The Current State and Future of Search Based Software Engineering

Mark Harman

Mark Harman is Professor of Software Engineering and head of the Software Engineering Group at the Department of Computer Science, King's College, London, where he also directs the work of the Centre for Research on Evolution, Search and Technology (CREST). He has worked extensively on program slicing, transformation and testing and more recently he was instrumental in founding the field of search-based software engineering. Prof. Harman's work is funded by the UK Engineering and Physical Sciences Research Council (EPSRC), the European Union, DaimlerChrysler, Berlin and Vizuri Ltd., London.

The Current State and Future of Search Based Software Engineering

Mark Harman
King's College London
Strand, London, WC2R 2LS
United Kingdom

Abstract

This paper describes work on the application of optimization techniques in software engineering. These optimization techniques come from the operations research and metaheuristic computation research communities. The paper briefly reviews widely used optimization techniques and the key ingredients required for their successful application to software engineering, providing an overview of existing results in eight software engineering application domains. The paper also describes the benefits that are likely to accrue from the growing body of work in this area and provides a set of open problems, challenges and areas for future work.

1. Introduction

Search based optimization techniques have been applied to a number of software engineering activities, right across the life-cycle from requirements engineering [5], project planning and cost estimation [1, 3, 4, 16, 28, 52] through testing [6, 7, 11, 14, 15, 36, 39, 54, 61, 87], to automated maintenance [12, 32, 38, 64, 65, 70, 77, 78], service-oriented software engineering [19], compiler optimization [24, 25] and quality assessment [13, 49]. The application of optimization techniques to software testing has recently witnessed intense activity to the point where, in 2004, there was sufficient material to warrant a survey paper on this sub-area of activity [60]. However, as the list of application areas above indicates, optimization can be applied right across the spectrum of software engineering activity.

A wide range of different optimization and search techniques can and have been used. The most widely used are local search (for example [52, 57, 64]), simulated annealing (for example [13, 41, 64]), genetic algorithms (for example [7, 34, 38, 87]) and genetic programming (for example [9, 29, 28, 86]). However, no matter what search technique is employed, it is the fitness function that captures the crucial information; it differentiates a good solution from a poor one, thereby guiding the search. Several studies have concerned the analysis of widely used fitness functions and the fitness landscapes they denote [6, 42, 46, 50, 51, 52, 63, 71].

The term SBSE was coined by Harman and Jones in 2001 [40] though there was work on the application of search techniques to software engineering problems as early as 1992 [89]. Since the 2001 paper there has been an explosion of activity in this area, with SBSE being increasingly applied to a wide range of diverse areas within software engineering.

It is natural in the development of new scientific and engineering fields that there is, at first, a 'gold rush' of largely undirected activity. Such activity is characterised by enthusiasm, early important results and the excitement that goes with them. However, in order to develop the mature roots that allow the field to grow, the second phase of exploration requires a deeper understanding of problem and solution characteristics. The field of Search Based Software Engineering (SBSE) is in transition between these two phases. This FoSE article aims to provide a survey of the existing work that forms part of the ongoing gold rush, updating the previous initial survey [22]. It also sets out the goals, objectives and properties to be expected from this phase transition.

The rest of the paper is organized as follows: Section 2 summarises the key optimization techniques that are applied. It is only possible to cover a few widely used techniques in detail, while the rest are summarised with pointers to the literature. Section 3 draws out the key ingredients for successful application of optimization techniques to problems in software engineering and Section 4 illustrates SBSE in practice, with an overview of previous work on successful applications of optimization in eight areas of software engineering activity. Section 5 explains the motivation behind metaheuristic approaches to software engineering optimization. Sections 6 and 7 describe open problems and the potential benefits that are likely to accrue from future work. Section 8 provides a road map for future work in optimization for software engineering. Section 9 summarises.

2. Optimization Techniques

This section provides an overview of optimization techniques, focusing on those that have been most widely applied on software engineering. Space constraints only permit an overview. For more detail, the reader is referred to the recent survey of search methodologies edited by Burke and Kendall [17].

The section starts with classical techniques, distinguishing these from metaheuristic search. Hitherto, classical techniques have been little used as optimization techniques for software engineering problems; authors have preferred to use more sophisticated metaheuristic search techniques. However, there has been some work using classical techniques. Bagnall et al. [5] applied a branch and bound approach to a formulation of the next release problem, while Barreto et al. [8] apply it to project staffing constraint satisfaction problems. Cortellessa et al. [26] use classical optimization techniques to address decision making problems in component sourcing, optimizing properties such as quality and reliability. Del Grosso et al. [27] use a combination of classical and metaheuristic techniques to test for buffer overflow.

2.1. Classic Techniques

Linear programming (LP) is a mathematical optimization technique that is guaranteed to locate the global optimum solution. The inputs to a linear programming model are a set $\{x_1, \ldots, x_n\}$ of n real, non-negative values, called the decision variables. The goal is to maximize the value of some linear expression in these decision variables subject to a set of constraints, expressed as linear equations in the decision variables.

That is

$$\text{Maximize} \sum_{i=1}^{n} c_i x_i$$

Where $\{c_1, \ldots, c_n\}$ is a set of problem–specific coefficients, subject to a set of m constraints of the form

$$\begin{array}{c} \sum_{i=1}^{n} a_{1i} x_i \leq b_1 \\ \vdots \\ \sum_{i=1}^{n} a_{mi} x_i \leq b_m \end{array}$$

Where a_{ij} and b_i are problem determined constants. The constraints can also be expressed using $\geq$ and $=$ in place of $\leq$ and the goal can be minimization rather than maximization.

This formulation is typically applied to problems such as resource and plant allocation. It requires a clear cut determination of a single objective to be optimized and a set of well understood constraints that can be captured as a set of linear equations. If a software engineering problem can be formulated in this way, then LP is a good choice because there exist efficient LP optimization algorithms and the solution is guaranteed to be globally optimal.

If some of the decision variables are further constrained to take on only integer values, then the result is a further constrained model. Integer programming models can capture a wider set of possible problem domains, because of this additional constraint. For example, it now becomes possible to model situations in which a decision variable is constrained to be a boolean choice variable; yielding a value of either 1 or 0. However, the additional constraints can lead to model formulations that are far harder to solve than their linear programming counterparts.

One other classical technique for optimization deserves mention in this section: branch and bound. This is an approach that seeks to tame the exponential explosion that is inherent in most search problems, by a simple iterative process of branching from a current solution, while simultaneously maintaining a set of bounds that prune the possible search space as it expands through branching.

2.2. Metaheuristic Search

This section provides a brief overview of three metaheuristic search techniques that have been most widely applied to problems in software engineering: hill climbing, simulated annealing and genetic algorithms.

2.2.1 Hill Climbing

Hill climbing starts from a randomly chosen candidate solution. At each iteration, the elements of a set of 'near neighbours' to the current solution are considered. Just what constitutes a near neighbour is problem specific, but typically neighbours are a 'small mutation away' from the current solution. A move is made to a neighbour that improves fitness. There are two choices: In next ascent hill climbing, the move is made to the first neighbour found to have an improved fitness. In steepest ascent hill climbing, the entire neighbourhood set is examined to find the neighbour that gives the greatest increase in fitness. If there is no fitter neighbour, then the search terminates and a (possibly local) maxima has been found. Figuratively speaking, a 'hill' in the search landscape close to the random starting point has been climbed.

Clearly, the problem with the hill climbing approach is that the hill located by the algorithm may be a local maxima, and may be far poorer than a global maxima in the search space. For some landscapes, this is not a problem because repeatedly restarting the hill climb at a different locations may produce adequate results (this is known as multiple-restart hill climbing). Despite the local maxima problem, hill climbing is a simple technique which is both easy to implement and surprisingly effective [38, 64].

2.2.2 Simulated Annealing

Simulated annealing [62] can be thought of as a variation of hill climbing that avoids the local maxima problem by permitting moves to less fit individuals. Simulated annealing is a simulation of metallurgical annealing, in which a a highly heated metal is allowed to reduce in temperature slowly, thereby increasing its strength. As the temperature decreases the atoms have less freedom of movement. However, the greater freedom in the earlier (hotter) stages of the process allow the atoms to 'explore' different energy states.

A simulated annealing algorithm will move from some point x_1 to a *worse* point x_1' with a probability that is a function of the drop in fitness and a 'temperature' parameter that (loosely speaking) models the temperature of the metal in metallurgical annealing. The effect of 'cooling' on the simulation of annealing is that the probability of following an unfavourable move is reduced. At the end of the simulated annealing algorithm, the effect is that of pure hill climbing. However, the earlier 'warmer' stages allow productive exploration of the search space, with the hope that the higher temperature allows the search to escape local maxima. The approach has found application in several problems in search based software engineering [13, 41, 84, 64].

2.2.3 Genetic Algorithms

Genetic algorithms use concepts of population and of recombination [47]. Of all optimization algorithms, genetic algorithms have been the most widely applied search technique in SBSE, though this has largely been for historical reasons, rather than as a result of any strong theoretical indications that these approaches are in some way superior.

A generic genetic algorithm [22] is presented in Figure 1. An iterative process is executed, initialised by a randomly chosen population. The iterations are called generations and the members of the population are called chromosomes, because of their analogs in natural evolution. The process terminates when a population satisfies some pre-determined condition (or a certain number of generations have been exceeded). On each generation, some members of the population are recombined, crossing over elements of their chromosomes. A fraction of the offspring of this union are mutated and, from the offspring and the original population a selection process is used to determine the new population. Crucially, recombination and selection are guided by the fitness function; fitter chromosomes having a greater chance to be selected and recombined.

There are many variations on this overall process, but the crucial ingredients are the way in which the fitness guides the search, the recombinatory and the population based nature of the process. There is an alternative form of evolutionary computation, known as evolution strategies [76], developed independently of work on Genetic Algorithms. However, evolution strategies have not been applied often in work on SBSE. An exception is the work of Alba and Chicano [2], who show that evolution strategies may outperform genetic algorithms for some test data generation problems.

```
Set generation number, $m$:= 0
Choose the initial population of candidate solutions, $P(0)$
Evaluate the fitness for each individual of $P(0)$, $F(P_i(0))$
loop
  Recombine: $P(m) := R(P(m))$
  Mutate : $P(m) := M(P(m))$
  Evaluate: $F(P(m))$
  Select: $P(m+1) := S(P(m))$
  $m := m + 1$
  exit when goal or stopping condition is satisfied
end loop;
```

Figure 1. A Generic Genetic Algorithm

There is also a variation of genetic algorithms, called genetic programming [53], in which the chromosome is not a list, but a tree. The tree is the abstract syntax tree of a simple program that is evolved using a similar genetic model to that employed by a genetic algorithm. Genetic programs are typically imperfect programs that are, nonetheless, sufficiently good for purpose. Fitness is usually measured using a testing-based approach that seeks to find a program best adapted to its specification (expressed as a set of input/output pairs). Genetic programming has been used in SBSE to form formulæ that capture predictive models of software projects [29, 28] and in testing [86].

3. Key Ingredients for Application of Optimization Techniques to Software Engineering

There are only two key ingredients for the application of search-based optimization to software engineering problems:

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

This simplicity and ready applicability has led to a dramatic increase in research in this area. With these two simple ingredients, it is possible to apply search techniques to a novel area of software engineering and to obtain interesting and potentially important results with relative ease.

Typically, a software engineer will have a suitable representation for their problem, so the first of the pre-requisites is easily satisfied. Furthermore, many problems in software engineering have a rich and varied set of software metrics associated with them that naturally form good initial candidates for fitness functions [37].

With these two ingredients it becomes possible to implement search algorithms. The results from the search algorithms can be compared, using random search to provide as baseline data. Naturally, the aim is to beat a random search, though in some areas, such as testing, even a purely random search has been found to be not without value, even beating human-directed search in some cases [66].

This is the current state of the art in search based software engineering. New areas are regularly being addressed, formulated as search problems, and attacked using a combination of search algorithms, compared to random search as a form of 'sanity check'. This research activity is important; it widens the scope of application of these techniques to cover ever larger areas of software engineering activity. However, in order that the field does not become merely broad, but flat, some research effort needs to be directed towards theory, generalisation and characterisation to augment this breadth with depth.

4. Existing Applications of Optimization Techniques to Software engineering Applications

This section provides a brief overview of areas of software engineering to which search based optimization techniques have been applied. As the section shows, search based approaches can be applied right across the software engineering life cycle in many different ways, using many different search algorithms.

This should come as no surprise. If software engineering is truly an engineering discipline, then it would be expected that optimization techniques (widely applied in all other engineering disciplines) would also apply in software engineering. Many of the problems faced by software engineers turn out to have natural counterparts as 'standard' optimization problems. Often, of course, there are some modifications and enhancements that are required and suitable representations and fitness functions must be formulated for each problem; therein lies interesting and exciting research.

4.1. Optimizing the search for accurate cost estimates

Dolado applied genetic programming to the problem of cost estimation, using a form of 'symbolic regression' [28]. The idea was to breed simple mathematical functions that fit the observed data for project effort in terms of function points. This has the advantage that the result is not merely a prediction system, but also a function that *explains* the behaviour of the prediction system. These problems are also addressed in the FoSE on software economics [80].

Kirsopp et al. [52] also used search techniques in software project cost estimation. Their approach predicts unknown project attributes in terms of known project attributes by seeking a set of near neighbour projects that share similar values for the known attributes. This approach works well where the existing base of project data is of consistently good quality, but can perform badly where some projects and/or attributes are miss-recorded. Kirsopp at al. showed that the problem of determining a set of good predictors can be formulated as a feature subset selection problem, to which they applied a hill climbing algorithm. This work was also one of the few in the literature that has considered the properties of the search landscape (see Section 6.2).

4.2. Optimizing the search for allocations in project planning

The allocation of teams to work packages in software project planing can be thought of as an application of a bin packing problem [23]. Motivated by this observation, Antoniol et al. [3, 4] and Chicano and Alba [21] applied search algorithms to software projects. Antoniol et al. applied genetic algorithms, hill climbing and simulated annealing to the problem of staff allocation to work packages. They also considered problems of re-working and abandonment of projects, which are clearly important aspects of most software engineering projects. Antoniol et al. applied the algorithms to real world data from a large Y2K maintenance project. Chicano and Alba consider the multi objective version of the problem applied to synthetic data. The multiple objectives are combined into a single fitness function using weights for each of the component objectives.

4.3. Optimizing the search for requirements to form the next release

Requirements engineering is a vital part of the software engineering process [20], to which SBSE has also been applied. Bagnall et al. [5] formulated the 'Next Release Problem (NRP)' as a search problem. In the NRP, the goal is to find the ideal set of requirements that balance customer requests, resource constraints, and requirement interdependencies. This problem, is illustrated in Figure 2, which is taken from Bagnall et al. [5]. The set of requirements r_1 to r_7 have dependencies shown by the edges. Different customers request different sets of requirements.

Bagnall et al. applied a variety of techniques, including greedy algorithms and simulated annealing to a set of synthetic data created to model features for the next release and the relationships between them. The NRP is an example of a feature subset selection search problem.

Greer and Ruhe proposed a GA-based approach for planning software releases [35]. Like many problems in software engineering, such as project planning, NRP and regression testing, there is a relationship between feature Subset Selection problems and Feature Ordering (Prioritization) problems. A comparison of approaches (both analytical and evolutionary) for prioritizing software requirements is proposed in [48].

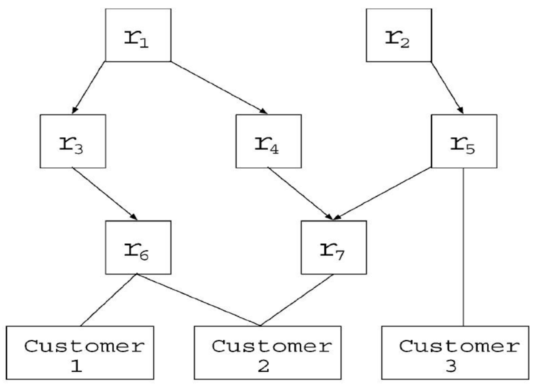

Figure 2. The next release problem, taken from [5]

4.4. Optimizing design decisions

There has been much work on the well-known software engineering concern of balancing cohesion and coupling. However, this work belongs more squarely under the heading reverse engineering [18], since the goal is to redraw the software module boundaries of *existing* designs in order to improve cohesion and coupling. Other than this, there has been only a little work on the application of search based approaches at the design stage of the software development life cycle. One notable exception is the work of Lutz [56], who considered the problem of hierarchical decomposition of software. The fitness function used by Lutz is based upon an information-theoretic formulation inspired by Shannon [79]. The function awards high fitness scores to hierarchies that can be expressed most simply (in information theoretic terms), with the aim of rewarding the more 'understandable' designs. The paper by Lutz is the only paper of which the author is aware to use information theoretic measurement as a fitness mechanism. This novel and innovative approach to fitness may have wider SBSE applications.

4.5. Optimizing source code

Some prior work has been done in the area of using metaheuristic search algorithms to search for optimization sequences. Cooper et al. [25] use biased random sampling to search a space of coarse-grained transformations for compiler optimization. The order of application of optimization steps plays a crucial role in the quality of the results and so the search problem is to identify the optimal application order. They compare the results of their experiments with those obtained using a fixed set of optimizations in a predetermined order. Ryan [74] worked on using search techniques to automate parallelization for supercomputers. He used a genetic algorithm based approach and also a genetic programming approach in which the program's correctness is determined by test cases.

Williams [88] also addressed this problem in two ways: In one, the set of transformations to apply to the program formed the chromosome. In the other approach, Williams applied the transformation steps directly to the program, as mutation steps, finding that this produced better results. Nisbet [69] focused on using a GA to find program restructuring transformations for FORTRAN programs to execute on parallel architectures. Fatiregun et al. [31, 32] showed how search based transformations could be used to reduce code size and construct amorphous program slices.

4.6. Optimizing test data generation

An entire FoSE article could be written about the application of search techniques to software test data generation and there is already a FoSE in this volume on Testing [10]. Fortunately, there is also already an excellent survey of work on search based testing [60]. Since that survey was published there has continued to be great interest in search based testing.

The general idea behind all approaches to search based test data generation is that the set of possible inputs to the program forms a search space and the test adequacy criterion is coded as a fitness function. For example, in order to achieve branch coverage, the fitness function assesses how close a test input comes to executing an uncovered branch; in order to find worst case execution time, the fitness is simply the duration of execution for the test case in question. A wide variety of testing goals have been attacked using search, including structural testing, functional and non functional testing, safety testing, robustness testing, stress testing, mutation testing, integration testing and exception testing.

4.7. Optimizing test data selection and prioritization

In test case selection, the aim is to select a set of test cases that achieve the same (or nearly the same) level of test adequacy as the entire set. Regression test prioritization seeks to order test cases used in regression testing so that test goals are achieved earlier in the sequence of test case application. These two related problems are examples of feature subset selection and prioritization problems for which search techniques and greedy algorithms have been shown to be effective.

Rothermel et al. address the prioritization problem using greedy algorithms [73, 72]. Li et al. [54] provide a comparison of greedy algorithms with other search techniques, including hill climbing and genetic algorithms finding that greedy algorithms perform best, with genetic algorithms performing well. Walcott et al. [85] applied genetic algorithms to the problem of time aware regression testing.

Mansour [68] presents results from the application to small laboratory programs of five selection algorithms for regression testing: Simulated Annealing, Reduction, Slicing, Dataflow and Firewall algorithms.

4.8. Optimizing maintenance and reverse engineering

Mancoridis et al. introduced the concept of software modularization as a clustering problem for which search is applicable [59]. This led to the development of a tool called Bunch [58] which uses a variety of search algorithms including hill climbing, simulated annealing and genetic algorithms for search based software modularisation. The goal of this work was to re-draw the module boundaries to increase cohesion and reduce coupling. Cohesion and coupling are combined into a single fitness function called MQ (modularization Quality).

The problem is essentially one of finding near cliques in a graph, the nodes of which denote modules and the edges of which denote dependence between modules. Mancoridis et al. [58] call this graph a Module Dependency Graph. The Bunch tool produces a hierarchical clustering of the graph, allowing the user to select the granularity of cluster size that best suits their application. An example of the results of clustering, using Bunch, is given in Figure 3, which is taken from Mitchell and Mancoridis [65].

Harman et al. [38], studied the effect of assigning a particular modularization granularity as part of the fitness function, while Mahdavi et al. [57] showed that combining the results from multiple hill climbs can improve on the results for simple hill climbing and genetic algorithm based approaches. Harman et al. [42] explored the robustness of the MQ fitness function in comparison with an alternative measure of cohesion and coupling used in work on clustering gene expression data.

Despite several attempts to improve on the basic hill climbing approach [38, 57, 64], this simple search technique has been found to be the most effective for this problem. Mitchell and Mancoridis recently published a survey of the Bunch project and related work [65].

Clustering is a very general problem to which a number of algorithms have been applied, not merely search based algorithms. Clustering is likely to find further application in software engineering applications, beyond the original work on software modular structure. For example, Cohen [24] recently showed how search based clustering algorithms could be applied to the problem of heap allocation java program optimization.

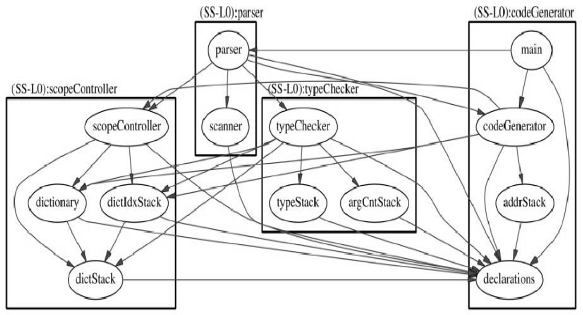

Figure 3. A Module Dependency Graph and its Modularisation using Bunch, taken from [65]

5. Motivation for Metaheuristic Search

Precise optimization algorithms such as linear programming, are straightforward deterministic algorithms. However, these deterministic optimization algorithms are often inapplicable in real world software engineering problems, because the problems have objectives that cannot be characterised by a set of linear equations. Often there are multiple criteria and complex fitness functions.

Many of the optimization problems that arise from software engineering practice are augmented versions of known NP complete problems and, as such, they are suited to the application of metaheuristic search techniques. This acknowledgment brings with it all of issues that are associated with the application of metaheuristic search techniques:

1. **Global Optimum**. There is no guarantee that the global optimum will be found.

2. **Predictability**. Each execution will potentially yield different results.

3. **Computational Expense**. A large number of individual candidate solutions may need to be considered before an acceptable quality solution is found.

Faced with these problems, it is natural to wonder how software engineering could ever profit from this application of such ‘expensive and unpredictable’ algorithms. However, as the previous section showed, there is a large community of researchers that has achieved successful results using SBSE. This section attempts to explain how this can be possible; how the three apparent daemons can be tamed.

1. **Global Optimum**
   In many applications, there is a threshold, above which a solution will be ‘good enough’ for purpose. Furthermore, optimization may not seek to find an optimal solution to a problem, but rather, it may seek to improve upon the current situation. For example, in generating test data, the goal is to cover more of the uncovered paths; in re-modularization, the goal is to improve cohesion and coupling; in cost estimation the goal is to

find models better able to predict cost and effort. In all these applications, a perfect solution, though desirable, is not essential for progress.

2. **Predictability**
   It is true that each execution of a metaheuristic search algorithm can yield different results, but all search algorithms are formulated in such a way that repeated executions can only improve on a 'best so far' result, rather then overturning a previous result. In this way the algorithms are not merely anytime algorithms they are also 'any execution'. The algorithms may be terminated at anytime and also after any number of executions to yield a results which are the 'best so far'.

3. **Computational Expense**
   Hitherto, search based optimization techniques have not been applied, on-line, to realtime software engineering problems. The kinds of problems to which search techniques seem to be readily applicable, are those where the solution is highly complex and where the software engineer is prepared to wait for an answer. While speedy answers may be attractive, they are not essential in many applications of SBSE. For example, in test data generation, the tester is prepared to wait for a set of test cases that achieve branch coverage; they can be otherwise occupied while the search algorithm seeks to find such a set. In any case, it may take a trained human several days of painstaking and potentially error-prone activity to produce such a set. The search based approach can typically achieve better results at lower cost, freeing the human to work on testing problems that require more imagination and creativity.

6. Open problems and Challenges in Optimization for Software Engineering

This section describes open problems and current work in search based software engineering. This list of topics is not exhaustive, but it gives a flavour of the richness and diversity of on-going work in this area.

6.1. Stopping Criteria

Many of the search algorithms require a stopping criterion. Typically this is taken to be some time or budget constraint on computation effort or it may be formulated as a criterion that must be met (or surpassed) by the proposed solution. Most previous work has adopted one of these two possible approaches to determining when to terminate the search.

However, the population based nature of the genetic algorithm raises a third possibility: terminate the search when the population has become homogeneous. In such a situation, where all individuals have very similar chromosomes, there is little realistic chance of further improvements in fitness. Any improvement that does occur will do so by mutation, and so it will not take advantage of the evolutionary operators.

This raises the question of how to measure solution similarity. Clearly, this is application specific. Metrics are required that can determine the similarity of a set of candidate solutions for software engineering problems. This is a challenge thrown up by SBSE to the software metrics research community. Furthermore, such metrics will need to determine similarity cheaply, for they will be applied at regular intervals during the search and to many individuals.

6.2. Landscape Visualisation

It is common in the search-based algorithm community to attempt to visualize the fitness landscape [50, 71]. A natural approach is to use the fitness function values as a measure of height in a landscape where each individual in the search space potentially occupies some location on the horizontal plane. However, most search problems involve individuals made up of more than two components (or genes in the case of genetic algorithms). Mapping an individual from the search space into a two dimensional plane is therefore non-trivial. Two approaches to visualising the search space have been used in SBSE work.

If there are only two decision variables or a projection of the search space onto only two variables is edifying, then it is possible to take a literal view of the search space. For example, consider the two search spaces depicted in Figure 4, taken from work on search based testing [61]. In this case the goal is to minimize the fitness function (depicted by height on the z axis). The left hand landscape represents a search space for which it is hard to find the global optimum, while the transformed version of this landscape in the right hand figure denotes a search space far more amenable to search. In this way visualisation can be used to explore the properties of search spaces.

In many cases, it is not possible to find a nontrivial search problem with only two decision variables. In this situation it is possible to map all n decision variables from a search space onto a flat plane in such a manner that near neighbours in the n dimensional search space lie close to one another on the 2D plane. For example consider the plot of peaks in a landscape depicted in Figure 5, taken from Kirsopp et al. [52]. This is a visualisation of the landscape from a hill climbing approach to feature subset selection. The features in this instance are software project attributes used in a case–based software project cost estimation system [52, 82]. In this figure, peaks that occur in the best quartile of the hills are denoted by 'x' symbols, those which occur in the next best quartile are denoted by 'o' symbols, those in the third quartile by '+' symbols and those in the worst quartile by '.' symbols. The division of peaks in

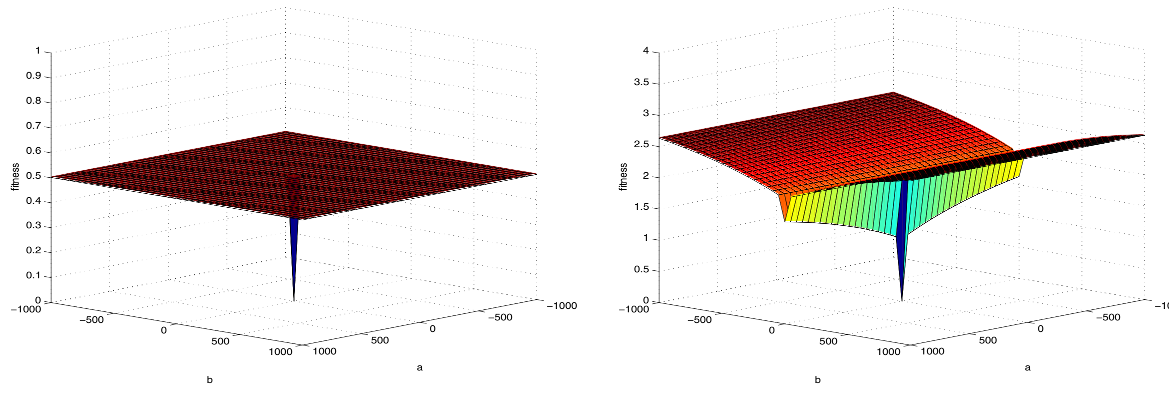

Figure 4. Transforming the Search Landscape

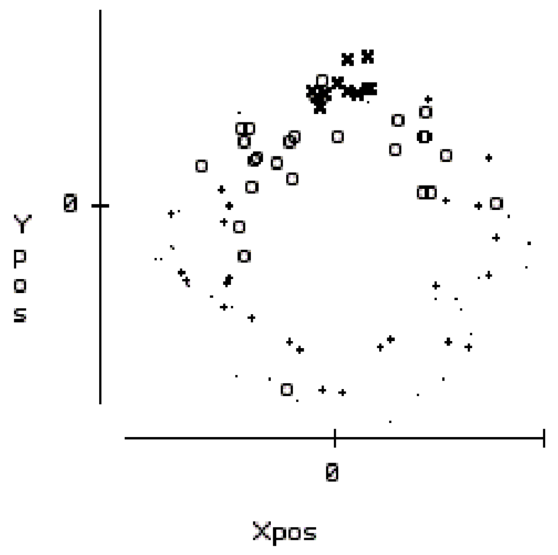

Figure 5. Search Space Fitness Landscape Peak Density Taken from Kirsopp et al.[52]

this way is coarse–grained and approximate, but nonetheless revealing: the figure clearly shows that, in this problem, high peaks tend to cluster together. Although the dimension squashing makes it impossible to say *where* this occurs on the original landscape, the peak clustering suggests a stable robust rejoin (see Section 7.2).

6.3. Characterizing the Software Engineering Search Spaces

The existing work on SBSE has produced some very valuable results, that have beaten previously available algorithms and approaches. However, the work has been somewhat ad hoc. All authors (the present one included) have tended to apply a rather arbitrary selection of search algorithms (typically employing a local and a global search, together with a random search as a baseline). This is natural for a new subject, in which the possibilities and parameters are still being explored.

What is now needed is a more concerted effort to characterise the difficulty of each of the software engineering problems for which search has already produced good results. This characterisation will help to determine the most suitable search technique to apply. It will also shed light on the nature of the search problem denoted by the software engineering application.

For new areas of software engineering that have yet to be attacked using search based approaches it remains acceptable, important even, for authors to continue to experiment with a variety of search algorithms in order to obtain baseline data and to validate the application of search. However, for the more widely visited software engineering problems such as test data generation and modularization, it will be important to build on this initial algorithmic experimentation with some more deep analysis of the nature of the search problem involved.

6.4. Human Competitive Results

Within the search optimization community, there is a great interest in results that can be said to be human competitive. The Genetic and Evolutionary Computation Conference (GECCO) recognised eight criteria (defined by Koza), that characterize the nature of human competitiveness. A result obtained by automatic computation is said to be human competitive if it meets any of the eight criteria, many of which refer to patents and existing results.

Often, SBSE is applied in areas where there is no existing best solution, for example in test data generation and modularization. In either of these fields it would seem likely that existing solutions could beat human competitors in a fair competition. One of the eight criteria is

The result holds its own or wins a regulated competition involving human contestants (in the form of either live human players or human-written computer programs).

Such experiments have not been tried in SBSE research (perhaps because the outcome is not in doubt). Nonetheless, in order to champion the field of SBSE research, it would be very attractive to have convincing examples of human competitive results, and these are sure to accrue from the growing interest and activity within this area.

7. Future Benefits to be Expected from Optimization in Software Engineering

This section briefly reviews some of the benefits that can be expected to come from further development of the field of search based software engineering. These benefits are pervading, though often implicit, themes in SBSE research. To borrow the nomenclature of Aspect Oriented Software Development, these are the 'cross cutting concerns' of the SBSE world; advantages that can be derived from almost all applications at various points in their use.

7.1. Scalability

One of the biggest problems facing software engineers is that of scalability of results. Many approaches that are attractive and elegant in the laboratory, turn out to be inapplicable in the field, because they lack scalability. However, while it is easy to ask that a proposed solution should be scalable, it is far less easy to construct a scalable solution in many cases.

One of the attractions of the search based model of optimization is that it is *naturally* parallelizable. Hill climbing can be performed in parallel, with each climb starting at a different point [57]. Genetic algorithms, being population based, are also naturally parallel; the fitness of each individual can be computed in parallel, with minimal overheads. Search algorithms in general and SBSE in particular, therefore offer a 'killer application' for the emergent paradigm of ubiquitous user–level parallel computing.

Notwithstanding a breakthrough in quantum computation technology, it seems likely that future improvements in processing speed are likely to be based on increasing parallelism. Already, limited parallelism is widely used in desktop computing and the importance of the drive toward super-fast parallel computers is recognised at the highest levels. In 2003, the Defense Advanced Research Projects Agency committed $150M to the High Productivity Computer Systems program, in which leading hardware manufacturers are now in competition to build a super-fast computer by 2010.

This trend towards greater parallelism, the need for scalable software engineering and the natural parallelism of many SBSE techniques all point to a likely significant development of parallel SBSE to address the issue of software engineering scale.

7.2. Robustness

In some software engineering applications, solution robustness may be as important as solution functionality. For example, it may be better to locate an area of the search space that is rich in fit solutions, rather than identifying an even fitter solution that is surrounded by a set of far less fit solutions.

In this way, the search seeks stable and fruitful areas of the landscape, such that near neighbours of the proposed solution are also highly fit according to the fitness function. This would have advantages where the solution needs to be not merely 'good enough' but also 'strong enough' to withstand small changes in problem character.

Hitherto, research on SBSE has tended to focus on the production of the fittest possible results. However, many application areas require solutions in a search space that may be subject to change. This makes robustness a natural 'second order property' to which the research community could and should turn its attention. For instance, in project planning, one may seek an allocation of teams to work packages that produces an early completion time, even when some of the estimates for work package duration turn out to be over-optimistic. Such an approach may produce a suboptimal solution in terms of the earliest completion time for the stated estimates of work package duration. However, this 'suboptimal' solution may, nonetheless, be optimally robust. This issue of robustness is related to the assessment of data sensitivity discussed in Section 8.3.

7.3. Feedback and Insight

False intuition is often the cause of major error in software engineering, leading to misunderstood specifications, poor communication of requirements and implicit assumptions in designs. Search based software engineering can address this problem. Unlike human–based search, automated search techniques carry with them no bias. They automatically scour the search space for the solutions that best fit the (stated) human assumptions in the fitness function.

This is one of the central strengths of the search based approach. It has been widely observed that search techniques are good at producing unexpected answers. For example, evolutionary algorithms have led to patented designs for digital filters [75] and the discovery of patented antenna designs [55]. Automated search techniques will effectively work in tandem with the human, in an iterative process of refinement, leading to better fitness functions and thereby, better encapsulation of human assumptions and intuition.

8. A Road-map for Future Work

This section provides a set of possible topic areas that the author believes are important for the development of search based software engineering research and practice. Naturally, this is a personal set of topics and is not intended to be exclusive. However, each of the topics described in this section has yet to receive widespread attention from the SBSE research community and, for each, there is the potential to reap great rewards.

8.1. Multi Objective Optimization

Software engineering problems are typically multi objective problems. The objectives that have to be met are often competing and somewhat contradictory. For example, in project planning, seeking earliest completion time at the cheapest overall cost will lead to some conflict of objectives. However, there does not necessarily exist a simple tradeoff between the two, making it desirable to find 'sweet spots' that optimize both.

Suppose a problem is to be solved that has n fitness function, $f_1, \ldots, f_n$ that take some vector of parameters $\overline{x}$. One simple–minded way to optimize these multiple objectives is to combine them into a single aggregated fitness, F, according to a set of coefficients, $c_i, \ldots, c_n$:

$$F = \sum_{i=1}^{n} c_i f_i(\overline{x})$$

This approach works when the values of the coefficients determine precisely how much each element of fitness matters. For example, if two fitness functions, f_1 and f_2 are combined using

$$F = 2 \cdot f_1(\overline{x}) + f_2(\overline{x})$$

then the coefficients $c_1 = 2, c_2 = 1$ explicitly capture the belief that the property denoted by fitness function f_1 is twice as important as that denoted by fitness function f_2. The consequence is that the search may be justified in rejecting a solution that produces a marked improvement in f_2, if it also produces a smaller reduction in the value of f_1.

Most work on SBSE uses software metrics in one form or another as fitness functions [37]. However, the metrics used are often those that are measured on an *ordinal scale* [81]. As such, it is not sensible to combine these metrics into an aggregate fitness in the manner described above.

The use of Pareto optimality is an alternative to aggregated fitness. It is superior in many ways. Using Pareto optimality, it is not possible to measure 'how much' better one solution is than another, merely to determine whether one solution is better than another. In this way, Pareto optimality combines a set of measurements into a single ordinal scale metrics, as follows:

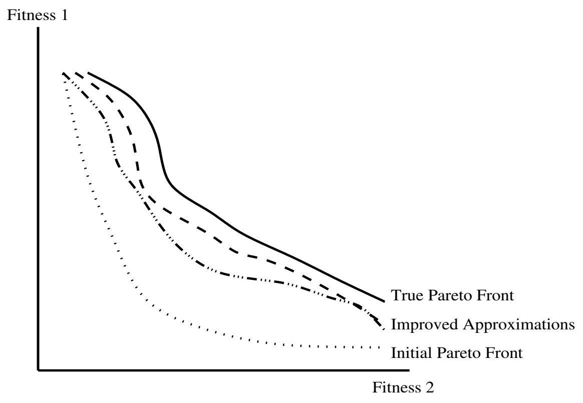

Figure 6. Pareto Optimality and Pareto Fronts

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

and, for strict inequality:

$$\begin{array}{c} F(\overline{x_1}) > F(\overline{x_2}) \\ \Leftrightarrow \\ \forall i. f_i(\overline{x_1}) \geq f_i(\overline{x_2}) \;\; \wedge \;\; \exists i. f_i(\overline{x_1}) > f_i(\overline{x_2}) \end{array}$$

Thus, 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.

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 6, which depicts the computation of Pareto optimality for two imaginary fitness functions. The longer the search algorithm is run the better the approximation becomes to the real Pareto front.

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 is less constrained. It can 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.

Often there are several candidate metrics that may be used to measure some attribute of interest. The software engineer may not be sure which is best, or even whether the metrics truly measure the same attribute. Using a Pareto optimal search algorithm, the software engineer can examine the Pareto front to see where the metrics are in agreement and where they differ. The points of difference allow the software engineer to gain insights into the behaviour of the metrics.

Current work on software metrics and measurement is rather static; viewing metrics as a mere passive measures of attributes of interest. However, SBSE opens up the possibility of a more active, dynamic approach, in which the metrics are the basis for *optimization*. For example, search could be used to aggressively seek out points of disagreement between a set of metrics, by defining fitness in terms of the level of disagreement. This technique uses SBSE as a means to validate software metrics. It allows all the tools and techniques associated with SBSE to be used to explore the search spaces implicitly denoted by metrics.

8.2. Interactive Optimization

All of the fitness functions so far considered in the literature on SBSE have been fully automated. This seems to be a pre-requisite; fast fitness computation is needed for repeated evaluation during the progress of the search. However, outside the SBSE domain of application, there has been extensive work on fitness functions that incorporate human judgement [33]. This form of search is known as interactive optimization.

In software engineering, interactive optimization could be used in a number of ways. Many problems may naturally benefit from human evaluation of fitness. For example, in design problems, the constraints that govern the design process may be ill–defined or subjective. It may also be possible to use a search based approach to explore the implicit assumptions in human assessment of solutions. For example, by identifying the building blocks that make up a good solution according to a human fitness evaluation, it may be possible to capture otherwise implicit design constraints and desirable features.

The key problem with any interactive approach to optimization lies in the requirement to repeatedly revert to the human for an assessment of fitness, thereby giving rise to possible fatigue and learning–effect bias. If this fatigue problem can be overcome in the software engineering domain (as it has in other application domains) then interactive optimization offers great potential benefits to SBSE.

8.3. Sensitivity Analysis

Many of the attributes that define the properties of a software engineering problem can only be estimated. For example, in the next release problem and in project planning, customer desirability and work package effort and duration are likely to be estimated. It would be surprising and quixotic to base an approach to software engineering upon a foundation that required perfect and accurate estimates.

SBSE has a role to play here too. In many software engineering applications, search can be used to explore the sensitivity of the solution space to the inputs. Instead of seeking to find the global optimum, one can instead ask questions such as

> Which input values perturb the shape of the search landscape most?

and

> Which input values contribute most to the location of known search space peaks?

This approach was used by Yoo [90], who introduced a pseudo–exhaustive search algorithm for software component selection (an instance of the feature subset selection problem). Yoo's algorithm partitions the search space into as set of equivalence classes that can be searched exhaustively, guaranteeing to locate the global optimum. Based on this algorithm, Yoo was able to measure sensitivity changes produced by inaccuracies in estimates of component cost. He implemented this in a tool called Mosaic, which visualizes this sensitivity information.

8.4. Hybrid Optimization Algorithms

Many problems have unpredictable landscapes. These can respond well to hybrid approaches. Though there has been work on the combination of existing non-search based algorithms [39], there has been little work on combinations of search algorithms [27, 57].

A better understanding of search landscapes may suggest the application of hybrid search techniques, which combine the best aspects of existing search algorithms. Hybrids are natural in search. For example, it is well known that, for any search application, it makes sense to conclude a run of a genetic algorithm with a simple hill climb from each solution found. After all, hill climbing is cheap and effective at locating the nearest local optima; why not apply it to the final population?

There are also other possible combinations of search algorithms that have been studied in applications to other engineering areas, for example simulated annealing and genetic algorithms have been combined to produce an annealing GA [83]. Estimation of Distribution Algorithms (EDAs) [67] are another form of hybrid search, in which the algorithm's behaviour is adapted as the search proceeds.

8.5. On Line Optimization

All applications of SBSE of which the author is aware concern what might be termed 'static' or 'offline' optimization problems. That is, problems where the algorithm is executed off line in order to find a solution to the problem in hand. This is to be contrasted with 'dynamic' or 'on line' SBSE, in which the solutions are repeatedly generated in real time and applied during the lifetime of the execution of the system to which the solution applies.

The static nature of the search problems studied in the existing literature on SBSE has tended to delimit the choice of algorithms and the methodology within which the use of search is applied. Particle Swarm Optimization [91] and Ant Colony Optimization [30] techniques have not been used in the SBSE literature. These techniques work well in situations where the problem is rapidly changing and the current best solution must be continually adapted.

For example, the paradigm of application for Ant Colony Optimization is dynamic network routing, in which paths are to be found in a network, the topology of which is subject to continual change. The ants lay and respond to a pheramone trail that allows them quickly to adapt to network connection changes.

It seems likely that the ever changing and dynamic nature of many software engineering problems would suggest possible application areas for Ant Colony Optimization and Particle Swarm Optimization techniques. It is somewhat surprising that highly adaptive search techniques like Ant Colony Optimization have yet to be applied in SBSE. However, these highly dynamic software engineering application areas have yet to be identified. Perhaps Service Oriented and Agent Oriented Software Engineering paradigms will provide candidate application areas for ant colony and particle swarm optimization.

9. Summary

This paper has provided an overview of the area of software engineering activity that has come to be known as Search Based Software Engineering (SBSE). In SBSE the goal is to use search based optimization algorithms to automate the construction of solutions to software engineering problems. SBSE also aims to better understand these problems by exploration of software engineering fitness functions and search spaces that they denote.

The first papers that applied search based optimization to software engineering problems can be traced back to the early 1990s. However, the past five years have witnessed a particularly dramatic increase in SBSE activity, with many new applications being addressed. This paper seeks to provide a brief survey of these application areas and the main results achieved so far. The paper also provides a set of topics for future research in SBSE and a description of some of the benefits that may accrue from its wider application.

10. Acknowledgements

This FoSE arose as a result of recent work undertaken by the author in collaboration with many other colleagues in the growing Search Based Software Engineering (SBSE) community. The discussions within this community have helped to form the ideas presented in this paper. Harman's work is currently funded by the EPSRC project, SEBASE (2006-2011), for which the other principal investigators are John Clark (University of York) and Xin Yao (University of Birmingham) and industrialists from DaimlerChrysler Berlin, Motorola UK and IBM UK. He is also supported by the EU Specific Targeted Research Project: EvoTest (2006-2009). This paper draws on these projects and from keynotes and tutorials on SBSE prepared in collaboration with Joachim Wegener at DaimlerChrysler [43, 44, 45]. Figures 2, 3, 4 and 5 are included by kind permission of Vic Rayward Smith, Spiros Mancoridis, Phil McMinn and Martin Shepperd respectively.

References

[1] J. Aguilar-Ruiz, I. Ramos, J. C. Riquelme, and M. Toro. An evolutionary approach to estimating software development projects. *Information and Software Technology*, 43(14):875–882, Dec. 2001.

[2] E. Alba and J. F. Chicano. Observations in using parallel and sequential evolutionary algorithms for automatic software testing. *Computers and Operations Research (COR) focused issue on Search Based Software Engineeering*. to appear.

[3] G. Antoniol, M. Di Penta, and M. Harman. A robust search–based approach to project management in the presence of abandonment, rework, error and uncertainty. In 10^{th} *International Software Metrics Symposium (METRICS 2004)*, pages 172–183, Los Alamitos, California, USA, Sept. 2004. IEEE Computer Society Press.

[4] G. Antoniol, M. D. Penta, and M. Harman. Search-based techniques applied to optimization of project planning for a massive maintenance project. In 21^{st} *IEEE International Conference on Software Maintenance*, pages 240–249, Los Alamitos, California, USA, 2005. IEEE Computer Society Press.

[5] A. Bagnall, V. Rayward-Smith, and I. Whittley. The next release problem. *Information and Software Technology*, 43(14):883–890, Dec. 2001.

[6] A. Baresel, D. W. Binkley, M. Harman, and B. Korel. Evolutionary testing in the presence of loop–assigned flags: A testability transformation approach. In *International Symposium on Software Testing and Analysis (ISSTA 2004)*, pages 108–118, Omni Parker House Hotel, Boston, Massachusetts, July 2004. Appears in Software Engineering Notes, Volume 29, Number 4.

[7] A. Baresel, H. Sthamer, and M. Schmidt. Fitness function design to improve evolutionary structural testing. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1329–1336, San Fran-

cisco, CA 94104, USA, 9-13 July 2002. Morgan Kaufmann Publishers.

[8] A. Barreto, M. Barros, and C. Werner. Staffing a sowftare project: A constraint satisfaction and optimization based approach. *Computers and Operations Research (COR) focused issue on Search Based Software Engineeering*.

[9] T. V. Belle and D. H. Ackley. Code factoring and the evolution of evolvability. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1383–1390, San Francisco, CA 94104, USA, 9-13 July 2002. Morgan Kaufmann Publishers.

[10] A. Bertolino. Software testing research: Achievements, challenges, dreams. In L. Briand and A. Wolf, editors, *Future of Software Engineering 2007*, Los Alamitos, California, USA, 2007. IEEE Computer Society Press. This volume.

[11] L. Bottaci. Instrumenting programs with flag variables for test data search by genetic algorithms. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1337–1342, New York, 9-13 July 2002. Morgan Kaufmann Publishers.

[12] S. Bouktif, G. Antoniol, E. Merlo, and M. Neteler. A novel approach to optimize clone refactoring activity. In *GECCO 2006: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, volume 2, pages 1885–1892, Seattle, Washington, USA, 8-12 July 2006. ACM Press.

[13] S. Bouktif, H. Sahraoui, and G. Antoniol. Simulated annealing for improving software quality prediction. In *GECCO 2006: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, volume 2, pages 1893–1900, Seattle, Washington, USA, 8-12 July 2006. ACM Press.

[14] L. C. Briand, J. Feng, and Y. Labiche. Using genetic algorithms and coupling measures to devise optimal integration test orders. In *SEKE*, pages 43–50, 2002.

[15] L. C. Briand, Y. Labiche, and M. Shousha. Stress testing real-time systems with genetic algorithms. In *Genetic and Evolutionary Computation Conference, GECCO 2005, Proceedings, Washington DC, USA, June 25-29, 2005*, pages 1021–1028. ACM, 2005.

[16] C. J. Burgess and M. Lefley. Can genetic programming improve software effort estimation? A comparative evaluation. *Information and Software Technology*, 43(14):863–873, Dec. 2001.

[17] E. Burke and G. Kendall. *Search Methodologies. Introductory tutorials in optimization and decision support techniques*. Springer, 2005.

[18] G. Canfora and M. Di Penta. New frontiers in reverse engineering. In L. Briand and A. Wolf, editors, *Future of Software Engineering 2007*, Los Alamitos, California, USA, 2007. IEEE Computer Society Press. This volume.

[19] G. Canfora, M. D. Penta, R. Esposito, and M. L. Villani. An approach for qoS-aware service composition based on genetic algorithms. In H.-G. Beyer and U.-M. O'Reilly, editors, *Genetic and Evolutionary Computation Conference, GECCO 2005, Proceedings, Washington DC, USA, June 25-29, 2005*, pages 1069–1075. ACM, 2005.

[20] B. Cheng and J. Atlee. From state of the art to the future of requirements engineering. In L. Briand and A. Wolf, editors, *Future of Software Engineering 2007*, Los Alamitos, California, USA, 2007. IEEE Computer Society Press. This volume.

[21] F. Chicano and E. Alba. Management of software projects with gas. In 6^{th} *Metaheuristics International Conference (MIC2005)*, Vienna, Austria, Aug. 2005.

[22] J. Clark, J. J. Dolado, M. Harman, R. M. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. Shepperd. Reformulating software engineering as a search problem. *IEE Proceedings — Software*, 150(3):161–175, 2003.

[23] E. J. Coffman, M. Garey, and D. Johnson. Approximation algorithms for bin-packing. *In Algorithm Design for Computer System Design*, 1984.

[24] M. Cohen, S. B. Kooi, and W. Srisa-an. Clustering the heap in multi-threaded applications for improved garbage collection. In *GECCO 2006: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, volume 2, pages 1901–1908, Seattle, Washington, USA, 8-12 July 2006. ACM Press.

[25] K. D. Cooper, P. J. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *Proceedings of the ACM Sigplan 1999 Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'99)*, volume 34.7 of *ACM Sigplan Notices*, pages 1–9, NY, May 5 1999. ACM Press.

[26] V. Cortellessa, F. Marinelli, and P. Potena. An optimization framework for "build–or–buy" decisions in sowftare architecture. *Computers and Operations Research (COR) focused issue on Search Based Software Engineeering*.

[27] C. Del Grosso, G. Antoniol, E. Merlo, and P. Galinier. Detecting buffer overflow via automatic test input data generation. *Computers and Operations Research (COR) focused issue on Search Based Software Engineeering*.

[28] J. J. Dolado. A validation of the component-based method for software size estimation. *IEEE Transactions on Software Engineering*, 26(10):1006–1021, 2000.

[29] J. J. Dolado. On the problem of the software cost function. *Information and Software Technology*, 43(1):61–72, Jan. 2001.

[30] M. Dorigo and C. Blum. Ant colony optimization theory: A survey. *Theoretical Computer Science*, 344(2-3):243–278, 2005.

[31] D. Fatiregun, M. Harman, and R. Hierons. Evolving transformation sequences using genetic algorithms. In 4^{th} *International Workshop on Source Code Analysis and Manipulation (SCAM 04)*, pages 65–74, Los Alamitos, California, USA, Sept. 2004. IEEE Computer Society Press.

[32] D. Fatiregun, M. Harman, and R. Hierons. Search-based amorphous slicing. In 12^{th} *International Working Conference on Reverse Engineering (WCRE 05)*, pages 3–12, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, Nov. 2005.

[33] P. Funes, E. Bonabeau, J. Herve, and Y. Morieux. Interactive multi-participant task allocation. In *Proceedings of the 2004 IEEE Congress on Evolutionary Computation*, pages 1699–1705, Portland, Oregon, 20-23 June 2004. IEEE Press.

[34] N. Gold, M. Harman, Z. Li, and K. Mahdavi. A search based approach to overlapping concept boundaries. In 22^{nd} *International Conference on Software Maintenance (ICSM 06)*, Philadelphia, Pennsylvania, USA, Sept. 2006. To appear.

[35] D. Greer and G. Ruhe. Software release planning: an evolutionary and iterative approach. *Information & Software Technology*, 46(4):243–253, 2004.

[36] Q. Guo, R. M. Hierons, M. Harman, and K. Derderian. Constructing multiple unique input/output sequences using evolutionary optimisation techniques. *IEE Proceedings — Software*, 152(3):127–140, 2005.

[37] M. Harman and J. Clark. Metrics are fitness functions too. In 10th *International Software Metrics Symposium (METRICS 2004)*, pages 58–69, Los Alamitos, California, USA, Sept. 2004. IEEE Computer Society Press.

[38] M. Harman, R. Hierons, and M. Proctor. A new representation and crossover operator for search-based optimization of software modularization. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1351–1358, San Francisco, CA 94104, USA, 9-13 July 2002. Morgan Kaufmann Publishers.

[39] M. Harman, L. Hu, R. M. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper. Testability transformation. *IEEE Transactions on Software Engineering*, 30(1):3–16, Jan. 2004.

[40] M. Harman and B. F. Jones. Search based software engineering. *Information and Software Technology*, 43(14):833–839, Dec. 2001.

[41] M. Harman, K. Steinhöfel, and A. Skaliotis. Search based approaches to component selection and prioritization for the next release problem. In 22nd *International Conference on Software Maintenance (ICSM 06)*, Philadelphia, Pennsylvania, USA, Sept. 2006. To appear.

[42] M. Harman, S. Swift, and K. Mahdavi. An empirical study of the robustness of two module clustering fitness functions. In *Genetic and Evolutionary Computation Conference (GECCO 2005)*, pages 1029–1036, Washington DC, USA, June 2005. Association for Computer Machinery.

[43] M. Harman and J. Wegener. Evolutionary testing: Tutorial. In *Genetic and Evolutionary Computation (GECCO)*, Chicago, July 2003.

[44] M. Harman and J. Wegener. Getting results with search–based software engineering: Tutorial. In 26th *IEEE International Conference and Software Engineering (ICSE 2004)*, pages 728–729, Los Alamitos, California, USA, 2004. IEEE Computer Society Press.

[45] M. Harman and J. Wegener. Search based testing. In 6th *Metaheuristics International Conference (MIC 2005)*, Vienna, Austria, Aug. 2005. To appear.

[46] E. Hart and P. Ross. GAVEL - a new tool for genetic algorithm visualization. *IEEE-EC*, 5:335–348, Aug. 2001.

[47] J. H. Holland. *Adaption in Natural and Artificial Systems*. MIT Press, Ann Arbor, 1975.

[48] J. Karlsson, C. Wohlin, and B. Regnell. An evaluation of methods for priorizing software requirements. *Information and Software Technology*, 39:939–947, 1998.

[49] T. M. Khoshgoftaar, L. Yi, and N. Seliya. A multiobjective module-order model for software quality enhancement. *IEEE Transactions on Evolutionary Computation*, 8(6):593–608, December 2004.

[50] Y.-H. Kim and B.-R. Moon. Visualization of the fitness landscape, A steady-state genetic search, and schema traces. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, page 686, San Francisco, CA 94104, USA, 9-13 July 2002. Morgan Kaufmann Publishers.

[51] Y.-H. Kim and B.-R. Moon. New usage of sammon's mapping for genetic visualization. In *Genetic and Evolutionary Computation – GECCO-2003*, volume 2723 of *LNCS*, pages 1136–1147, Berlin, 12-16 July 2003. Springer-Verlag.

[52] C. Kirsopp, M. Shepperd, and J. Hart. Search heuristics, case-based reasoning and software project effort prediction. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1367–1374, San Francisco, CA 94104, USA, 9-13 July 2002. Morgan Kaufmann Publishers.

[53] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, 1992.

[54] Z. Li, M. Harman, and R. Hierons. Meta-heuristic search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering*. To appear.

[55] D. S. Linden. Innovative antenna design using genetic algorithms. In D. W. Corne and P. J. Bentley, editors, *Creative Evolutionary Systems*, chapter 20. Elsevier, Amsterdam, The Netherland, 2002.

[56] R. Lutz. Evolving good hierarchical decompositions of complex systems. *Journal of Systems Architecture*, 47:613–634, 2001.

[57] K. Mahdavi, M. Harman, and R. M. Hierons. A multiple hill climbing approach to software module clustering. In *IEEE International Conference on Software Maintenance*, pages 315–324, Los Alamitos, California, USA, Sept. 2003. IEEE Computer Society Press.

[58] S. Mancoridis, B. S. Mitchell, Y.-F. Chen, and E. R. Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *Proceedings; IEEE International Conference on Software Maintenance*, pages 50–59. IEEE Computer Society Press, 1999.

[59] S. Mancoridis, B. S. Mitchell, C. Rorres, Y.-F. Chen, and E. R. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *International Workshop on Program Comprehension (IWPC'98)*, pages 45–53, Los Alamitos, California, USA, 1998. IEEE Computer Society Press.

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

[61] P. McMinn, M. Harman, D. Binkley, and P. Tonella. The species per path approach to search-based test data generation. In *International Symposium on Software Testing and Analysis (ISSTA 06)*, pages 13–24, Portland, Maine, USA., 2006.

[62] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller. Equation of state calculations by fast computing machines. *Journal of Chemical Physics*, 21:1087–1092, 1953.

[63] B. S. Mitchell. *A Heuristic Search Approach to Solving the Software Clustering Problem*. PhD Thesis, Drexel University, Philadelphia, PA, Jan. 2002.

[64] B. S. Mitchell and S. Mancoridis. Using heuristic search techniques to extract design abstractions from source code.

In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1375–1382, San Francisco, CA 94104, USA, 9-13 July 2002. Morgan Kaufmann Publishers.

[65] B. S. Mitchell and S. Mancoridis. On the automatic modularization of software systems using the bunch tool. *IEEE Transactions on Software Engineering*, 32(3):193–208, 2006.

[66] F. Mueller and J. Wegener. A comparison of static analysis and evolutionary testing for the verification of timing constraints. In *4th IEEE Real-Time Technology and Applications Symposium (RTAS '98)*, pages 144–154, Washington - Brussels - Tokyo, June 1998. IEEE.

[67] H. Mühlenbein and G. Paaß. From recombination of genes to the estimation of distributions: I. Binary parameters. In H.-M. Voigt, W. Ebeling, I. Rechenberg, and H.-P. Schwefel, editors, *Parallel Problem Solving from Nature – PPSN IV*, pages 178–187, Berlin, 1996. Springer.

[68] Nashat Mansour, Rami Bahsoon and G. Baradhi. Empirical comparison of regression test selection algorithms. *Systems and Software*, 57(1):79–90, 2001.

[69] A. Nisbet. GAPS: A compiler framework for genetic algorithm (GA) optimised parallelisation. In P. M. A. Sloot, M. Bubak, and L. O. Hertzberger, editors, *High-Performance Computing and Networking, International Conference and Exhibition, HPCN Europe 1998, Amsterdam, The Netherlands, April 21-23, 1998, Proceedings*, volume LNCS 1401, pages 987–989. Springer, 1998.

[70] M. O'Keeffe and M. O'Cinneide. Search-based software maintenance. In *Conference on Software Maintenance and Reengineering (CSMR'06)*, pages 249–260, Mar. 2006.

[71] H. Pohlheim. Visualization of evolutionary algorithms - set of standard techniques and multidimensional visualization. In *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 1, pages 533–540, San Francisco, CA 94104, USA, July 1999. Morgan Kaufmann.

[72] G. Rothermel, S. Elbaum, A. G. Malishevsky, P. Kallakuri, and X. Qiu. On test suite composition and cost-effective regression testing. *ACM Trans. Softw. Eng. Methodol.*, 13(3):277–331, 2004.

[73] G. Rothermel, R. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, Oct. 2001.

[74] C. Ryan. *Automatic re-engineering of software using genetic programming*. Kluwer Academic Publishers, 2000.

[75] T. Schnier, X. Yao, and P. Liu. Digital filter design using multiple pareto fronts. *Soft Computing*, 8(5):332–343, April 2004.

[76] H. Schwefel and T. Bäck. Artificial evolution: How and why? In D. Quagliarella, J. Périaux, C. Poloni, and G. Winter, editors, *Genetic Algorithms and Evolution Strategy in Engineering and Computer Science*, pages 1–19. John Wiley and Sons, 1998.

[77] O. Seng, M. Bauer, M. Biehl, and G. Pache. Search-based improvement of subsystem decompositions. In H.-G. Beyer and U.-M. O'Reilly, editors, *Genetic and Evolutionary Computation Conference, GECCO 2005, Proceedings, Washington DC, USA, June 25-29, 2005*, pages 1045–1051. ACM, 2005.

[78] O. Seng, J. Stammel, and D. Burkhart. Search-based determination of refactorings for improving the class structure of object-oriented systems. In *GECCO 2006: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, volume 2, pages 1909–1916, Seattle, Washington, USA, 8-12 July 2006. ACM Press.

[79] C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423 and 623–656, July and October 1948.

[80] M. Shepperd. Software economics. In L. Briand and A. Wolf, editors, *Future of Software Engineering 2007*, Los Alamitos, California, USA, 2007. IEEE Computer Society Press. This volume.

[81] M. J. Shepperd. *Foundations of software measurement*. Prentice Hall, 1995.

[82] M. J. Shepperd and C. Schofield. Estimating software project effort using analogies. *IEEE Transactions on Software Engineering*, 23(11):736–743, 1997.

[83] M. Tang and J. Dong. Simulated annealing genetic algorithm for surface intersection. In *Advances in Natural Computation*, volume 3612 of *Lecture Notes in Computer Science*, pages 48–56. Springer, Aug. 2005.

[84] N. Tracey, J. Clark, and K. Mander. Automated program flaw finding using simulated annealing. In *International Symposium on Software Testing and Analysis (ISSTA 98)*, pages 73–81, March 1998.

[85] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos. Time aware test suite prioritization. In *International Symposium on Software Testing and Analysis (ISSTA 06)*, pages 1 – 12, Portland, Maine, USA., 2006. ACM Press.

[86] S. Wappler and J. Wegener. Evolutionary unit testing of object-oriented software using strongly-typed genetic programming. In *GECCO 2006: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, volume 2, pages 1925–1932, Seattle, Washington, USA, 8-12 July 2006. ACM Press.

[87] J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology Special Issue on Software Engineering using Metaheuristic Innovative Algorithms*, 43(14):841–854, 2001.

[88] K. P. Williams. *Evolutionary Algorithms for Automatic Parallelization*. PhD thesis, University of Reading, UK, Department of Computer Science, Sept. 1998.

[89] S. Xanthakis, C. Ellis, C. Skourlas, A. Le Gall, S. Katsikas, and K. Karapoulios. Application of genetic algorithms to software testing (Application des algorithmes génétiques au test des logiciels). In 5^{th} *International Conference on Software Engineering and its Applications*, pages 625–636, Toulouse, France, 1992.

[90] S. Yoo. The use of a novel semi-exhaustive search algorithm for the analysis of data sensitivity in a feature subset selection problem. Master's thesis, King's College London, Department of Computer Science, 2006.

[91] X. Zhang, H. Meng, and L. Jiao. Intelligent particle swarm optimization in multiobjective optimization. In *Proceedings of the 2005 IEEE Congress on Evolutionary Computation*, volume 1, pages 714–719, Edinburgh, UK, Sept. 2005. IEEE Press.