equations. In each tree, terminal nodes represent constants or variables, and each non-terminal node stores a simple function, from a predetermined set of available functions that can be used in the composition of an equation. The available functions proposed in the original paper were: plus, minus, multiplication, division, square, square root, natural logarithm and exponential.

Fitness Function. As pointed out by Dolado [25], classical measures used to evaluate the fitting of equations to some data can be use as fitness functions for this software estimation problem. In the paper, the following measures were considered: *mean squared error*, which quantifies the error of the considered function being used as estimator, and the *correlation coefficient*, which measures the variation between the predicted and the actual values.

Algorithm. Genetic programming (GP), as a variation of the well-known genetic algorithm (GA), can be used to manipulate complex individuals, expressed by data structures representing trees, source codes, design projects, or any other

structure. Similarly to GA, GP performs genetic operations such as selection, crossover and mutation to evolve its populations to seek for more adapted solutions. Dolado [25] employs genetic programming to find a function for software cost estimation.

In that application, as previously described, the candidate functions are described by trees, representing well-formed equations. Given that representation, the usual evolutionary process is performed. Initially, an initial population P, with N individuals (equations) is generated. While a terminate condition in not met, new populations are produced iteratively. First, the members of the current population are evaluated using the fitness function. Next, individuals are selected as input to the genetic operators, including crossover and mutation, which create new individuals that will form the new population.

Results. The proposed search based approach was evaluated over twelve datasets and compared to standard regression analysis. To evaluate the candidate functions, the *mean magnitude of relative error* (MMRE) and the *prediction at level l* (PRED(l)) were used. As reported, the proposed Genetic Programming strategy performed better, considering the PRED(0.25) measure, in eleven out of the twelve cases, but with a slight worse value of the MMRE in some cases.

Even though, from the predictive point of view, both methods did not show considerably satisfactory results, authors pointed out that since GP allows the exploration of a large space of candidate cost functions, this method can provide confidence in the limits of prediction. Additionally, results showed that linear models, regardless of the approach employed, obtained the best predictions in general.

Other authors have also reported interesting applications of search based approaches in software estimation [18, 26, 58].

9 A Taxonomy of Optimisation Problems and Their Mapping to Software Engineering

The Search Based Software Engineering (SBSE) research field has grown rapidly. From its formal definition to this date, a huge number of software engineering problems have been mathematically formulated as optimisation problems and tackled with a considerable variety of search techniques. This growth has taken place in several directions, but more concentrated in a few particular areas. As this natural development occurs, and the number of SBSE problems increases, also grows the necessity of strategies that would help structuring the field. This section introduces a taxonomy, of Search Based Software Engineering problems and instantiates it with the four examples described in the case studies of this the paper.

Our goal in introducing the taxonomy is four fold:

i. to allow researchers to understand the relationship among problems and hypothesise about these relationships.

ii. to highlight to future research directions by identifying unexplored opportunities.
iii. to allow the development of automated search tools to enable effective and efficient search of SBSE problems in any particular repository.
iv. to facilitate re-use, especially regarding approximation algorithms and theoretical bounds.

The proposed taxonomy of Search Based Software Engineering problems will involve Perspectives, Dimensions and Characteristics.

The two Perspectives, SOFTWARE ENGINEERING and OPTIMISATION, will reflect the different points of view under which a particular SBSE problem can be analysed. The Dimensions, as in other taxonomies, will represent, for each Perspective, the SBSE problem features. Finally, for each Dimension, the Characteristics will correspond to the possible feature values under which a particular SBSE problem can be identified. For all Dimensions, the Characteristics are collectively exhaustive. However, only the Dimensions "Objective Space Dimensionality", "Instance Space Characterisation", "Constrained" and "Problem Linearity" are mutually exclusive, for all others, more than one Characteristic may be selected for a particular SBSE problem.

In Tables 2 and 3, the proposed taxonomy of Search Based Software Engineering problems is presented.

For the SOFTWARE ENGINEERING Perspective (Table 2), four Dimensions are identified: "Software Development Stage(s)", "Software Development Model(s)", "Main Subject Descriptor(s)" and "Main Implicit Subject Descriptor(s)".

1. The **"Software Development Stage(s)"** positions the problem under one, or more, stages in the software engineering process. The Characteristics available for this Dimension are representative of the standard software development process.
2. Next, the **"Software Development Model(s)"** identifies a particular set of development models in which the problem occurs.
3. The Dimension named **"Main Subject Descriptor(s)"** describes the software engineering subject addressed by the problem. The Characteristics present in this Dimension were obtained from the 1998 version of the ACM Computing Classification System [1]. More specifically, the possible values for this feature are those defined as a "Subject Descriptor" under the level D.2 (SOFTWARE ENGINEERING), in the third level of the classification structure, with values, and corresponding subjects, ranging from D.2.0 (General) to D.2.13 (Reusable Software) and D.2.m (Miscellaneous).
4. Finally, the **"Main Implicit Subject Descriptor(s)"** Dimension details the previous subject descriptor(s), by allowing the selection of the more specific subject descriptors present in the fourth level of the ACM Computing Classification System [1], once again under the level D.2 (SOFTWARE ENGINEERING).

Table 2. Taxonomy of Search Based Software Engineering Problems - SOFTWARE ENGINEERING Perspective

A. SOFTWARE ENGINEERING Perspective

1. Software Development Stage(s)
   (a) Software Planning
   (b) Requirement Engineering
   (c) Software Design
   (d) Implementation/Coding
   (e) Integration
   (f) Testing/Validation
   (g) Deployment
   (h) Maintenance
2. Software Development Model(s)
   (a) Waterfall Model
   (b) Spiral Model
   (c) Iterative and Incremental Development
   (d) Agile Development
3. Main Subject Descriptor(s)
   (a) Subject Descriptors under SOFTWARE ENGINEERING (D.2), in the 1998 ACM Computing Classification System.
4. Main Implicit Subject Descriptor(s)
   (a) Implicit Subject Descriptors under SOFTWARE ENGINEERING (D.2), in the 1998 ACM Computing Classification System.

For the OPTIMISATION Perspective, other six Dimensions are defined.

1. The **"Objective Space Dimensionality"** is descriptive of the number of objective functions present in the formulation.
2. **"The Instance Space Characterisation"** Dimension evaluates the problem variables as continuous or discrete.
3. Next, **"Constrained"** accounts for the presence of restrictions.
4. **"Problem Linearity"** indicates, for both objective and restriction functions, their linearity.
5. The following Dimension, **"Base NPO Problem Type(s)"**, attempts to extract the problem category, using the classification proposed by Garey and Johnson [34] and employed in the Compendium of NP Optimisation Problems [22]. The general types present in the Compendium are: Graph Theory, Network Design, Sets and Partitions, Storage and Retrieval, Sequencing and Scheduling, Mathematical Programming, Algebra and Number Theory,

Games and Puzzles, Logic, Automata and Language Theory, Program Optimisation and Miscellaneous.

6. Finally, **"Base NPO Problem"** tries to relate the considered SBSE problem with a generally defined NP optimisation problem, in a way one could employ the known results, including approximation algorithms and theoretical bounds, previously available in the literature regarding that general problem. For that purpose, once again, the "Compendium of NP Optimisation Problems" will be used.

At this point, it is worth mentioning that the "Compendium of NP Optimisation Problems" presents a considerable variety of optimisation problems in the most different categories, however, it lacks a formal definition of a basic optimisation problem, under which several known problems could be classified. To tackle this absence, the definition of a **BASIC OPTIMISATION PROBLEM**, which would fall under the **MISCELLANEOUS** type, defined with the same basic ingredients employed in the Compendium, is presented below.

BASIC OPTIMISATION PROBLEM

Instance: Finite or infinite set U, for each $u \in U$ a fitness value $f(u) \in Z^+$.
Solution: An element, $u' \in U$.
Measure: Fitness value of u, i.e., $f(u)$.

9.1 Classification Examples

In order to illustrate the representation of Search Based Software Engineering problems under the proposed taxonomy, it is presented, next, the classification of SBSE problems discussed in section 8: The Regression Test Case Selection problem (Table 4), the Next Release problem (Table 5), the Structural Test Case Generation problem (Table 6) and the Software Cost Estimation problem (Table 7).

The Multi-Objective Regression Test Case Selection problem [106] extends previously published mono-objective formulations. The paper discusses two variations, one which considers two objectives (code coverage and execution time), used here, and the other covering three objectives (code coverage, execution time and fault detection). Consider, for this problem, the special case where a set of test cases which covers 100% of the code is sought. In addition, consider that all test cases have the same execution time. In that case, the Test Case Selection Problem problem can be seen as a application of the **MINIMUM SET COVER** problem.

MINIMUM SET COVER

Instance: Collection C of subsets of a finite set S.
Solution: A set cover for S, i.e., a subset $C' \subseteq C$ such that every element in S belongs to at least one member of C'.
Measure: Cardinality of the set cover, i.e., $|C'|$.

S will represent the set of all statements in the source code. Collection C will contain the test cases that can be selected. Each test case covers a number of

Table 3. Taxonomy of Search Based Software Engineering Problems - OPTIMISATION Perspective

A. OPTIMISATION Perspective
1. Objective Space Dimensionality
(a) Mono-objective
(b) Multi-objective
2. Instance Space Characterisation
(a) Discrete
(b) Continuous
3. Constrained
(a) Yes
(b) No
4. Problem Linearity
(a) Linear
(b) Nonlinear
5. Base NPO Problem Type(s)
(a) Problem Categories as defined in the Compendium of NP Optimisation Problems
6. Base NPO Problem(s)
(a) Problems as defined in the Compendium of NP Optimisation Problems

statements in the source code, which means that each test case can be representative of a subset of S. Thus, the solutions are set covers for S, that is, a subset of test cases, $C' \subseteq C$, such that all statements in S are covered, meaning that each statement is covered by, at least, one of the members of C'. The solution sought is the one with the lowest cardinality, which will have lowest execution time, since all test cases have the same execution time.

For the other dimensions in the OPTIMISATION perspective (Table 4), the Multi-Objective Test Case Selection Problem can be classified as Multi-objective, having a Discrete instance space, Unconstrained and Linear. Over the SOFTWARE ENGINEERING perspective, the problem falls under the Testing/ Validation development stage and is not particular to any specific development model. Furthermore, it has as main subject descriptor the choice "D.2.5 Testing and Debugging", and "Testing Tools" as implicit subject descriptor.

The Next Release Problem (NRP), in its original formulation as a constrained mono-objective optimisation problem [13], involves determining a set of customers which will have their selected requirements delivered in the next software release. This selection prioritises customers with higher importance to the company (maximise $\sum_{i=1}^{n} w_i$), and must respect a pre-determined budget (subject to $\sum_{i=1}^{n} c_i \leq B$).

Table 4. Typification of the Regression Test Case Selection under the Proposed Taxonomy

A. SOFTWARE ENGINEERING Perspective
1. Software Development Stage(s)
Testing/Validation
2. Software Development Model(s)
Waterfall Model
Spiral Model
Iterative and Incremental Development
Agile Development
3. Main Subject Descriptor(s)
D.2.5 Testing and Debugging
4. Main Implicit Subject Descriptor(s)
Testing Tools
B. OPTIMISATION Perspective
1. Objective Space Dimensionality
Multi-objective
2. Instance Space Characterisation
Discrete
3. Constrained
No
4. Problem Linearity
Linear
5. Base NPO Problem Type(s)
SETS AND PARTITIONS
6. Base NPO Problem(s)
MINIMUM SET COVER

In typifying this problem under the proposed taxonomy, it is easy to see that, regarding the SOFTWARE ENGINEERING perspective (Table 5), the NRP is positioned under the Requirement Engineering software development stage, occurring specially in the Iterative and Incremental and Agile Development Models. In addition, the Next Release Problem addresses the subject "D.2.1 Requirements/Specifications", present in the ACM Computing Classification System. Under the OPTIMISATION perspective, as stated above, it is a mono-objective problem, since it aims to solely maximise the importance of the customers which will have their requirements delivered. It has a discrete instance space and should be classified as a constrained problem, since considers a pre-defined budged as restriction. Since both objective and restriction functions are linear, the overall

Table 5. Typification of the Next Release Problem under the Proposed Taxonomy

A. SOFTWARE ENGINEERING Perspective
1. Software Development Stage(s)
Requirement Engineering
2. Software Development Model(s)
Iterative and Incremental Development
Agile Development
3. Main Subject Descriptor(s)
D.2.1 Requirements/Specification
4. Main Implicit Subject Descriptor(s)
B. OPTIMISATION Perspective
1. Objective Space Dimensionality
Mono-objective
2. Instance Space Characterisation
Discrete
3. Constrained
Yes
4. Problem Linearity
Linear
5. Base NPO Problem Type(s)
MATHEMATICAL PROGRAMMING
6. Base NPO Problem(s)
MAXIMUM KNAPSACK

problem can be considered linear as well. Finally, it involves the solution of a MATHEMATICAL PROGRAMMING problem, as defined in the "Compendium of NP Optimisation Problems". In fact, the Next Release Problem can be seen as a specialisation of the MAXIMUM KNAPSACK Problem, as discussed next.

Consider the mathematical definition of the **MAXIMUM KNAPSACK** problem, as presented in [22]:

MAXIMUM KNAPSACK

Instance: Finite set U, for each $u \in U$ a size $s(u) \in Z^+$ and a value $v(u) \in Z^+$, a positive integer $B \in Z^+$.
Solution: A subset $U' \subseteq U$ such that $\sum_{u \in U'} s(u) \leq B$.
Measure: Total weight of the chosen elements, i.e., $\sum_{u \in U'} v(u)$.

As defined in **Instance**, the set U can represent the customers which may have their requirements delivered in the next software release. Each customer, $u \in U$,

has an importance, represented here by $v(u)$, which expresses the function w_i in the NRP definition. The goal is to maximise the sum of the importance of the selected customers. However, only solutions respecting the problem restriction can be considered. In the definition of the MAXIMUM KNAPSACK problem, each element $u \in U$ has a size, $s(u) \in Z^+$, which will limit the amount of elements that can be selected. In the Next Release problem, the function $s(u)$ will represent the overall cost of implementing all requirements desired by customer u, that is, $s(u) = c_i$.

For the Structural Test Data Generation problem, the classification under the proposed taxonomy, as shown in Table 6, considering the SOFTWARE ENGINEERING perspective, would place this problem under the Testing/Validation

Table 6. Typification of the Structural Test Data Generation Problem under the Proposed Taxonomy

A. SOFTWARE ENGINEERING Perspective
   1. Software Development Stage(s)
      Testing/Validation
   2. Software Development Model(s)
      Waterfall Model
      Spiral Model
      Iterative and Incremental Development
      Agile Development
   3. Main Subject Descriptor(s)
      D.2.5 Testing and Debugging
   4. Main Implicit Subject Descriptor(s)
      Testing Tools

B. OPTIMISATION Perspective
   1. Objective Space Dimensionality
      Mono-objective
   2. Instance Space Characterisation
      Continuous
   3. Constrained
      No
   4. Problem Linearity
      Linear
   5. Base NPO Problem Type(s)
      MISCELLANEOUS
   6. Base NPO Problem(s)
      BASIC OPTIMISATION PROBLEM

development stage. Regarding development models, this test data generation issue arises in all models covered by the taxonomy. "D.2.5 Testing and Debugging" would be the main subject descriptor of such problem and "Testing Tools" would work as implicit descriptor.

Considering now the OPTIMISATION perspective, this problem could be characterised as mono-objective, since it composes two measures (approach level and branch distance) in a single evaluation function. Additionally, it works with a usually continuous instance space formed of input vectors. It is also an unconstrained and linear problem. Finally, the Structural Test Data Generation problem can be seen as a simple instantiation of the BASIC OPTIMISATION PROBLEM described earlier, where, given a target structure t in p, the problem involves simply searching for an input vector $\boldsymbol{i} \in D$, representing elements in the instance space U, with minimum value given by evaluation function $fit(t, \boldsymbol{i})$, representing $f(u)$ in the description of the BASIC OPTIMISATION PROBLEM.

Finally, the Software Cost Estimation problem is associated with the Software Planning development phase and all development models (Table 7). The most adequate mains and implicit subject descriptors would be "D.2.9 Management" and "Cost Estimation", respectively.

Similarly to the Test Data Generation problem, this problem is an instantiation of the BASIC OPTIMISATION PROBLEM. In this case, the problem seeks solutions represented by well-formed functions, forming the instance set U, looking for a solution with minimum value given by function $f(u)$, associated with measures such as *minimum squared error* or *correlation coefficient*. Additionally, the problem should be classified as mono-objective, continuous, unconstrained and nonlinear.

10 Next Steps: Getting Started

This section is primarily aimed at those who have not used SBSE before, but who have a software engineering application in mind for which they wish to apply SBSE. Throughout this section the emphasis is unashamedly on obtaining the first set of results as quickly as possible; SBSE is attractive partly because it has a shallow learning curve that enables beginner to quickly become productive. There is an excitement that comes with the way in which one can quickly assemble a system that suggests potentially well optimised solutions to a problem that the experimenter had not previously considered.

By minimising the time from initial conception to first results, we seek to maximise this excitement. Of course subsequent additional work and analysis will be required to convert these initial findings into a sufficiently thorough empirical study for publication. The goal of the section is to take the reader from having no previous work on SBSE to the point of being ready to submit their first paper on SBSE in seven simple steps. The first four of these steps are sufficient to gain the first results (and hopefully also the excitement that comes with the surprises and insights that many authors have experienced through using SBSE for the first time).

Table 7. Typification of the Software Cost Estimation Problem under the Proposed Taxonomy

A. SOFTWARE ENGINEERING Perspective
1. Software Development Stage(s)
   Software Planning
2. Software Development Model(s)
   Waterfall Model
   Spiral Model
   Iterative and Incremental Development
   Agile Development
3. Main Subject Descriptor(s)
   D.2.9 Management
4. Main Implicit Subject Descriptor(s)
   Cost Estimation

B. OPTIMISATION Perspective
1. Objective Space Dimensionality
   Mono-objective
2. Instance Space Characterisation
   Continuous
3. Constrained
   No
4. Problem Linearity
   Nonlinear
5. Base NPO Problem Type(s)
   MISCELLANEOUS
6. Base NPO Problem(s)
   BASIC OPTIMISATION PROBLEM

Step 1: The first step is to choose a representation of the problem and a fitness function (see Section 3). The representation is important because it must be one that can be readily understood; after all, you may find that you are examining a great many candidate solutions that your algorithms will produce.

One should, of course, seek a representation that is suitable for optimisation. A great deal has been written on this topic. For the first exploration of a new SBSE application the primary concern is to ensure that a small change to your representation represents a small change to the real world problem that your representation denotes. This means that a small change in the representation (which will be reflected by a move in a hill climb or a mutation in a genetic algorithm) will cause the search to move from one solution to a 'near neighbour'

solution in the solution space. There are many other considerations when it comes to representation, but for a first approach, this should be sufficient to contend with.

Perhaps for the first experiments, the most important thing is to get some results. These can always be treated as a baseline, against which you measure the progress of subsequent improvements, so 'getting it a bit wrong' with the initial choices need not be a wasted effort; it may provide a way to assess the improvements brought to the problem by subsequent development.

Step 2: Once there is a way to represent candidate solutions, the next step is to chose a fitness function. There may be many candidate fitness functions. Choose the simplest to understand and the easiest to implement to start off with. Once again, this may provide a baseline against which to judge subsequent choices. Ideally, the fitness function should also be one that your implementation will be able to compute inexpensively, since there will be a need for many fitness evaluations, regardless of the search technique adopted.

Step 3: In order to ensure that all is working, implement a simple random search. That is, use a random number generator to construct an entirely arbitrary set of candidate solutions by sampling over the representation at random. This allows you to check that the fitness function is working correctly and that it can be computed efficiently. One can also examine the spread of results and see whether any of the randomly found solutions is any good at solving the problem. Despite its simplicity, random search is simple and fast and many researchers have found that it is capable of finding useful solutions for some Software Engineering applications (for instance, in testing, where it has been very widely studied [55, 79]).

Step 4: Having conducted steps 1-3 we are in a position to conduct the first application of a search based technique. It is best to start with hill climbing. This algorithm is exceptionally easy to implement (as can be seen from Section 4). It has the advantage that is is fast and conceptually easy to understand. It can also be used to understand the structure of the search space. For instance, one can collect a set of results from multiple restart hill climbing and examine the peaks reached. If all peaks found are identical then hill climbing may have located a large 'basin of attraction' in the search space. If all peaks are of very different heights (that is, they have different fitness values) then the search space is likely to be multimodal. If many hill climbs make no progress then the search space contains many plateaux (which will be a problem for almost any search based optimisation technique).

It is possible that simply using hill climbing, the results will be acceptable. If this is the first time that SBSE has been applied to the Software Engineering problem in hand, then the results may already be at the level at which publication would be possible. If the peaks are all acceptable but different then the approach already solves the problem well and can give many diverse solutions.

This was found to be the case in several software engineering applications. For instance, in software modularisation [68], the hill climbing approach produced

exceptionally good results for determining the module boundaries of source code that remained unsurpassed by more sophisticated algorithms from 1998 [64] until 2011 [77]. Furthermore, the multi objective genetic algorithm that found better solutions only managed to do so at a cost of two orders of magnitude more fitness evaluations [77].

Korel [59] was one of the first authors to apply a search based optimisation approach to a software engineering problem. He used a hill climbing approach for software test input generation. Though this problem has been very widely studied, this hill climbing approach still proved to be an attractive approach 17 years later, when it was studied and (favourably) compared empirically and theoretically to a more sophisticated genetic algorithm [49], suggesting that a hybrid that combined both hill climbing (local search) and genetic algorithms (global search) was the best approach for test data generation [50].

Hill climbing can also be used to help to understand the nature of the solutions found. For example, through multiple hill climbs, we can find the set of common building blocks that are found in all good solutions. This can help to understand the problem and also may be a way to make subsequent techniques more effective. This approach to locating building blocks using multiple runs of a hill climber was applied by Mahdavi et al. [62], who used a distributed cluster of machines to perform multiple hill climbs in parallel for the software modularisation problem. The results were the building blocks of 'good modules' (those which were cohesive and exhibited low coupling) for a particular architecture of dependencies. Mahdavi et al. also showed that the initial identification of building blocks could improve the performance of the subsequent search.

For all these reasons, faced with a large number of possible algorithms with which to start, it seems sensible to adopt hill climbing as the first optimisation algorithm. If the results are promising then within a very short space of time, the novice SBSE researcher will have migrated from finishing reading this tutorial to starting to write their own first SBSE paper; perhaps in as little as a matter of days.

Step 5: The natural next step is to try some different search based algorithms. A good place to start is those described in Section 4 since these have been commonly applied and so there will be a wealth of previous results with which to compare. Following this, the SBSE researcher is already going beyond what can be covered in a single tutorial such as this and is thus referred to the literature on search based optimisation. A good overview of search techniques can be found in the text book by Burke and Kendall [19].

Step 6: Having implemented several SBSE algorithms and obtained results, the next step is to analyse these results. In this paper we have set out some of the common statistical techniques used to analyse SBSE work in Section 6. Naturally, this can only provide an overview of some commonly used approaches, but it should be sufficient to address some of the most obvious initial questions; those to which a referee would naturally expect answers. Using these techniques the results can be developed to become the basis for an experimental or

empirical study, by selecting some non trivial real world examples to which the new SBSE approach can be applied. The most natural questions to address are those of efficiency and effectiveness. That is, the study should, at least, answer the questions: how good are the solutions you find and how much computational (and/or human) effort is required to obtain them?

Step 7: One of the attractive aspects of SBSE is the way it re-unites different areas of software engineering, as explained in Section 2. A good paper on SBSE will contain a thorough account of the related work and this may help to achieve this re-unification goal, drawing together apparently unrelated areas of software engineering. In so doing, SBSE work can play a vital role in the development of the wider field of Software Engineering itself, providing a more solid search-based understanding of the underlying optimisation problems that are found in each application area.

Using SBSE, we seek to apply search algorithms to software engineering problems so there are two natural sources of related work for any paper on SBSE; the previous work that tries to solve the same (or similar) Software Engineering problem(s) and the previous work that uses a similar search based approach. You may find that two apparently quite different software engineering problems have been attacked using the same (or similar) search based formulation (perhaps representation is shared or a similar fitness can be used).

An SBSE paper can be considerably enhanced by exploring these links, since such connections mean that the paper can have an impact on at least two software engineering domains, rather than merely the one for which the results are presented. Hopefully, in developing a related work section, the taxonomy in Section 9 will be helpful. The many surveys on SBSE are also a source of valuable summary information concerning potentially related techniques and application areas.

Step 8: At this point, the researcher has sufficient information and results to consider writing a paper. Naturally, there will be a choice about where to send the paper that can only be made by the author(s). There is also the question of how to set the problem formulation, research questions and results into as format that will appeal to (firstly) referees and (ultimately) readers.

There are many excellent papers that give guidance on how to write good software engineering papers, such as that by Mary Shaw [86]. Those papers that present results on SBSE generally (though not exclusively) fall in the category of empirical software engineering papers, for which the systematic review of Ali et al. [4] sets out useful guidelines relevant to SBSE.

11 SBSE Limitations and Techniques for Overcoming Them

In this final section we review some of the problems and issues that can arise when using SBSE and some simple techniques for overcoming them. Many of these issues are common to all approaches to optimisation based on algorithms

guided by fitness and many of the solution approaches could be described as the application of 'common sense' (though some may be slightly surprising or counter-intuitive). Where possible, we provide pointers to the SBSE literature, indicating where the proposed solution approaches have already been explored in the specific context of Software Engineering problem domains.

11.1 My Ideal Fitness Function Is Too Computationally Expensive

The most natural fitness function for a problem may turn out to be computationally expensive and this may mean that the whole search process takes a long time. In most applications of SBSE, it is the computation of fitness that occupies the largest part of the overall computational cost of the SBSE implementation. As such, it makes sense to consider techniques for controlling and reducing this cost, where it is manageable. This issue, therefore, can be considered to be the problem of 'how can we compute fitness faster?'. We consider three approaches: use a cheaper surrogate, parallelise and imbue the search with domain knowledge.

Find a cheaper surrogate: Often, a computationally demanding fitness function can be reserved for evaluating the final result or for occasional fitness computation, while a surrogate (or surrogates) is/are used for most of the fitness evaluations used to guide the search. Even if the surrogate fitness function is not a perfect guide, it can be computationally cheaper overall, to use a less accurate fitness function (that still provides *some* guidance for the majority of fitness computations). This approach has been used very little in Software Engineering, partly because many of the fitness functions used in SBSE tend to be computationally inexpensive (they often come from works on metrics [42], which are pre-designed to be computationally practical). Even when the metrics used as fitness functions do prove to be computationally expensive, it is typically hard to find surrogates. However, as SBSE increasingly finds applications in new software engineering areas there may also be a wider choice of available metrics and it may turn out that the most suitable metrics are also those that are more computationally expensive. We can therefore expect that there will be a greater reliance on surrogate fitness computations in future work on SBSE. To minimise the negative impact of using a surrogate that only captures part of the true fitness or which includes noise, it may be advantageous to use multiple surrogate fitness computations (as discussed later on in this section).

Parallelise: SBSE algorithms are known as 'embarrassingly parallel' [32] because of their potential for scalability through parallel execution of fitness computations [72]. Recent work has shown how this parallelism can be exploited on General Purpose Graphical Processing devices (GPGPUs) [110] with scale ups in overall computation time up to a factor of 20. Because of the inherent parallelism of SBSE algorithms and the wide availability of cheap multicore devices we can expect a great deal more scalability research in future work on SBSE. In the era of multicore computing, SBSE is well placed to make significant strides forward in simple effective and scalable parallel solutions.

Use domain knowledge: Domain Knowledge can be used to guide the search to fruitful areas of the landscape, without actually determining the precise solution. For example, in selection and prioritisation problems, we may know that the solution we seek must include certain items (for selection these can be hard wired onto the solution) or we may know that a certain relative placement of some individuals in an ordering is highly likely. This can happen often as the result of human considerations concerned with management and business properties. For instance, in selecting and prioritising requirements, it is not always possible to take account of all the socio-political issues that may determine the ideal solution set. The manager may simply say something like 'whatever the solution you adopt, we must include these five requirements, because the CEO deems them essential for our business strategy, going forward'. This can be an advantage for search, because it simultaneously and effortlessly adapts the solution to the business needs while reducing the size of the search space. Wherever possible, domain knowledge should be incorporated into the SBSE approach.

11.2 My Fitness Function Is Too Vague and Poorly Understood to Make It Something I Can *Compute*

It is a popular misconception that SBSE must use a fitness function that is both precise and accurate. It is true that this is advantageous and valuable (if possible), but neither is essential. In software measurement, we seek metrics that meet the 'representation condition' [87], which states that the ordering imposed by the metric on the individuals it measures should respect the 'true ordering' of these individuals in the real world.

It is a natural condition to require of a metric, M; if $M(a) > M(b)$ then we should reasonably expect that the real world entity a is truly 'better' than b in some sense and *vice versa*. However this requirement is not essential for SBSE. If a fitness function merely *tends* to give the right answer, then it may well be a useful, if inefficient, fitness function candidate. That is, if the probability is greater than 0.5 that a pairwise fitness comparison on two individuals a and b with metric M will produce the correct outcome, then may potentially be used in SBSE should it prove to have compensatory properties; our search will be guided to some extent, though it may make many wrong moves.

However, if a metric is simply not defined for some part of the candidate solutions space or can only be defined by subjective assessment for all or part of the solution space, then we need a way to handle this. There are two natural approaches to this problem: use a human or use multiple fitness functions.

Involve Human Judgement: Fitness functions do not need to be calculated entirely automatically. If they can be fully automated, then this is advantageous, because one of the overall advantages of SBSE is the way it provides a generic approach to the problem of Automating Software Engineering [36]. However, it has been argued [37] that some Software Engineering tasks, such as those associated with comprehension activity are inherently subjective and require a human input to the fitness computation. This is easily accommodated in SBSE,

since may algorithms allow for 'human-in-the-loop fitness computation. Such subjective, human-guided, fitness has been used in interactive evolutionary algorithms for SBSE applied to design-oriented problems [89] and Requirements Engineering [92].

Use Multiple Fitness Functions: It is not necessary to use only the 'best' fitness function to guide the search. If the best fitness function is still only the best at capturing *part* of the search space there is nothing to prevent the use of other fitness functions that capture different, perhaps smaller, parts of the solution space. Fitness functions can be combined in a number of ways to provide an 'agglomerated' fitness function. Both weighting and Pareto optimal approaches have been widely studied in the SBSE literature [36]. However, using several fitness functions, each of which applies to different parts of the solutions space has not been explored in the literature. Given the complicated and heterogenous nature of many Software Engineering problems, this approach is under-explored and should receive more attention. In future work on SBSE, we may seek to bundle patchworks of different fitness functions to solve a single problem, deploying different fitness functions at different times, for different stake holders, or for different parts of the solutions space.

11.3 My Search Space Is Too Difficult for Search to Work Well

The performance of a search based optimisation algorithms depends crucially on the search landscape that a fitness function creates when used to guide the search for a specific problem. If a particular search algorithm performs poorly on a search space then there are two obvious solutions that immediately present themselves; do something to modify the way fitness is computed or choose a different algorithm (one that is better suited to the landscape). In order to take either course of action, it is important to undertake research into the properties of the search landscape in order to understand which is the best algorithm to apply. There has been much work on analysis and characterisation of SBSE landscapes and fitness functions, but more is required in order to provide a more complete understanding of the properties to which SBSE is applied.

Analyse Different Fitness Functions: Different characterisations of fitness can achieve very different results. This has been demonstrated empirically, in SBSE problems, where the choice of fitness can have different robustness to noise in the data [51]. The initial choice of fitness function may lead to a search landscape contains too many plateaux, or other features that make search hard (needle in a haystack, multimodal features, deceptive basins of attraction etc.). In these situations, it makes sense to analyse the effect of different fitness functions; each will characterise the problems differently and may have very different behaviours, even if all agree on the local or global optima.

Use Multiple Fitness Functions: Even if your search problem is inherently single objective in nature, it may make sense to consider experimenting with multi objective approaches. It has been shown in previous work on SBSE for

module clustering [77] that better results can be obtained for using a multi objective optimisation on a single objective problem. This happens because the other objectives may help the search to find routes to the global optima in the single objective space. Therefore, searching for solutions that satisfy multiple objectives may, perhaps counter-intuitively, help to solve a single objective problem. If you find that one of your fitness characterisation has an unattractive search landscape, yet it provides useful guidance in some cases, you might consider incorporating additional fitness functions.

Use Secondary Fitness: For problems in which there are too many plateaux, you may consider the use of a secondary fitness function, to be used to distinguish candidate solutions that lie on a plateaux according to the primary fitness function. This has been used in the SBSE problem of search based transformation. In transformation, the goal is to find a new version of the program that is better (according to some metric) by searching the space of transformations of the original program (or the transformations sequences that can be applied to it). This is a very well studied area of SBSE [21, 29, 30, 52, 75, 85], dating back to the work of Ryan and Williams [83, 104] on auto-parallelisation transformations.

One of the problems for this SBSE problem, is that there are many transformations the application of which fails to affect the primary fitness function. For example, suppose we seek to shrink the size of a program by either removing redundant computation or by slicing [44]. In this situation, there are many transformations that will not reduce the size of the program to which they are applied. All such transformations will lie on a plateau of fitness with regard to their effect on code reduction for some specific program. However, we can distinguish among such transformations. Those that are not applicable are worse than those that are applicable. Those that are applicable, but have no effect at all are worse than those that alter the code without reducing its size. In the early stages of the search those transformations that have a larger effect on the syntax may also be preferred. This suggests a secondary fitness that can be used to guide a search to the edges of a plateaux in the search space induced by the primary fitness function. This has been employed to improve the performance of search based slicing [30].

Landscape Analysis: In the more general optimisation literature, the issue of landscape analysis and algorithmic characterisation is well studied [103]. For example, there has been work on the analysis of plateaux in search landscapes [91]. There has also been much work on SBSE landscape analysis and algorithm characterisation. Early work in this area for the project estimate feature selection [57] and modularisation [67] has been championed in the SBSE literature [36] as exemplary of the kinds of analyse that can be achieved, empirically, using a simple (but effective) multiple restart simple hill climbing approach. There has also been recent theoretical analysis of SBSE algorithm performance [61] and theoretical and empirical analyses of search based testing for structural test data generation [7, 49, 50].

Choose the Right Algorithm for the Search Space: Choosing the right algorithm for the problem is as fundamental to search as choosing the right tool for the job is in any engineering endeavour. The field of optimisation is well known for its many versions of the 'no free lunch theorem' [102]. There is plenty of evidence [3, 48, 78] to indicate that SBSE problems are as wide and varied as those found in the general optimisation literature. It is, therefore, foolhardy to believe that one search based optimisation technique will be superior for all possible problems that one might encounter.

The most popular algorithms (by far) that have been used hitherto in SBSE work are variations on the theme of population-based evolutionary algorithm [48]. This is not the result of evidence that evolutionary algorithms are superior to other algorithms. Quite the contrary in fact: There is evidence to suggest that, for some problems, such as structure test data generation (a very widely studied problem), simpler local search algorithms may be better [49], and that some form of hybridisation that seeks to achieve the best of both local and global search [50] may be the best performing approach so far known.

11.4 The Final Fitness I Obtained Is Simply Too Poor: The Solutions Are Just not Good Enough

Usually, even with a relatively 'out of the box' choice of fitness function and search based optimisation technique, the results obtained will be better than those obtained using a purely random search. However, you may still feel that the results are not as good as you would like, or, if you have as specific threshold fitness value in mind, you may find that your algorithm fails to achieve this threshold, even when you allow it considerable computation resources. In this situation, you should not give up and assume that 'SBSE does not work' (it is just possible that it may not, but it is certainly too soon to be sure!). It may be that your algorithm is performing poorly because of some of the parameter choices or because it is the wrong algorithm for this particular problem. Even should all else fail, you may be able to extract useful information from the suboptimal results you have obtained.

Avoid Premature Convergence: Premature convergence on a local optima often turns out to underlie the observation that an SBSE algorithm fails to produce 'good enough' results. This can happen because, for example, too much elitism has been incorporated into an evolutionary approach, or because some domain knowledge has been injected in a way that strongly biases solutions towards only one part of the search space. In general, it may be helpful to think of your search process as a compromise between exploration and exploitation (a common distinction in the more general optimisation literature). If you fail to explore sufficiently, then premature convergence will result. If you fail to exploit sufficiently, then you may have a broad spread of solutions across the search space, none of which is of particularly high quality. It is a good overarching principle to seek to favour exploration in the earlier stages of the search process and exploitation subsequently. This principle is captured, elegantly, by the

cooling parameter of simulated annealing, though there are ways to incorporating similar ideas into almost all search algorithms.

Try Other Search Algorithms: As noted above, the reasons for poor performance could simply be that the wrong search algorithm is used for the search space in hand. If the fitness landscape resembles one enormous hill (or lots of hills of equal fitness) then hill climbing is clearly an attractive candidate. For landscapes with so-called 'royal road' properties [70], an evolutionary algorithm will be favourable. These distinctions are starting to be explored in the SBSE literature [49]. It is always advisable to explore with several search based algorithms in any SBSE work, to include (as a sanity check) random search, together with at least one local and one global search technique, simply to get a sense for the variabilities involved. Of course, comparing these will require some thought and careful planning, as explained in Section 6.

Look for Building Blocks: It is unlikely, but suppose you discover that you cannot get the higher quality results you seek after trying several fitness functions and many different algorithms. What then? Well, in this situation, you will have a large set of results, albeit a set of sub optimal results. There is a great deal of value that can be obtained from such a set of results. You can use them to understand the structure of the search space. This may helpful to explain why your results turn out the way they do. Furthermore, you can search for building blocks in the solutions, that can help you to identify *partial* solutions or fragments of good solutions that can help to identify better solutions. Such building blocks may lead a human to a 'eureka' moment, when they gain insight into the structure of some essential or sufficient ingredient of a good solution. They can also be used to constrain subsequent search based approaches, that may then prove more successful. This two-stage search approach has been shown to be effective in SBSE work; it has been used to identify the building blocks of good software modularisations for a subsequent search over a constrained (and therefore much smaller) landscape in which the building blocks are now fixed [62].

12 Conclusion

We hope that this tutorial paper has been a useful guide to the development of the reader's (perhaps first) SBSE paper. We look forward to reading and learning from your work on SBSE.

References

1. ACM. The 1998 ACM computing classification system (2009), `http://www.acm.org/about/class/1998`
2. Adamopoulos, K., Harman, M., Hierons, R.M.: How to Overcome the Equivalent Mutant Problem and Achieve Tailored Selective Mutation Using Co-evolution. In: Deb, K., et al. (eds.) GECCO 2004. LNCS, vol. 3103, pp. 1338–1349. Springer, Heidelberg (2004)

3. Afzal, W., Torkar, R., Feldt, R.: A systematic review of search-based testing for non-functional system properties. Information and Software Technology 51(6), 957–976 (2009)
4. Ali, S., Briand, L.C., Hemmati, H., Panesar-Walawege, R.K.: A systematic review of the application and empirical investigation of search-based test-case generation. IEEE Transactions on Software Engineering (2010) to appear
5. Antoniol, G., Gueorguiev, S., Harman, M.: Software project planning for robustness and completion time in the presence of uncertainty using multi objective search based software engineering. In: ACM Genetic and Evolutionary Computation COnference (GECCO 2009), Montreal, Canada, July 8-12, pp. 1673–1680 (2009)
6. Antoniol, G., Di Penta, M., Harman, M.: Search-based techniques applied to optimization of project planning for a massive maintenance project. In: 21st IEEE International Conference on Software Maintenance, pp. 240–249. IEEE Computer Society Press, Los Alamitos (2005)
7. Arcuri, A.: It does matter how you normalise the branch distance in search based software testing. In: Proceedings of the International Conference on Software Testing, Verification and Validation, pp. 205–214. IEEE (2010)
8. Arcuri, A., Briand, L.: A practical guide for using statistical tests to assess randomized algorithms in software engineering. In: 33rd International Conference on Software Engineering (ICSE 2011), pp. 1–10. ACM, New York (2011)
9. Arcuri, A., White, D.R., Yao, X.: Multi-objective Improvement of Software Using Co-evolution and Smart Seeding. In: Li, X., Kirley, M., Zhang, M., Green, D., Ciesielski, V., Abbass, H.A., Michalewicz, Z., Hendtlass, T., Deb, K., Tan, K.C., Branke, J., Shi, Y. (eds.) SEAL 2008. LNCS, vol. 5361, pp. 61–70. Springer, Heidelberg (2008)
10. Arcuri, A., Yao, X.: Coevolving Programs and Unit Tests from their Specification. In: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), Atlanta, Georgia, USA, November 5-9, pp. 397–400. ACM (2007)
11. Arcuri, A., Yao, X.: A Novel Co-evolutionary Approach to Automatic Software Bug Fixing. In: Proceedings of the IEEE Congress on Evolutionary Computation (CEC 2008), Hongkong, China, June 1-6, pp. 162–168. IEEE Computer Society (2008)
12. Asadi, F., Antoniol, G., Guéhéneuc, Y.-G.: Concept locations with genetic algorithms: A comparison of four distributed architectures. In: Proceedings of 2nd International Symposium on Search based Software Engineering (SSBSE 2010), Benevento, Italy. IEEE Computer Society Press (2010) to appear
13. Bagnall, A.J., Rayward-Smith, V.J., Whittley, I.M.: The next release problem. Information and Software Technology 43(14), 883–890 (2001)
14. Baker, J.E.: Reducing bias and inefficiency in the selection algorithm. In: Proceedings of the 2nd International Conference on Genetic Algorithms and their Application, Hillsdale, New Jersey, USA, Lawrence Erlbaum Associates (1987)
15. Binkley, D., Harman, M., Lakhotia, K.: FlagRemover: A testability transformation for transforming loop assigned flags. ACM Transactions on Software Engineering and Methodology. (2010) to appear
16. Black, J., Melachrinoudis, E., Kaeli, D.: Bi-criteria models for all-uses test suite reduction. In: Proceedings of the 26th International Conference on Software Engineering (ICSE 2004), pp. 106–115. ACM Press (May 2004)

17. Bowman, M., Briand, L.C., Labiche, Y.: Solving the Class Responsibility Assignment Problem in Object-Oriented Analysis with Multi-Objective Genetic Algorithms. Technical Report SCE-07-02, Carleton University (August. 2008)
18. Burgess, C.J., Lefley, M.: Can genetic programming improve software effort estimation? a comparative evaluation. Information and Software Technology 43, 863–873 (2001)
19. Burke, E., Kendall, G.: Search Methodologies. Introductory tutorials in optimization and decision support techniques. Springer, Heidelberg (2005)
20. Chen, T.Y., Lau, M.F.: Heuristics towards the optimization of the size of a test suite. In: Proceedings of the 3rd International Conference on Software Quality Management, vol. 2, pp. 415–424 (1995)
21. Clark, J., Dolado, J.J., Harman, M., Hierons, R.M., Jones, B., Lumkin, M., Mitchell, B., Mancoridis, S., Rees, K., Roper, M., Shepperd, M.: Reformulating software engineering as a search problem. IEE Proceedings — Software 150(3), 161–175 (2003)
22. Crescenzi, P., Kann, V. (eds.): A compendium of NP-optimization problems, `http://www.nada.kth.se/`
23. Deb, K., Goldberg, D.: A comparative analysis of selection schemes used in genetic algorithms. In: Foundations of Genetic Algorithms, pp. 69–93. Morgan Kaufmann, San Francisco (1991)
24. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Transactions on Evolutionary Computation 6, 182–197 (2002)
25. Dolado, J.J.: On the problem of the software cost function. Information and Software Technology 43(1), 61–72 (2001)
26. Dolado, J.J.: A Validation of the Component-based Method for Software Size Estimation. IEEE Transactions on Software Engineering 26(10), 1006–1021 (2000)
27. Durillo, J.J., Zhang, Y., Alba, E., Nebro, A.J.: A Study of the Multi-Objective Next Release Problem. In: Proceedings of the 1st International Symposium on Search Based Software Engineering (SSBSE 2009), Cumberland Lodge, Windsor, UK, May 13-15, pp. 49–58. IEEE Computer Society Press (2009)
28. Elbaum, S.G., Malishevsky, A.G., Rothermel, G.: Prioritizing test cases for regression testing. In: International Symposium on Software Testing and Analysis, pp. 102–112. ACM Press (2000)
29. Fatiregun, D., Harman, M., Hierons, R.: Evolving transformation sequences using genetic algorithms. In: 4th International Workshop on Source Code Analysis and Manipulation (SCAM 2004), pp. 65–74. IEEE Computer Society Press, Los Alamitos (2004)
30. Fatiregun, D., Harman, M., Hierons, R.: Search-based amorphous slicing. In: 12th International Working Conference on Reverse Engineering (WCRE 2005), pp. 3–12. Carnegie Mellon University, Pittsburgh (2005)
31. Finkelstein, A., Harman, M., Afshin Mansouri, S., Ren, J., Zhang, Y.: "Fairness Analysis" in Requirements Assignments. In: Proceedings of the 16th IEEE International Requirements Engineering Conference (RE 2008), Barcelona, Catalunya, Spain, September 8-12, pp. 115–124. IEEE Computer Society (2008)
32. Foster, I.: Designing and building parallel programs:Concepts and tools for parallel software. Addison-Wesley (1995)
33. Sapna, P.G., Mohanty, H.: Automated Test Scenario Selection Based on Levenshtein Distance. In: Janowski, T., Mohanty, H. (eds.) ICDCIT 2010. LNCS, vol. 5966, pp. 255–266. Springer, Heidelberg (2010)

34. Garey, M.R., Johnson, D.S.: Computers and Intractability: A guide to the theory of NP-Completeness. W. H. Freeman and Company (1979)
35. Gu, Q., Tang, B., Chen, D.: Optimal regression testing based on selective coverage of test requirements. In: International Symposium on Parallel and Distributed Processing with Applications (ISPA 2010), pp. 419–426 (September 2010)
36. Harman, M.: The current state and future of search based software engineering. In: Briand, L., Wolf, A. (eds.) Future of Software Engineering 2007, pp. 342–357. IEEE Computer Society Press, Los Alamitos (2007)
37. Harman, M.: Search based software engineering for program comprehension. In: 15th International Conference on Program Comprehension (ICPC 2007), Banff, Canada, pp. 3–13. IEEE Computer Society Press (2007)
38. Harman, M.: The relationship between search based software engineering and predictive modeling. In: 6th International Conference on Predictive Models in Software Engineering, Article Number 1, Timisoara, Romania (2010) (keynote paper)
39. Harman, M.: Why the Virtual Nature of Software Makes It Ideal for Search Based Optimization. In: Rosenblum, D.S., Taentzer, G. (eds.) FASE 2010. LNCS, vol. 6013, pp. 1–12. Springer, Heidelberg (2010)
40. Harman, M.: Making the case for MORTO: Multi objective regression test optimization. In: 1st International Workshop on Regression Testing (Regression 2011), Berlin, Germany (March 2011)
41. Harman, M.: Refactoring as testability transformation. In: Refactoring and Testing Workshop (RefTest 2011), Berlin, Germany (March 2011)
42. Harman, M., Clark, J.: Metrics are fitness functions too. In: 10th International Software Metrics Symposium (METRICS 2004), pp. 58–69. IEEE Computer Society Press, Los Alamitos (2004)
43. Harman, M., Hassoun, Y., Lakhotia, K., McMinn, P., Wegener, J.: The impact of input domain reduction on search-based test data generation. In: ACM Symposium on the Foundations of Software Engineering (FSE 2007), Dubrovnik, Croatia, pp. 155–164. Association for Computer Machinery (September 2007)
44. Harman, M., Hierons, R.M.: An overview of program slicing. Software Focus 2(3), 85–92 (2001)
45. Harman, M., Jones, B.F.: Search based software engineering. Information and Software Technology 43(14), 833–839 (2001)
46. Harman, M., Krinke, J., Ren, J., Yoo, S.: Search based data sensitivity analysis applied to requirement engineering. In: ACM Genetic and Evolutionary Computation Conference (GECCO 2009), Montreal, Canada, July 8-12, pp. 1681–1688 (2009)
47. Harman, M., Lakhotia, K., McMinn, P.: A Multi-Objective Approach to Search-based Test Data Generation. In: Proceedings of the 9th annual Conference on Genetic and Evolutionary Computation (GECCO 2007), London, England, July 7-11, pp. 1098–1105. ACM (2007)
48. Harman, M., Mansouri, A., Zhang, Y.: Search based software engineering: A comprehensive analysis and review of trends techniques and applications. Technical Report TR-09-03, Department of Computer Science, King's College London (April 2009)
49. Harman, M., McMinn, P.: A theoretical and empirical analysis of evolutionary testing and hill climbing for structural test data generation. In: International Symposium on Software Testing and Analysis (ISSTA 2007), London, United Kingdom, pp. 73–83. Association for Computer Machinery (2007)

50. Harman, M., McMinn, P.: A theoretical and empirical study of search based testing: Local, global and hybrid search. IEEE Transactions on Software Engineering 36(2), 226–247 (2010)
51. Harman, M., Swift, S., Mahdavi, K.: An empirical study of the robustness of two module clustering fitness functions. In: Genetic and Evolutionary Computation Conference (GECCO 2005), Washington DC, USA, pp. 1029–1036. Association for Computer Machinery (2005)
52. Harman, M., Tratt, L.: Pareto optimal search-based refactoring at the design level. In: GECCO 2007: Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation, pp. 1106–1113. ACM Press, London (2007)
53. Jean Harrold, M., Gupta, R., Lou Soffa, M.: A methodology for controlling the size of a test suite. ACM Transactions on Software Engineering and Methodology 2(3), 270–285 (1993)
54. Holland, J.H.: Adaptation in Natural and Artificial Systems. University of Michigan Press, Ann Arbor (1975)
55. Ince, D.C., Hekmatpour, S.: Empirical evaluation of random testing. The Computer Journal 29(4) (August 1986)
56. Kirkpatrick, S., Gellat, C.D., Vecchi, M.P.: Optimization by simulated annealing. Science 220(4598), 671–680 (1983)
57. Kirsopp, C., Shepperd, M., Hart, J.: Search heuristics, case-based reasoning and software project effort prediction. In: GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference, July 9-13, pp. 1367–1374. Morgan Kaufmann Publishers, San Francisco (2002)
58. Kirsopp, C., Shepperd, M.J., Hart, J.: Search heuristics, case-based reasoning and software project effort prediction. In: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2002, pp. 1367–1374. Morgan Kaufmann Publishers Inc., San Francisco (2002)
59. Korel, B.: Automated software test data generation. IEEE Transactions on Software Engineering 16(8), 870–879 (1990)
60. Lakhotia, K., Harman, M., McMinn, P.: Handling dynamic data structures in search based testing. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2008), pp. 1759–1766. ACM Press, Atlanta (2008)
61. Lehre, P.K., Yao, X.: Runtime analysis of search heuristics on software engineering problems. Frontiers of Computer Science in China 3(1), 64–72 (2009)
62. Mahdavi, K., Harman, M., Mark Hierons, R.: A multiple hill climbing approach to software module clustering. In: IEEE International Conference on Software Maintenance, pp. 315–324. IEEE Computer Society Press, Los Alamitos (2003)
63. Maia, C.L.B., do Carmo, R.A.F., de Freitas, F.G., Lima de Campos, G.A., de Souza, J.T.: A multi-objective approach for the regression test case selection problem. In: Proceedings of Anais do XLI Simpòsio Brasileiro de Pesquisa Operacional (SBPO 2009), pp. 1824–1835 (2009)
64. Mancoridis, S., Mitchell, B.S., Rorres, C., Chen, Y.-F., Gansner, E.R.: Using automatic clustering to produce high-level system organizations of source code. In: International Workshop on Program Comprehension (IWPC 1998), pp. 45–53. IEEE Computer Society Press, Los Alamitos (1998)
65. McMinn, P.: Search-based software test data generation: A survey. Software Testing, Verification and Reliability 14(2), 105–156 (2004)
66. McMinn, P.: Search-based testing: Past, present and future. In: Proceedings of the 3rd International Workshop on Search-Based Software Testing (SBST 2011). IEEE, Berlin (to appear, 2011)

67. Mitchell, B.S., Mancoridis, S.: Using heuristic search techniques to extract design abstractions from source code. In: GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference, July 9-13, pp. 1375–1382. Morgan Kaufmann Publishers, San Francisco (2002)
68. Mitchell, B.S., Mancoridis, S.: On the automatic modularization of software systems using the bunch tool. IEEE Transactions on Software Engineering 32(3), 193–208 (2006)
69. Mitchell, B.S., Traverso, M., Mancoridis, S.: An architecture for distributing the computation of software clustering algorithms. In: IEEE/IFIP Proceedings of the Working Conference on Software Architecture (WICSA 2001), pp. 181–190. IEEE Computer Society, Amsterdam (2001)
70. Mitchell, M., Forrest, S., Holland, J.H.: The royal road for genetic algorithms: Fitness landscapes and GA performance. In: Varela, F.J., Bourgine, P. (eds.) Proc. of the First European Conference on Artificial Life, pp. 245–254. MIT Press, Cambridge (1992)
71. Mühlenbein, H., Schlierkamp-Voosen, D.: Predictive models for the breeder genetic algorithm: I. continuous parameter optimization. Evolutionary Computation 1(1), 25–49 (1993)
72. Munawar, A., Wahib, M., Munetomo, M., Akama, K.: A survey: Genetic algorithms and the fast evolving world of parallel computing. In: 10th IEEE International Conference on High Performance Computing and Communications (HPCC 2008), pp. 897–902. IEEE (2008)
73. Munroe, R.: XKCD: Significant, `http://xkcd.com/882/`
74. Offutt, J., Pan, J., Voas, J.: Procedures for reducing the size of coverage-based test sets. In: Proceedings of the 12th International Conference on Testing Computer Software, pp. 111–123 (June 1995)
75. O'Keeffe, M., Ó Cinnéide, M.: Search-based refactoring: an empirical study. Journal of Software Maintenance 20(5), 345–364 (2008)
76. Pinto, G.H.L., Vergilio, S.R.: A multi-objective genetic algorithm to test data generation. In: 22nd IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2010), pp. 129–134. IEEE Computer Society (2010)
77. Praditwong, K., Harman, M., Yao, X.: Software module clustering as a multi-objective search problem. IEEE Transactions on Software Engineering (to appear, 2011)
78. Räihä, O.: A survey on search–based software design. Computer Science Review 4(4), 203–249 (2010)
79. Reid, S.C.: An empirical analysis of equivalence partitioning, boundary value analysis and random testing. In: 4th International Software Metrics Symposium. IEEE Computer Society Press, Los Alamitos (1997)
80. Rothermel, G., Harrold, M., Ronne, J., Hong, C.: Empirical studies of test suite reduction. Software Testing, Verification, and Reliability 4(2), 219–249 (2002)
81. Rothermel, G., Harrold, M.J., Ostrin, J., Hong, C.: An empirical study of the effects of minimization on the fault detection capabilities of test suites. In: Proceedings of International Conference on Software Maintenance (ICSM 1998), Bethesda, Maryland, USA, pp. 34–43. IEEE Computer Society Press (November 1998)
82. Ruhe, G., Greer, D.: Quantitative Studies in Software Release Planning under Risk and Resource Constraints. In: Proceedings of the International Symposium on Empirical Software Engineering (ISESE 2003), Rome, Italy, September 29 - October 4, pp. 262–270. IEEE (2003)

83. Ryan, C.: Automatic re-engineering of software using genetic programming. Kluwer Academic Publishers (2000)
84. Saliu, M.O., Ruhe, G.: Bi-objective release planning for evolving software systems. In: Crnkovic, I., Bertolino, A. (eds.) Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE) 2007, pp. 105–114. ACM (September 2007)
85. Seng, O., Stammel, J., Burkhart, D.: Search-based determination of refactorings for improving the class structure of object-oriented systems. In: Genetic and Evolutionary Computation Conference (GECCO 2006), Seattle, Washington, USA, July 8-12, vol. 2, pp. 1909–1916. ACM Press (2006)
86. Shaw, M.: Writing good software engineering research papers: minitutorial. In: Proceedings of the 25th International Conference on Software Engineering (ICSE 2003), Piscataway, NJ, May 3-10, pp. 726–737. IEEE Computer Society (2003)
87. Shepperd, M.J.: Foundations of software measurement. Prentice Hall (1995)
88. Simons, C.L., Parmee, I.C.: Agent-based Support for Interactive Search in Conceptual Software Engineering Design. In: Keijzer, M. (ed.) Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation (GECCO 2008), Atlanta, GA, USA, July 12-16, pp. 1785–1786. ACM (2008)
89. Simons, C.L., Parmee, I.C., Gwynllyw, R.: Interactive, evolutionary search in upstream object-oriented class design. IEEE Transactions on Software Engineering 36(6), 798–816 (2010)
90. de Souza, J.T., Maia, C.L., de Freitas, F.G., Coutinho, D.P.: The human competitiveness of search based software engineering. In: Proceedings of 2nd International Symposium on Search based Software Engineering (SSBSE 2010), Benevento, Italy, pp. 143–152. IEEE Computer Society Press (2010)
91. Sutton, A.M., Howe, A.E., Whitley, L.D.: Estimating Bounds on Expected Plateau Size in MAXSAT Problems. In: Stützle, T., Birattari, M., Hoos, H.H. (eds.) SLS 2009. LNCS, vol. 5752, pp. 31–45. Springer, Heidelberg (2009)
92. Tonella, P., Susi, A., Palma, F.: Using interactive ga for requirements prioritization. In: Proceedings of the 2nd International Symposium on Search Based Software Engineering (SSBSE 2010), Benevento, Italy, September 7-9, pp. 57–66. IEEE (2010)
93. Tracey, N., Clark, J., Mander, K., McDermid, J.: An automated framework for structural test-data generation. In: Proceedings of the International Conference on Automated Software Engineering, Hawaii, USA, pp. 285–288. IEEE Computer Society Press (1998)
94. Turing, A.M.: Computing machinery and intelligence. Mind 49, 433–460 (1950)
95. Wada, H., Champrasert, P., Suzuki, J., Oba, K.: Multiobjective Optimization of SLA-Aware Service Composition. In: Proceedings of IEEE Workshop on Methodologies for Non-functional Properties in Services Computing, Honolulu, HI, USA, July 6-11, pp. 368–375. IEEE (2008)
96. Wang, H., Chan, W.K., Tse, T.H.: On the construction of context-aware test suites. Technical Report TR-2010-01, Hong Kong University (2010)
97. Wegener, J., Baresel, A., Sthamer, H.: Evolutionary test environment for automatic structural testing. Information and Software Technology 43(14), 841–854 (2001)
98. Wen, F., Lin, C.-M.: Multistage Human Resource Allocation for Software Development by Multiobjective Genetic Algorithm. The Open Applied Mathematics Journal 2, 95–103 (2008)

99. White, D.R., Clark, J.A., Jacob, J., Poulding, S.M.: Searching for Resource-Efficient Programs: Low-Power Pseudorandom Number Generators. In: Keijzer, M. (ed.) Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation (GECCO 2008), Atlanta, GA, USA, July 12-16, pp. 1775–1782. ACM (2008)
100. Whitley, D.: The GENITOR algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. In: Schaffer, J.D. (ed.) Proceedings of the International Conference on Genetic Algorithms, San Mateo, California, USA, pp. 116–121. Morgan Kaufmann (1989)
101. Whitley, D.: A genetic algorithm tutorial. Statistics and Computing 4, 65–85 (1994)
102. Whitley, D.: An overview of evolutionary algorithms: practical issues and common pitfalls. Information and Software Technology 43(14), 817–831 (2001)
103. Whitley, D., Sutton, A.M., Howe, A.E.: Understanding elementary landscapes. In: Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation (GECCO 2008), pp. 585–592. ACM, New York (2008)
104. Williams, K.P.: Evolutionary Algorithms for Automatic Parallelization. PhD thesis, University of Reading, UK, Department of Computer Science (September 1998)
105. Yoo, S.: A novel mask-coding representation for set cover problems with applications in test suite minimisation. In: Proceedings of the 2nd International Symposium on Search-Based Software Engineering, SSBSE 2010 (2010)
106. Yoo, S., Harman, M.: Pareto efficient multi-objective test case selection. In: International Symposium on Software Testing and Analysis (ISSTA 2007), pp. 140–150. Association for Computer Machinery, London (2007)
107. Yoo, S., Harman, M.: Using hybrid algorithm for pareto efficient multi-objective test suite minimisation. Journal of Systems and Software 83(4), 689–701 (2010)
108. Yoo, S., Harman, M.: Regression testing minimisation, selection and prioritisation: A survey. Journal of Software Testing, Verification and Reliability (to appear, 2011)
109. Yoo, S., Harman, M., Tonella, P., Susi, A.: Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge. In: ACM International Conference on Software Testing and Analysis (ISSTA 2009), Chicago, Illinois, USA, July 19-23, pp. 201–212 (2009)
110. Yoo, S., Harman, M., Ur, S.: Highly scalable multi-objective test suite minimisation using graphics card. Rn/11/07, Department of Computer Science, University College London (January 2011)
111. Zhang, Y.-Y., Finkelstein, A., Harman, M.: Search Based Requirements Optimisation: Existing Work and Challenges. In: Rolland, C. (ed.) REFSQ 2008. LNCS, vol. 5025, pp. 88–94. Springer, Heidelberg (2008)
112. Zhang, Y., Harman, M., Finkelstein, A., Mansouri, A.: Comparing the performance of metaheuristics for the analysis of multi-stakeholder tradeoffs in requirements optimisation. Journal of Information and Software Technology (to appear, 2011)
113. Zhang, Y., Harman, M., Mansouri, A.: The multi-objective next release problem. In: GECCO 2007: Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation, pp. 1129–1137. ACM Press, London (2007)