JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. 2000; **00**:1–7 *Prepared using* smrauth.cls *[Version: 2003/05/07 v1.1]*

Search-Based Refactoring: an empirical study

Mark O'Keeffe*,† and Mel Ó Cinnéide*,‡

School of Computer Science & Informatics, University College Dublin, Ireland.

SUMMARY

Object-oriented systems that undergo repeated addition of functionality commonly suffer a loss of quality in their underlying design. This problem must often be remedied in a costly refactoring phase before further maintenance programming can take place. Recently search-based approaches to automating the task of software refactoring, based on the concept of treating object-oriented design as a combinatorial optimisation problem, have been proposed. However, because search-based refactoring is a novel approach it has yet to be established which search techniques are most suitable for the task. In this article we report the results of an empirical comparison of simulated annealing, genetic algorithms and multiple ascent hill-climbing in search-based refactoring. A prototype automated refactoring tool is employed, capable of making radical changes to the design of an existing program in order that it conform more closely to a contemporary quality model. Results show multiple-ascent hill climbing to outperform both simulated annealing and genetic algorithm over a set of five input programs. Copyright © 2000 John Wiley & Sons, Ltd.

KEY WORDS: search-based software engineering; automated design improvement; refactoring tools

1. Introduction

Object-oriented systems that undergo repeated addition of functionality commonly suffer a loss of quality in their underlying design. This problem, known as *software decay* [9] or *design erosion* [26], occurs when changes are made to a program without due consideration to its overall structure and design rationale. Design erosion can be combatted by refactoring, or improving the design of a program without changing its behaviour, but even with the use of

*Correspondence to: Computer Science & Informatics Centre, University College Dublin, Belfield, Dublin 4, Ireland.
†E-mail: mark.okeeffe@ucd.ie
‡E-mail: mel.ocinneide@ucd.ie

Received 30 August 2007

contemporary programming tools this requires significant effort on the part of the maintenance programmer.

Recently, search-based approaches to automating the task of software refactoring have been proposed by the authors [26] and Seng et al [29]. These approaches are based on the concept of object-oriented design as a combinatorial optimisation problem, where a fitness function defining design quality is constructed from a weighted sum of object-oriented metrics, and are inspired by the successful application of search-based approaches in other areas of software engineering such as subsystem clustering and test-data generation.

Once formulated suitably as a solution representation, change-effecting operator and fitness function, the problem of automated refactoring can be tackled using a wide variety of search techniques. As is the case with other search-based software engineering applications, however, the effectiveness of the various techniques may vary considerably. Because search-based refactoring is a novel approach it remains to be established which search techniques are most suitable in the general case; while the authors have compared the differing performance of hill-climbing and simulated annealing searches in two case studies [26], no thorough comparison of the effectiveness of local and evolutionary search techniques for this problem has yet been carried out[†].

In this paper we report the results of an empirical comparison of simulated annealing, genetic algorithm and multiple ascent hill-climbing searches. We have extended the CODe-Imp search-based refactoring tool [26] to employ a genetic algorithm search with a similar representation, crossover operator and mutation operator to that described by Seng et al [29], as well as increasing the power of the tool by adding to the number of different refactorings available for use in searching for a superior design.

The remainder of this paper is structured as follows: in section 2 we outline related work in search-based software engineering, such as module clustering, as well as discussing the state of the art in search-based refactoring. In section 3 we describe the experimental methodology for the study reported here, including the CODe-Imp tool, solution representation, change operators, fitness function, search techniques and input programs. Section 4 contains the results of the study, comprised of comparisons of various parameter sets for each of the search techniques employed, and a comparison of the relative performance of these techniques. We conclude and suggest some directions for future work in section 5.

2. Related Work

Search-Based Software Engineering (SBSE) can be defined as the application of search-based approaches to solving optimisation problems in software engineering [11]. Such problems include *module clustering*, where a software system is reorganised into loosely coupled clusters of highly cohesive modules to aid reengineering [8, 12, 18, 23], test data generation [16, 20],

[†]This article expands on the GECCO 2007 paper *Getting the Most from Search-Based Refactoring* [27]. A larger set of input programs, greater number of data points in each experiment and more detailed discussion of results and conclusions are the primary contributions of this article over the previous paper.

automated testing [31] and project management problems such as requirements scheduling [1, 33] and project cost estimation [3, 6, 7]. Clarke et al [5] and Harman et al [10, 11] provide thorough reviews of work in the field.

The great strength of search-based software engineering is that it can address problems where it is unclear how exactly an optimal solution can be reached; all that is required is that alternative candidate solutions can be examined and ranked by means of an evaluation/fitness function [5]. This is a particularly useful trait in problem domains with conflicting or competing goals, the expectation that there is no perfect solution, and many potential solutions to consider – a common situation in software engineering [14]. In this context the equivalence of software metrics and fitness functions has been pointed out [11], an idea that leads one to consider the possibility that any software property that can be measured can, in turn, be optimised.

Once a software engineering task is framed as a search problem, by defining it in terms of solution representation, fitness function and change operator, there are numerous approaches that can be applied to solving that problem, from local searches such as exhaustive search and hill-climbing to meta-heuristic searches such as genetic algorithms (GA) and ant colony optimisation. Module clustering, for example, has been addressed using exhaustive search [19], hill-climbing [12, 17, 19, 22], genetic algorithms [8, 12, 19, 22] and simulated annealing (SA) [22]. In those studies that compared search techniques, hill-climbing was, perhaps surprisingly, found to produce better results than meta-heuristic GA searches [12, 21]. These results were echoed in search-based auto-parallelisation [32], where local searches also out-performed GA. In software clustering the meta-heuristic *simulated annealing* search was found by Mitchell et al. [22] to perform similarly to hill-climbing in terms of solution quality, but better in terms of search efficiency.

The concept of treating object-oriented design as a combinatorial optimisation problem that can be solved using a search-based approach was introduced by the authors [25], and later small case studies based on the QMOOD quality model [2] were conducted which suggested that both simulated annealing and hill-climbing are effective in solving this problem [26]. Seng et al [29] describe a similar approach but use a genetic algorithm to solve the combinatorial optimisation problem. The evaluation function employed by Seng is novel rather than previously validated, but is based on well-known metrics such as *Response For a Class* (RFC) and *Weighted Methods per Class* (WMC) from the Chidamber & Kemerer [4] object-oriented metrics suite, among others. The authors report success in automatically repositioning displaced methods in the class structure using a GA search. However, as only the Move Method refactoring ([9], p.142) is considered the extent of change within the class structure is limited. Harman & Tratt [13] propose an alternative basis for search-based refactoring, using the concept of *Pareto optimality*, where one design is considered superior to an alternative only if all its metric values are at least as good, and at least one metric value is better. This approach can help avoid some practical and measurement-theoretical problems with evaluation functions based on a weighted sum of metrics, but to date has been implemented only in a refactoring tool limited to the Move Method refactoring.

By its very nature the search-based approach to solving software engineering problems involves the reuse of existing knowledge. Solution representations can be reused for different problems, existing change operators can be the basis for change operators in a new domain, and well-known search algorithms can be used to seek superior solutions. For this reason,

experience gained in addressing one SBSE problem can often be of benefit to the field as a whole. The work described in this article provides a thorough comparison of the strengths and weaknesses of several search techniques in the domain of search-based refactoring, and also an illustration of the application of search-based techniques to a software engineering problem characterised by a highly complex solution representation and a change operator of limited and varying applicability. This will be discussed further in the following sections.

3. Experimental Methodology

In this section we describe CODe-Imp, a prototype search-based refactoring tool designed to facilitate experimentation in automatically improving the design of existing programs. In common with other search-based software engineering applications, search-based refactoring requires a solution representation, a change operator that allows for movement in the space of alternative solutions and a fitness or evaluation function that allows solutions to be ranked in terms of desirability. With these three elements in place, various search techniques can be applied in solving the problem. In sections 3.1, 3.2 and 3.3 respectively we describe the solution representation, change operator and fitness function employed in this study. In section 3.4 we briefly discuss the four search techniques employed, while in section 3.5 we describe the input programs used. Due to space constraints, precise details of quality metrics, automated refactorings and search algorithms are omitted from this report. We refer the interested reader to "Search-Based Refactoring for Software Maintenance" [24] which contains a more detailed description of our experimental methodology than can be provided here.

3.1. Solution Representation

In search–based refactoring, the solution representation can be a program itself, its Abstract Syntax Tree (AST) or a more abstract model. The key requirements are that it must be possible to determine what transformations can be made to the representation in order to move through the space of alternative solutions, and it must be possible to apply corresponding refactorings to the program in question in order to implement the solution.

This study employs the tool CODe-Imp (Combinatorial Optimisation Design-Improvement), developed by the authors in order to test the thesis that the maintainability of object-oriented programs can be improved by automatically refactoring them to adhere more closely to a pre-defined quality model. CODe-Imp takes Java 1.4 source code as input and extracts design metric information via a Java Program Model (JPM), calculates quality values according to an evaluation or fitness function and effects change in the current solution by applying refactorings to the AST as required by a given search technique. Output consists of the refactored input code as well as a design improvement report including quality change and metric information [26].

The CODe-Imp JPM is a comprehensive model, with 57 different node types, but nevertheless is a cleaner and more abstract view of a Java program than the AST used, which features 90 different node types. The two main differences between a JPM node and the AST node (or subtree) that it represents are: the JPM node does not contain the list of

tokens forming the code it models, whereas the AST does, and the JPM node contains lists of the attributes, methods and constructors that are accessed/called by that node.

The three functions of the solution representation are therefore split between the JPM and AST. When the evaluation function compares two solutions, it is the JPM which is examined because it contains information regarding the interdependence of attributes, methods and constructors as well as structural information such as number of methods per class. Similarly, it is the JPM which must be queried in order to determine which refactorings can legally be applied by the change operator. Finally, it must be possible to translate the virtual solution back into a concrete solution, in this case Java code. This is the responsibility of the AST.

3.2. Change Operator

In the context of search–based refactoring, the change operator is a transformation of the solution representation that corresponds to a refactoring that can be carried out on the source code. The refactoring configuration of CODe-Imp, which was extended by the authors for the experiments reported here, consists of the fourteen refactorings described below. In CODe-Imp, complementary pairs of refactorings are employed in order that all changes made to the input design during the course of the search be reversible. This is a requirement of some search techniques that must move freely through the solution space, such as simulated annealing. All refactorings employed operate at the method/field level of granularity and higher, in order to focus on improvement of design rather than implementation issues such as correct factorisation of methods.

- **Push Down Field** moves a field from some class to those subclasses that require it.
- **Pull Up Field** moves a field from some class(es) to the immediate superclass.
- **Push Down Method** moves a method from some class to those subclasses that require it.
- **Pull Up Method** moves a method from some class(es) to the immediate superclass.
- **Extract Hierarchy** adds a new subclass to a non-leaf class C in an inheritance hierarchy. A subset of the subclasses of C will inherit from the new class.
- **Collapse Hierarchy** removes a non-leaf class from an inheritance hierarchy.
- **Increase Field Security** increases the security of a field from public to protected, protected to package, or from package to private.
- **Decrease Field Security** decreases the security of a field from private to package, package to protected, or from protected to public.
- **Increase Method Security** increases the security of a method from public to protected, protected to package, or from package to private.
- **Decrease Method Security** decreases the security of a method from private to package, package to protected, or from protected to public.
- **Replace Inheritance with Delegation** replaces an inheritance relationship between two classes with a delegation relationship; the former subclass will have a field of the type of the former superclass.

- **Replace Delegation with Inheritance** replaces a delegation relationship between two classes with an inheritance relationship; the delegating class becomes a subclass of the former delegate class.
- **Make Superclass Abstract** declares a constructor-less class explicitly abstract.
- **Make Superclass Concrete** removes the explicit declaration of an abstract class without abstract methods.

These refactorings are defined precisely in terms of the preconditions used to test for their applicability, the actions taken to perform them and the postconditions that apply after they are performed. Space constraints preclude us from reproducing these definitions here.

In CODe-Imp, behaviour preservation is input/output, meaning that the refactorings employed do not alter the output of a given program, for any possible input, as long as the necessary preconditions for the refactoring apply. Preconditions are checked in CODe-Imp using static program analysis. Static analysis is considered a conservative approximation of the actual behaviour of a program; for example, when we analyse statically we consider that any method that could be called, considering the program syntax, *is* called. In contrast, dynamic program analysis involves running the program in question and recording which methods *are* called, for a certain input. Refactoring using dynamic analysis requires access to the test suite of the program, and furthermore requires that the test suite defines the required functionality of that program – otherwise 'refactorings' could be applied that would preserve behaviour for all tests, but cause the program not to meet requirements. Furthermore, because unit tests can rely on the internal structure of a program, it is possible that refactorings that preserve input/output behaviour for all possible inputs can invalidate test cases [28], and therefore could not be applied by a search-based refactoring tool employing dynamic analysis.

3.3. Fitness Function

The fitness function employed here is an implementation of the Understandability function from Bansiya's QMOOD[‡] hierarchical design quality model [2], consisting of a weighted sum of metric quotients between two designs. Use of this design quality evaluation function was previously found by the authors to result in tangible improvements to object-oriented program design in the context of search-based refactoring [26]. The QMOOD model includes the eleven metrics described below. The weight for each metric in the Understandability function is listed beside each metric acronym; metrics are weighted positively where high values are considered to contribute to understandability and negatively where low values contribute.

- **Data Access Metric (DAM, 0.33)** The ratio of the number of non-public attributes to the total number of attributes declared in the class. This metric corresponds to the property *encapsulation*.
- **Cohesion Among Methods of Class (CAM, 0.33)** The relatedness among methods of a class, computed using the summation of the intersection of parameters of a method

[‡]Quality Model for Object-Oriented Design

with the maximum independent set of all parameter types in the class. This metric corresponds to the property *cohesion*.

- **Number Of Methods (NOM, -0.33)** A count of all the methods defined in a class. This metric corresponds to the property *complexity*.
- **Number of Polymorphic Methods (NOP, -0.33)** A count of the number of methods that can exhibit polymorphic behaviour. This metric corresponds to the property *polymorphism*.
- **Direct Class Coupling (DCC, -0.33)** A count of the different number of classes that a class is directly related to. The metric includes classes that are directly related by attribute declarations and message passing (parameters) in methods. This metric corresponds to the property *coupling*.
- **Design Size in Classes (DSC, -0.33)** A count of the total number of classes in the design, including inner classes. This metric corresponds to the property *design size*.
- **Average Number of Ancestors (ANA, -0.33)** The average number of classes from which each class inherits information. This metric corresponds to the property *abstraction*.

In addition, metrics for the design properties *messaging*, *inheritance*, *aggregation* and *hierarchies* are included in the QMOOD model but unweighted in the Understandability function. The form of QMOOD evaluation functions is shown in equation 1, where $p1$ and $p2$ are object-oriented programs, w_m is the weight on the metric m for that evaluation function, and $\text{metric}_m(p)$ is the value for metric m on program p. In this study, the QMOOD Understandability function is used to give a relative quality value between some refactored program and the same program as it existed before refactoring.

$$\text{q}_p = \sum_{m=1}^{n} w_m \text{metric}_m(p) \tag{1}$$

3.4. Search Techniques

3.4.1. Multiple Ascent Hill-Climbing

A variation on the standard first ascent hill-climbing algorithm, shown below, multiple ascent hill-climbing (HCM) is capable of achieving improved results due to its ability to escape from local optima. HCM initially acts identically to a first ascent hill-climbing search, but when a local optimum is reached a pre–defined number of random refactorings are carried out in order to move away from that point in the solution space. The search is then restarted from the randomly chosen solution. This procedure is repeated a set number of times, depending on the *number of descents* parameter. The number of random refactorings made each time is the *descent depth* parameter. This search technique is considered a primary candidate for search-based refactoring because local searches have been shown to be effective in the similar domain of module clustering [12, 21], as well as in previous exploratory work [26].

Algorithm 1 Multiple (First) Ascent Hill-Climbing

```
 1: currentNode = startNode;
 2: bestNode = currentNode;
 3: for all d such that 0 ≤ d ≤ numDescents do
 4:    madeAscent = TRUE;{First ascent hill climbing begins}
 5:    while madeAscent do
 6:       madeAscent = FALSE;
 7:       L = NEIGHBOURS(currentNode);
 8:       for all x in L do
 9:          if EVAL(x) > EVAL(currentNode) then
10:             currentNode = x;
11:             madeAscent = TRUE;
12:             break for
13:          end if
14:       end for
15:    end while{First ascent hill climbing ends}
16:    if EVAL(currentNode) > EVAL(bestNode) then
17:       bestNode = currentNode;
18:    end if
19:    if d < numDescents then
20:       for all p such that 1 ≤ p ≤ descentDepth do
21:          L = NEIGHBOURS(currentNode);
22:          currentNode = some x in L;
23:       end for
24:    end if
25: end for
26: return bestNode;
```

3.4.2. Simulated Annealing

Simulated Annealing (SA) is a meta-heuristic search technique inspired by the metallurgic process of annealing, where a molten metal is cooled slowly in order to produce or preserve certain characteristics in the solid form [15]. SA has been applied to a wide range of search-based software engineering problems, and has been found to be effective in the context of software clustering [22]. SA has the advantage that it is very robust against local optima in the search space, but the disadvantage that its many parameters can make it hard to configure for any given problem.

A simulated annealing search essentially involves making series of tentative changes to some solution of a combinatorial optimisation problem. Changes which increase the quality of the solution are accepted, and the changed solution becomes the starting point for the next series of tentative changes. In addition, some changes which reduce the quality of the solution are accepted in order to allow the search to escape from local minima. Such (negative) changes

are accepted with a probability that decreases steadily during the annealing process (equation 2; where p is the probability of accepting a given solution, δq is the magnitude of quality reduction relative to the current solution, and T is the temperature value).

$$p = e^{-\frac{\delta q}{T}} \tag{2}$$

Algorithm 2 Simulated Annealing, exponential cooling schedule

```
1: T = T_start;
2: currentNode = startNode;
3: bestNode = currentNode;
4: tentativeNode = NULL;
5: while T > 1 − T_start do
6:   for m = 1 to M do
7:     L = NEIGHBOURS(currentNode);
8:     tentativeNode = some x in L;
9:     if EVAL(tentativeNode) > EVAL(currentNode) then
10:      currentNode = tentativeNode;
11:      if EVAL(currentNode) > EVAL(bestNode) then
12:        bestNode = currentNode;
13:      end if
14:    else
15:      δq = EVAL(tentativeNode) − EVAL(currentNode);
16:      r = RANDOM(0 to 1);
17:      if r < e^(−δq/T) then
18:        currentNode = tentativeNode;
19:      end if
20:    end if
21:  end for
22:  T = T∗ coolingFactor;
23: end while
24: return bestNode;
```

3.4.3. Genetic Algorithm

For this study we have extended the set of search techniques available to CODe-Imp by including a Genetic Algorithm (GA) implementation similar to that of Seng et al. [29]. For the purpose of genetic algorithm implementation, the standard solution representation (the AST) can be considered the *phenotype*, while the sequence of refactorings carried out in order to reach that solution can be considered the *genotype*. The mutation operator employed here is identical to that described by Seng, and simply consists of adding one random refactoring to the genotype. Our crossover operator is similar to Seng's, and consists of 'cut and splice' crossover of two genotypes, so the length of the offspring genotypes can differ from that of

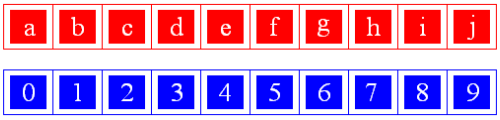

Figure 1. Crossover step one – two genotypes are selected for mating. Each box containing a letter or number represents one refactoring applied to the starting solution.

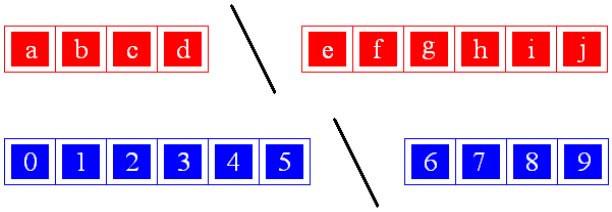

Figure 2. Crossover step two – random crossover points are independently selected in the two genotypes.

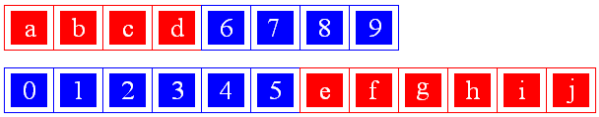

Figure 3. Crossover step three – the end of the blue genotype is added to the start of the red genotype and vice versa. Two new genotypes are formed, but may not represent valid solutions.

the parents. It is of course likely that in the process of splicing genotypes we will encounter a situation where the necessary preconditions for some refactoring are not met. In such a case we discard the refactoring in question, rather than discard the entire genotype. The operation of the crossover operator is illustrated in figures 1 to 4.

3.4.4. Steepest Ascent Hill-Climbing

Although not considered a primary candidate for use in search-based refactoring due to its long run-times and inability to escape local optima, the performance of steepest ascent hill-climbing was used as a reference point in this study in order to carry out normalisation of results across

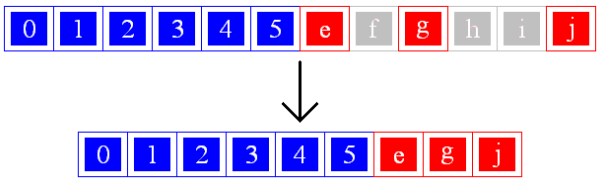

Figure 4. Crossover step four – the new genotypes are made valid by discarding refactorings whose preconditions are not met at the point where they are applied to the solution.

different input programs. Because the extent to which a design can be improved varies greatly depending on such factors as how many refactorings can legally be applied, this normalisation was vital in order to establish relative performance of the search techniques mentioned above in the general case. Steepest ascent hill-climbing provided an ideal reference point due to its deterministic nature, as the same quality gain is obtained from each run on a given input program.

Algorithm 3 Genetic Algorithm

1: Generation number $(g) = 0$
2: Population consists of *populationSize* identical (starting) solutions.
3: The genotype of each solution is empty.
4: **while** $g < g_{max}$ **do**
5: $g = g + 1$
6: With chance *mutChance*, apply mutation operator to each member of the population.
7: Re-order population according to fitness.
8: **for** $populationSize * mateProportion$ offspring **do**
9: Select first parent starting with fittest solution; each solution has chance *mateChance* to be accepted.
10: Select second parent starting with next-fittest solution to first parent; each solution has chance *mateChance* to be accepted.
11: Apply crossover operator to parents and add offspring to population.
12: **end for**
13: Re-order population according to fitness.
14: Eliminate least fit solutions in order to maintain population size *populationSize*.
15: **end while**
16: **return** fittest solution discovered

program	classes	SLOC	hierarchies	refactorings
SpecCheck	41	4836	5	351
Beaver	93	4999	9	177
EAOP	27	1164	3	200
Mango	51	1131	0	28
Grammatica	75	7000	7	761

Table I.

3.5. Input

Input consisted of five Java 1.4 programs; four randomly selected from java-source.net (hosted on SourceForge[§]), and a self-contained subset of the *Spec-Benchmarks*[¶] standard performance evaluation framework, to which it was known a large number of refactorings could be applied. The programs selected were as follows:

A *SpecCheck*, a benchmarking program
B *Beaver*, a parser generator
C *EAOP*, an aspect-oriented programming library
D *Mango*, a collections library
E *Grammatica*, a Java parser generator

and are described in table I, where 'classes', 'SLOC' and 'hierarchies' indicate the size of the input program in classes, Source Lines Of Code, and inheritance hierarchies respectively, and 'refactorings' indicates the number of refactorings that can initially be applied to the input program in its original form.

4. Results

Experiments were carried out on a 2.2GHz AMD Athlon powered PC with 2GB CL2 RAM. Mean processing time per solution examined was less than one second, including model building, metric extraction, quality assessment, discovery of legal refactorings, and actual (AST) refactoring. Total run-time varied between six minutes and eighteen hours as discussed below, depending on the search technique employed, number of refactorings possible for the input program and the number of refactorings applied. However, CODe-Imp was designed for robustness rather than speed and makes no use of concurrent processes, so there is potential to greatly decrease these run-times.

[§]http://sourceforge.net/
[¶]http://www.spec.org/

Statistical analyses were carried out using Graphpad Prism[‖]. A confidence interval of 95% was used in all statistical tests. Error bars on all figures indicate standard deviation from the mean. In the following three sections, experiments aimed at discovering suitable parameters for multiple ascent hill-climbing, simulated annealing and genetic algorithm searches in search-based refactoring are reported. We compare the performance of these three search techniques, and the straightforward *steepest ascent* hill climbing search, in section 4.4.

4.1. Multiple Ascent Hill-Climbing

In order to determine suitable parameters for multiple ascent hill-climbing (HCM) with the QMOOD Understandability function, the set of input programs were automatically refactored using CODe-Imp under the sixteen permutations of *number of descents* 1, 2, 3 or 4 and *descent depth* 5, 10, 15 or 20.

Figure 5 shows the mean normalised quality gain over the set of input programs with sixteen different multiple ascent hill-climbing configurations. Three replications were performed for each configuration, for each input. The mean quality gain value for each input/configuration pair was taken as a single n, so variance seen in figure 5 is due solely to the differing results between input programs. Results were normalised against the constant quality gain obtained from steepest ascent hill-climbing (HCS); a normalised quality gain of one for a HCM configuration corresponds to equal performance with HCS over the set of input programs.

Analysis of these data using Kruskal-Wallis tests revealed a statistically significant variation in median quality gain for the number of descents parameter (p=0.0186). Dunn's post-tests revealed a statistically significant difference between HCM search configurations including number of descents parameter values of 1 and 3, with the latter producing greater quality gains. HCM was determined to produce smaller quality gains than HCS for all configurations including a number of descents of one, by a statistically significant margin, using Wilcoxon signed rank tests. No significant difference was observed between the performance of HCS and HCM for any configuration including a number of descents of two or more.

These results support the recommendation of multiple ascent hill-climbing configurations including a *number of descents* parameter value of two, giving a total of three ascents, for search-based refactoring under the QMOOD Understandability evaluation function. Configurations including only one descent are seen to perform poorly compared to steepest ascent hill-climbing. Increasing the number of descents above two does not result in greater mean quality gains, though it may result in greater variation and therefore greater quality gains in individual runs.

The *descent depth* parameter does not appear to have a predictable effect on mean quality gain within the range examined. A value of approximately five can therefore be recommended for this parameter in search-based refactoring under the QMOOD Understandability evaluation function, based on the higher computational cost of the other parameter values examined.

[‖] GraphPad Software, Inc., 2005

[p]

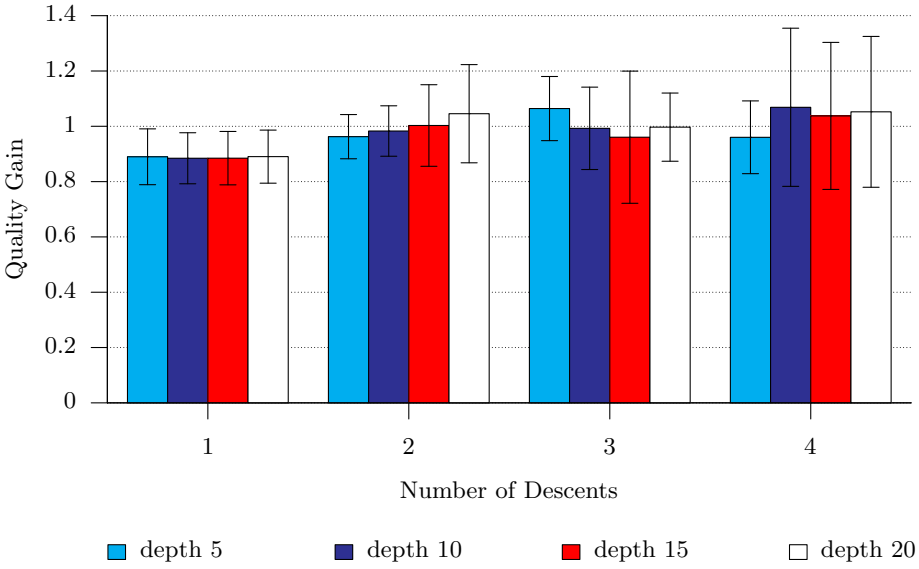

Figure 5. Mean normalised quality gain over the set of input programs with multiple ascent hill-climbing. A graphed value of 1 is equivalent to equal performance with steepest-ascent hill-climbing.

4.2. Simulated Annealing

In order to determine a suitable simulated annealing cooling schedule for use with the QMOOD Understandability function, the set of input programs were automatically refactored using CODe-Imp under all sixteen permutations of *cooling factor* 0.990, 0.9925, 0.9950 or 0.9975 and initial minimum change acceptance probability (p_i) of 1%, 5%, 10% or 25%. The *Markov chain length* parameter was constant at a value of 1.

The starting temperature of a simulated annealing cooling schedule is determined by p_i, which represents the chance that a move in the solution space with a very large quality drop will be accepted in the initial stages of the SA search. Normally, a p_i value of approximately 80% is used in simulated annealing applications. However, this is based on the assumption that the starting solution is of extremely low quality – a randomly-generated solution to the traveling salesman problem, for example. When we consider the size of the space of all possible designs for a given object-oriented program, including those with any number of featureless classes, those with a class for every method, and so on, it seems likely that any program

[p]

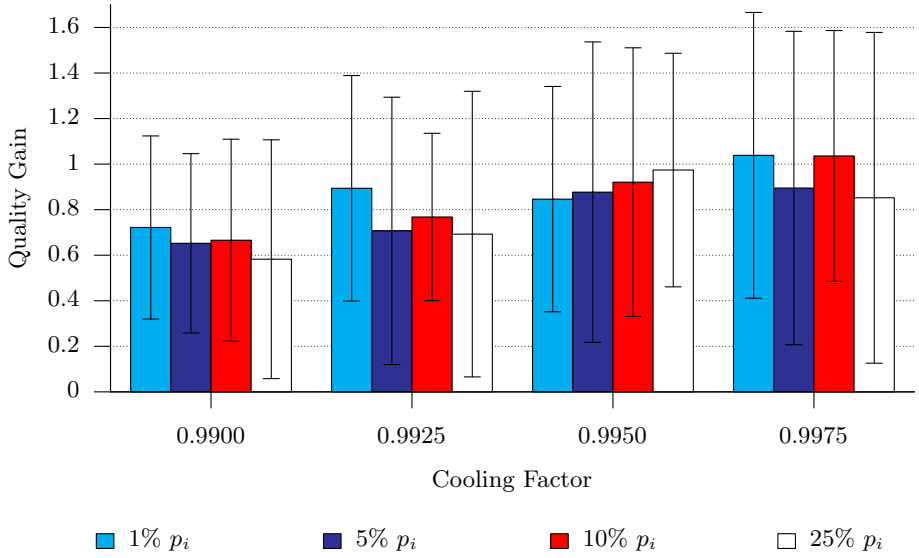

Figure 6. Mean normalised quality gain over the set of input programs with simulated annealing. A graphed value of 1 is equivalent to equal performance with steepest-ascent hill-climbing.

designed with some thought to quality is in fact in the top few percent of designs, even if it has suffered design erosion. Because of this uncertainty, p_i was used as an independent variable in this study, while Markov chain length was fixed at a value of one.

Figure 6 shows the mean normalised quality gain over the set of input programs with sixteen different simulated annealing cooling schedules. Three replications were performed for each cooling schedule, for each input. The mean quality gain value for each input/schedule pair was taken as a single n, so variance seen in figure 6 is due solely to the differing results between input programs. Results were normalised against the constant quality gain obtained from steepest ascent hill-climbing; a normalised quality gain of one for a cooling schedule corresponds to equal performance with HCS over the set of input programs.

Analysis of these data using Kruskal-Wallis tests revealed no statistically significant variation in median quality gain for either cooling factor or p_i parameters. No significant difference was observed between the performance of SA and HCS for any cooling schedule, using Wilcoxon signed rank tests.

The most striking aspect of these results is the extremely large variation in mean quality gain for different input programs. This is present for all cooling schedules examined, but is particularly large for the higher cooling factor values – for a cooling factor of 0.9975 results ranged from approximately half the quality gain of HCS, in the case of input E, to approximately double the quality gain of HCS, in the case of input B. This variation has two important consequences: firstly, it hinders analysis of the relationship between mean quality gain and cooling factor, which was found to be significant for three of the five input programs: statistical analysis using 2-way ANOVA revealed a significant correlation ($p = 0.0476$) for input A, an extremely significant correlation ($p < 0.0001$) for input B and a very significant correlation ($p = 0.0022$) for input E between these variables. Secondly, the large variation in quality gain for each cooling schedule over the set of input programs means that none of these cooling schedules can be recommended for use in search-based refactoring under the QMOOD Understandability function for an arbitrary input. As a result, use of SA in a search-based refactoring tool would likely require experimentation with different cooling schedules for each input in order to be confident that high-quality solutions were obtained, thus increasing the computational cost of applying the tool.

4.3. Genetic Algorithm

In order to determine suitable parameters for a genetic algorithm in search-based refactoring under the QMOOD Understandability function, the set of input programs were automatically refactored under all nine permutations of *mutation proportion* 0.8, 0.5 or 0.2 and *mating chance* 0.8, 0.5 or 0.2. Population size was fixed at ten, and number of generations was limited to one hundred. Although these are low values for these parameters, the long run-time of the GA implementation made them suitable for comparison with the other search techniques employed, as will be discussed in section 4.4.

Figure 7 shows the mean normalised quality gain over the set of input programs with nine different genetic algorithm configurations. Three replications were performed for each configuration, for each input. The mean quality gain value for each input/configuration pair was taken as a single n, so variance seen in figure 7 is due solely to the differing results between input programs. Results were normalised against the constant quality gain obtained from steepest ascent hill-climbing; a normalised quality gain of one for a HCM configuration corresponds to equal performance with HCS over the set of input programs.

Analysis of these data using Kruskal-Wallis tests and Dunn's multiple comparison post-tests revealed a statistically significant variation ($p = 0.002$) in median quality gain for the mutation proportion parameter, where values of both 0.5 and 0.8 performed significantly better than a value of 0.2. The variation in median quality gain was not found to be significant in the case of the mate chance parameter.

Because quality gain varied depending on mutation proportion values, and mean gains are greatest for mutation proportion of 0.8, high values can be recommended for this parameter. Mate chance did not significantly affect results in this study, so less computationally expensive values of approximately 0.2 can be recommended for this parameter. However, for all configurations GA was found to produce significantly smaller quality gains than HCS, using Wilcoxon signed rank tests. It should be noted that in many cases the application of a single

[p]

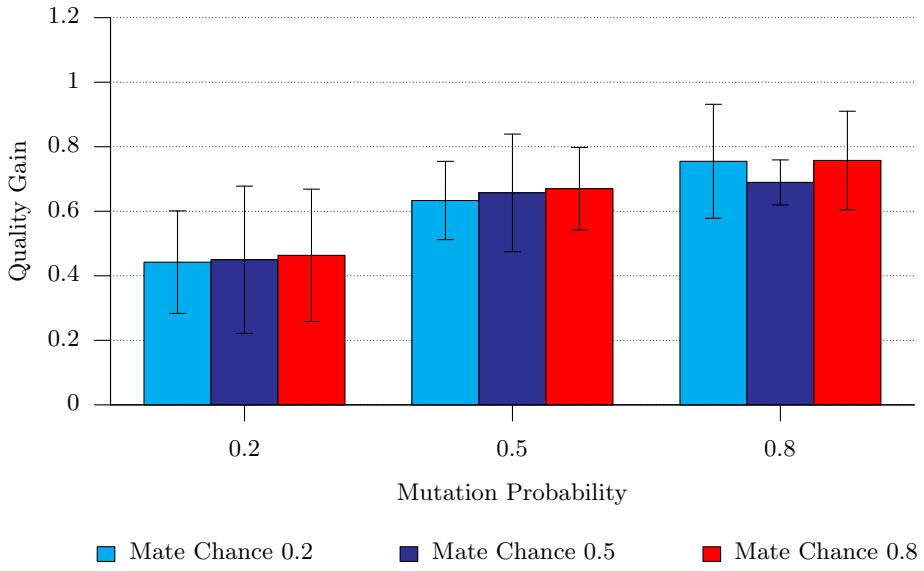

Figure 7. Mean normalised quality gain over the set of input programs with a genetic algorithm. A graphed value of 1 is equivalent to equal performance with steepest-ascent hill-climbing.

refactoring did not affect the QMOOD Understandability function value, so the mutation operator employed did not always produce a change in solution quality. This is likely a factor in the observed ineffectiveness of low mutation probabilities in this study.

4.4. Comparison of Searches

Figure 8 shows the mean quality increase for each search technique for the entire set of input programs. For each program/search pair the highest mean quality gain for any set of parameters was taken as the performance of that search technique for that program. These values were then normalised against the performance of steepest ascent hill-climbing (HCS) for each input program, before mean quality gain over the set of input programs was calculated. Statistical analyses using the (non-parametric) Wilcoxon matched pairs test revealed no significant difference in median quality gain between any of the four search techniques. However, two other criteria are pertinent in comparing their performance.

[p]

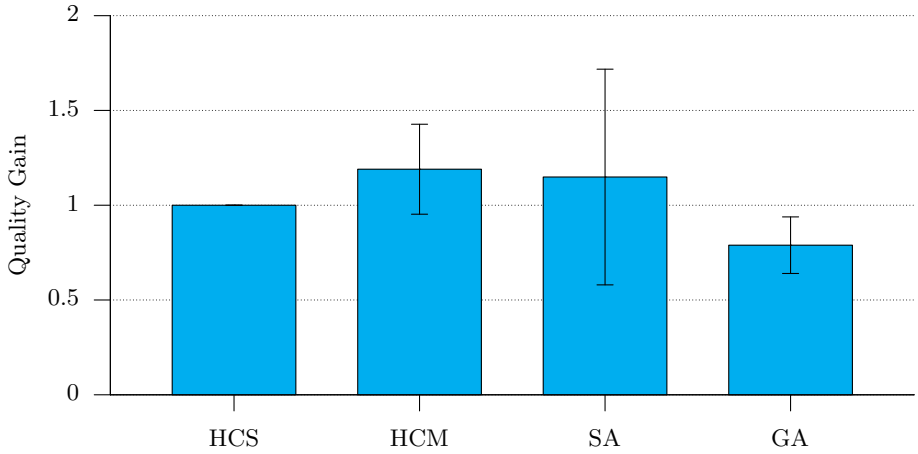

Figure 8. Mean normalised quality gain over the set of input programs for each search technique in most effective configuration.

Firstly, a search technique upon which a search-based refactoring tool is based must be capable of producing good results for any input, as the user will not wish to run the tool several times with different search techniques in order to obtain high–quality results. It is therefore important that quality gain is consistently high across the set of possible input programs. As can be seen from figure 8, the standard deviation of normalised quality gain is largest for simulated annealing, smaller for multiple ascent hill-climbing and genetic algorithm searches and, of course, zero for steepest-ascent hill climbing. Because the performance of SA varied greatly in this study, depending on the characteristics of the input program, the indications are that a search-based refactoring tool should not rely on this search technique alone.

Secondly, the computational cost of performing the various searches is of course a factor in choosing between them. Figures 9 and 10 show the maximum values for mean normalised quality gain for each input with each search technique, and the run-times required to obtain them, respectively. It can be seen that:

- GA was the slowest search by a wide margin for most inputs, and performed relatively poorly in terms of quality gain where it was not.
- The run-time of effective configurations of SA varied greatly, but a long run-time did not necessarily equate to a relatively large quality gain.

[p]

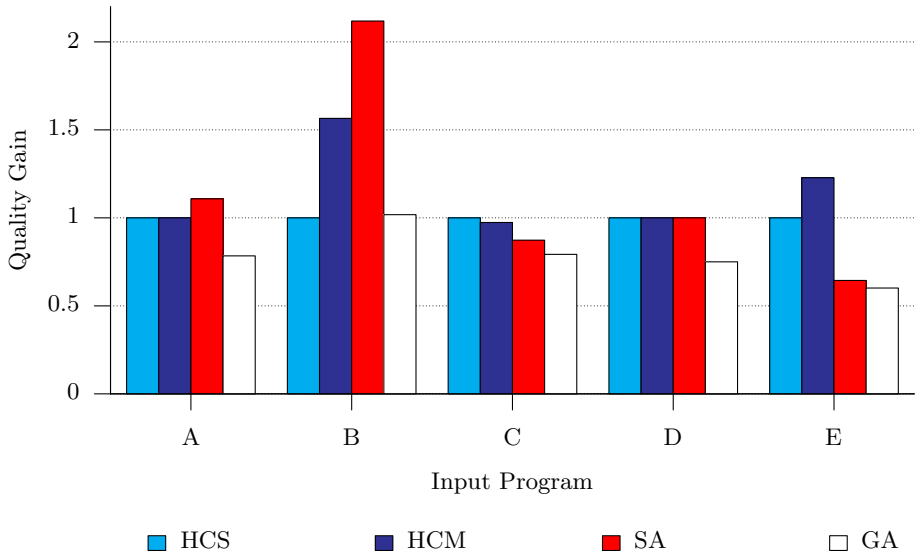

Figure 9. Maximum values for mean normalised quality gain for each input, with each search technique.

- The run-time of HCM varied considerably, but where it was long a large quality gain was also found.

Because run-times may reflect properties of a search technique implementation rather than the inherent efficiency of the technique, we also examine the number of evaluation function calls required. Figure 11 shows the mean number of solutions examined in obtaining the maximum values for mean normalised quality gain that are shown in figure 9. It can be seen that, in contrast to run-times, HCS examines the largest number of solutions while GA examines the smallest. GA is extremely inefficient in the number of solutions it examines relative to its total runtime – partly a result of the complexity of the solution representation in this domain, which necessitates more intense computation than the binary string representation used in other GA applications. SA also examines few solutions relative to its run-time, when compared to hill-climbing searches. In this case, however, the difference is due to the nature of the search algorithm. In hill-climbing searches it is likely that many, if not all, of a solution's neighbours will be assessed for quality before a refactoring is accepted. Hence, each time the Java Program Model (JPM) is queried for possible refactorings, many evaluation function

[p]

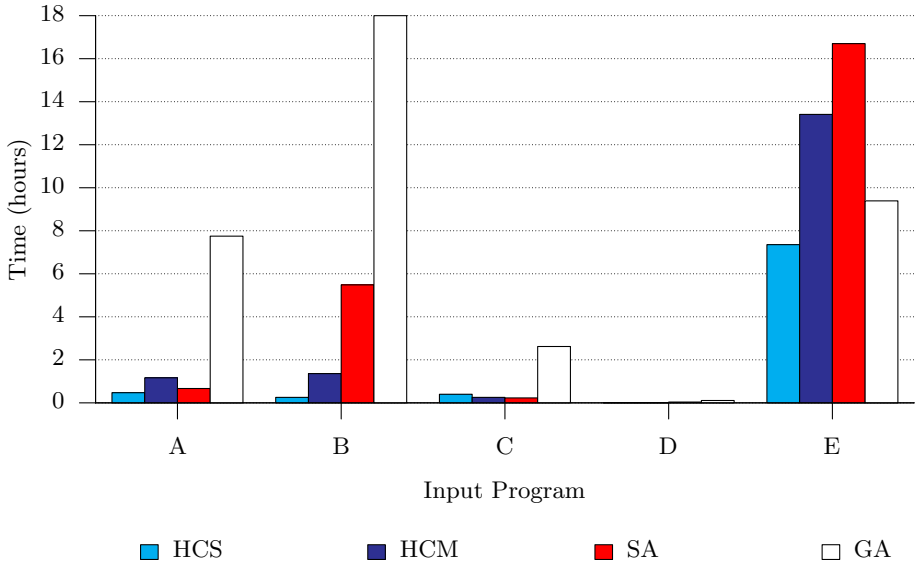

Figure 10. Run-time in obtaining maximum values for mean normalised quality gain for each input, with each search technique.

calls are made. Because the building of the JPM and analysis of what refactorings are possible is computationally expensive, this approach results in more solutions examined for a given runtime.

In summary, simulated annealing has several disadvantages; it is hard to recommend a cooling schedule that will generally be effective, results varied considerably across input programs and the search is quite slow. No significant advantage in terms of quality gain was observed that would make up for these shortcomings over the set of input program, but it should be noted that SA produced by far the greatest quality gain for one input – a fact that indicates that SA can be very effective when configured well for a particular search-based refactoring problem. The genetic algorithm has the advantage that it is easy to establish a set of parameters that work well in general, but the disadvantages that it is costly to run and, for several input programs, did not match the quality gains obtained from the much simpler steepest-ascent hill-climbing search technique in this study. Again, no significant advantage in terms of quality gain was observed that would make up for these shortcomings. Multiple-ascent hill climbing was considered the most efficient search technique in this study; it produced high-

[p]

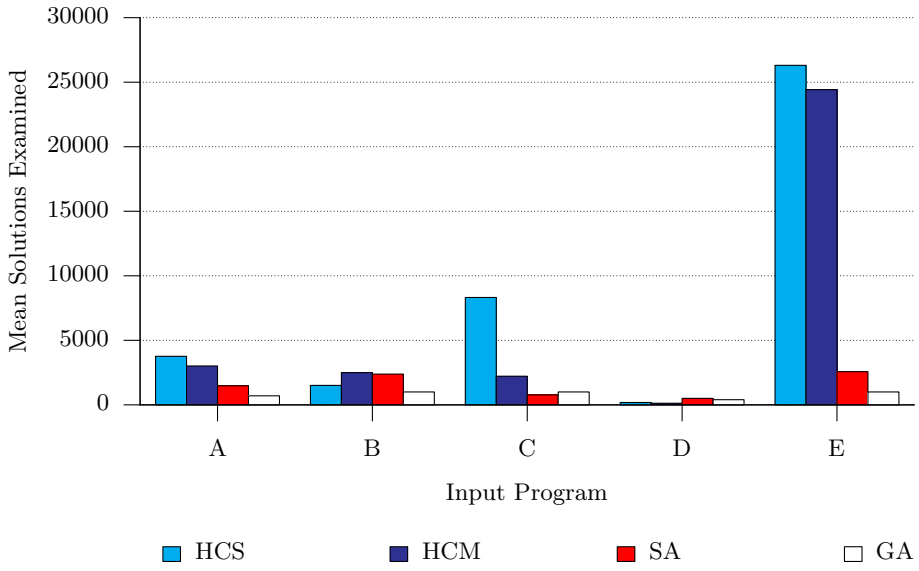

Figure 11. Mean number of evaluation function calls in obtaining maximum values for mean normalised quality gain for each input, with each search technique.

quality results across all the input programs, is relatively easy to recommend parameters for, and runs more quickly than any of the other techniques examined. Steepest ascent hill-climbing produced surprisingly high quality solutions, suggesting that the search space is less complex than might be expected, but is slow when we consider its known inability to escape local optima.

5. Conclusions & Future Work

Based on the results reported above, we conclude that multiple-ascent hill climbing is the most suitable search technique for use in search-based refactoring, at this time. However, the set of input programs used had certain characteristics that can limit the generality of conclusions that can be drawn from these results. Firstly, four of the five programs were open-source, so we can expect that they were created and maintained by enthusiastic programmers working without hard deadlines [30]. As a result, it is likely that these programs have suffered less design erosion

than might be present in commercial programs, and therefore less design quality improvement may be possible. The effect of this factor on search-based refactoring is an area for future study.

A conclusion that is reenforced by the relatively small size of the input programs used in this study is the need for a less memory-intensive solution representation for the genetic algorithm. The current CODe-Imp solution representation is too large for a population of them to be held simultaneously in memory, so the population is implemented as a representation of the original input program (the prototype) and a population of genotypes which store the sequence of refactorings performed. Whenever a mating or mutation operation must be performed on an individual, the phenotype must first be recreated by applying the sequence of refactorings to a copy of the prototype. This GA therefore examines very few solutions relative to its total runtime. Future work should include the development of a more lightweight solution representation that would allow the efficiency of GA search to be increased.

Another characteristic of the set of input programs that does not reflect industry norms is size. The input programs used here have an average size of 57 classes and approximately 4k source lines of code, which would represent a very small program in industry. Multiple ascent hill-climbing will partially adapt to the size of input program due to the nature of the algorithm; although the effectiveness of different search configurations may vary, if an arbitrarily large number of quality-increasing refactorings are possible, a large number of refactorings will be carried out. It is therefore reasonable to assert that HCM will be less sensitive to inappropriate search configurations than simulated annealing, where the number of refactorings that will be (tentatively) performed is predetermined by the cooling schedule. Much larger studies than it has been possible to perform here will be required to determine whether suitable SA cooling schedules can be chosen based on input program characteristics such as size and number of possible refactorings. We therefore conclude that HCM, at this time, should be considered the search technique of choice for search-based refactoring, unless a great amount of resources are available with which to calibrate an SA search for each input.

REFERENCES

1. Anthony J. Bagnall, Victor J. Rayward-Smith, and I. M. Whittley. The next release problem. *Information & Software Technology*, 43(14):883–890, 2001.
2. Jagdish Bansiya and Carl G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Trans. Software Eng.*, 28(1):4–17, 2002.
3. Colin J. Burgess and Martin Lefley. Can genetic programming improve software effort estimation? A comparative evaluation. *Information & Software Technology*, 43(14):863–873, 2001.
4. S. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20:476–493, June 1994.
5. John A. Clark, José J. Dolado, Mark Harman, Robert M. Hierons, B. Jones, M. Lumkin, Brian S. Mitchell, Spiros Mancoridis, K. Rees, Marc Roper, and Martin J. Shepperd. Formulating software engineering as a search problem. *IEE Proceedings - Software*, 150(3):161–175, 2003.
6. José Javier Dolado. A validation of the component-based method for software size estimation. *IEEE Trans. Software Eng.*, 26(10):1006–1021, 2000.
7. José Javier Dolado. On the problem of the software cost function. *Information & Software Technology*, 43(1):61–72, 2001.
8. D. Doval, S. Mancoridis, and B.S. Mitchell. Automatic clustering of software systems using a genetic algorithm. In *International Conference on Software Tools and Engineering Practice (STEP'99)*, 1999.

9. Martin Fowler. *Refactoring: improving the design of existing code.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
10. Mark Harman. The current state and future of search based software engineering. In *Future of Software Engineering, 2007. (FOSE '07)*, May 2007.
11. Mark Harman and John A. Clark. Metrics are fitness functions too. In *IEEE METRICS*, pages 58–69, 2004.
12. Mark Harman, Robert M. Hierons, and Mark Proctor. A new representation and crossover operator for search-based optimization of software modularization. In *GECCO*, pages 1351–1358, 2002.
13. Mark Harman and Laurence Tratt. Pareto optimal search based refactoring at the design level. In *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1106–1113, New York, NY, USA, 2007. ACM Press.
14. Mark Harman and Joachim Wegener. Getting results from search-based approaches to software engineering. In *ICSE*, pages 728–729, 2004.
15. Scott Kirkpatrick, D. Gelatt Jr., and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.
16. Kiran Lakhotia, Mark Harman, and Phil McMinn. A multi-objective approach to search-based test data generation. In *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1098–1105, New York, NY, USA, 2007. ACM Press.
17. Kiarash Mahdavi, Mark Harman, and Robert M. Hierons. A multiple hill climbing approach to software module clustering. In *ICSM*, pages 315–324, 2003.
18. Spiros Mancoridis, Brian S. Mitchell, Yih-Farn Chen, and Emden R. Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *ICSM*, pages 50–, 1999.
19. Spiros Mancoridis, Brian S. Mitchell, C. Rorres, Yih-Farn Chen, and Emden R. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *IWPC*, pages 45–, 1998.
20. Christoph C. Michael, Gary McGraw, and Michael Schatz. Generating software test data by evolution. *IEEE Trans. Software Eng.*, 27(12):1085–1110, 2001.
21. Brian S. Mitchell. *A Heuristic Search Approach to Solving the Software Clustering Problem.* PhD thesis, Drexel University Philadelphia, USA, 2002.
22. Brian S. Mitchell and Spiros Mancoridis. Using heuristic search techniques to extract design abstractions from source code. In *GECCO*, pages 1375–1382, 2002.
23. Brian S. Mitchell, Martin Raverso, and Spiros Mancoridis. An architecture for distributing the computation of software clustering algorithms. In *WICSA*, pages 181–190, 2001.
24. Mark O'Keeffe. *Search-Based Refactoring for Software Maintenance.* PhD thesis, School of Computer Science and Informatics, University College Dublin, October 2007.
25. Mark O'Keeffe and Mel Ó Cinnéide. A stochastic approach to automated design improvement. In James F. Power and John T. Waldron, editors, *Proceedings of the 2nd International Conference on the Principles and Practice of Programming in Java*, pages 59–62. ACM SIGAPP, Computer Science Press, Trinity College Dublin, Ireland., June 2003.
26. Mark O'Keeffe and Mel Ó Cinnéide. Search-based software maintenance. In *Proceedings of the 10th European Conference on Software Maintenance and Reengineering (CSMR 2006)*, pages 249– 260, 2006.
27. Mark O'Keeffe and Mel Ó Cinnéide. Getting the most from search-based refactoring. In *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1114–1120, New York, NY, USA, 2007. ACM Press.
28. Jens Uwe Pipka. Refactoring in a "test first"-world. In *Proceedings of XP2002*, 2002.
29. Olaf Seng, Johannes Stammel, and David Burkhart. Search-based determination of refactorings for improving the class structure of object-oriented systems. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1909–1916, New York, NY, USA, 2006. ACM Press.
30. Ioannis Stamelos, Lefteris Angelis, Apostolos Oikonomou, and Georgios L. Bleris. Code quality analysis in open source software development. *Inf. Syst. J.*, 12(1):43–60, 2002.
31. Joachim Wegener, André Baresel, and Harmen Sthamer. Evolutionary test environment for automatic structural testing. *Information & Software Technology*, 43(14):841–854, 2001.
32. K P Williams. *Evolutionary algorithms for automatic parallelization.* PhD thesis, University of Reading, UK, Department of Computer Science, September 1998.
33. Yuanyuan Zhang, Mark Harman, and S. Afshin Mansouri. The multi-objective next release problem. In *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1129–1137, New York, NY, USA, 2007. ACM Press.