Specialising Software for Different Downstream Applications Using Genetic Improvement and Code Transplantation

Justyna Petke, Mark Harman, William B. Langdon, and Westley Weimer

Abstract—Genetic improvement uses automated search to find improved versions of existing software. Genetic improvement has previously been concerned with improving a system with respect to all possible usage scenarios. In this paper, we show how genetic improvement can also be used to achieve specialisation to a specific set of usage scenarios. We use genetic improvement to evolve faster versions of a C++ program, a Boolean satisfiability solver called MiniSAT, specialising it for three different applications, each with their own characteristics. Our specialised solvers achieve between 4 and 36 percent execution time improvement, which is commensurate with efficiency gains achievable using human expert optimisation for the general solver. We also use genetic improvement to evolve faster versions of an image processing tool called ImageMagick, utilising code from GraphicsMagick, another image processing tool which was forked from it. We specialise the format conversion functionality to greyscale images and colour images only. Our specialised versions achieve up to 3 percent execution time improvement.

Index Terms—Genetic improvement, GI, code transplants, code specialisation, SAT, ImageMagick, GraphicsMagick

1 Introduction

Genetic improvement (GI) [1], [2], [3], [4], [5], [6], [7] uses automated search to find improved versions of existing software. We report on GI-based specialisation applied to MiniSAT [8],[1] a popular Boolean satisfiability (SAT) solver, and to ImageMagick,[2] an open-source image processing software.

MiniSAT is an open-source C++ program. It implements the core technologies of modern SAT solving, including unit propagation, conflict-driven clause learning and watched literals [10]. We chose SAT solving as our target, because of its widespread applicability in software engineering. More specifically, we chose the SAT solving system MiniSAT because it has been iteratively improved over many years by expert human programmers. They have addressed the demand for more efficient SAT solvers and also responded to repeated calls for competition entries to the MiniSAT-hack track of SAT competitions [11]. As such, the SAT solver we seek to improve by specialisation is already highly optimised by expert programmers, and therefore denotes a significant challenge for any further automated improvement.

We use the version of the solver from the first MiniSAT-hack track competition, MiniSAT2-070721,[3] as our host system to be improved by GI with transplantation. Furthermore, this competition, in which humans provide modifications to a baseline MiniSAT solver, provides a natural baseline for evaluation. It also provides a source of candidate 'genetic material' (code fragments that can be transplanted), which we call the *code bank*.

ImageMagick is an open source C program that has been around for over 25 years. It can be used to create, edit, compose, or convert bitmap images. It can also read and write over 200 image file formats. Millions of website use ImageMagick to process images. Many plugins depend on the ImageMagick library, including, PHP's *imagick*, Ruby's *rmagick* and *paperclip*, and nodejs's *imagemagick*. Another popular image processing software, called GraphicsMagick,[4] was forked from ImageMagick in 2002 and it's still in use today.

We use the 5.5.2 version of the ImageMagick software as our host system to be improved by genetic improvement. Furthermore, we use code from the first version of GraphicsMagick. It was forked from ImageMagick-5.5.2. One of the reason's for the fork was to change coding practices to improve the tool's efficiency[5].

1. This paper is an extension of our previous EuroGP conference paper [9], which received the GECCO'14 Humie silver medal (http://www.sigevo.org/gecco-2014/humies.html).
2. https://www.imagemagick.org/

3. Solver available at: http://minisat.se/MiniSat.html.
4. http://www.graphicsmagick.org/
5. http://marc.info/?l=imagemagick-developer&m=104777007831767&w=2

• J. Petke, M. Harman, and W.B. Langdon are with the University College London, London WC1E 6BT, United Kingdom.
E-mail: {j.petke, mark.harman}@ucl.ac.uk, W.Langdon@cs.ucl.ac.uk.
• W. Weimer is with the University of Virginia, Charlottesville, VA 22903.
E-mail: weimer@cs.virginia.edu.

*Manuscript received 27 Oct. 2015; revised 17 Feb. 2017; accepted 5 Apr. 2017. Date of publication 28 June 2017; date of current version 20 June 2018.
(Corresponding author: Justyna Petke.)
Recommended for acceptance by A. Roychoudhury.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2017.2702606*

0098-5589 © 2017 IEEE. Translations and content mining are permitted for academic research only. Personal use is also permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

Definition. The problem of program specialisation is to construct, automatically, from an original general program, different specialised versions, each targeting a specific subarea of the original program's application domain.

The motivation for automated specialisation comes from the observation that programs might have been constructed to target general-purpose solutions to whole classes of related problems [12], [13], [14]. Subsequently, a more specific version of the program may be required for either a subset of the original problem domain, or a slightly different problem domain than that initially envisaged by the programmer [15], [16], [17], [18]. The multiplicity of different platforms, devices and usage scenarios for software systems poses a challenge due to the sheer number of different specialised versions that may be required [19].

Early work on program specialisation focused on techniques such as partial evaluation and mixed computation, which have a long intellectual heritage dating back to 1970s [20], [21], [22]. In a functional programming language, partial evaluation can be achieved by partial application of a function to a subset of its arguments [12], while in imperative styles of programming language, some of the inputs to the program are fully specified (and thereby become fixed at compile time) while others remain free (to be instantiated at runtime) [15].

Partial evaluation consists of 'hardwiring' the consequences of the fixed arguments/inputs into the program code, thereby specialising the program. Hitherto, all work on partial evaluation has focused on the application of program transformation rules to optimise the specialised program for the application sub-domain of interest [12], [23]. This has the advantage that the resulting specialised program is correct, so long as the original program is correct and all of the transformation steps are correct. However, it means that specialisation can only find programs that result from the sequence of meaning preserving transformations, and the specialisation criterion needs to be specified as a subset of input parameters (or a predicate over these parameters [24]). The software engineer needs to decide which of these parameters capture desired behaviour.

We seek to use GI to target a more general form of specialisation. In our approach, the sub-application domain is captured, not by instantiating particular arguments to a function call, nor by selecting fixed values for input, but by a set of test cases that capture the desired behaviour of the specialised program. This allows us to specialise according to any subset of test cases; our specialisation criterion can therefore refer to both input and output (and the relationships between them).

Using a set of test cases to capture the specialisation criterion also has the advantage that it draws the specialisation problem within reach of genetic programming [25], [26], on which our GI is based. By using genetic programming we are not restricted to the deterministic application of a set of meaning-preserving transformations. Instead, we can use evolution to explore the space of candidate specialised programs within a neighbourhood of the original, defined by our genetic operators.

Unlike traditional genetic programming [25], [26], but in common with more recent work on GI [1], [3], [5], [7], [9], [27], [28], GI-based specialisation seeks, not to construct a program from *scratch* but rather, to improve an *existing* program for a specific application domain.

Multiple specialisations can be performed, thereby yielding different specific versions of a program from a single general program. We take the general SAT solver, MiniSAT, and specialise it for three different, real-world, downstream applications. Our aim is that the three GI-evolved specialised MiniSAT versions should outperform any human-optimised general version of MiniSAT, thereby demonstrating the potential of GI-based program specialisation.

We also use a very popular image processing software, ImageMagick, and specialise it for two downstream applications. In particular, we focus on its conversion from jpg to png format function that was identified to be slow on various internet fora.[6] By allowing GI access to code from GraphicsMagick, software that was forked from ImageMagick-5.5.2, we demonstrate the potential of GI for using code from various software variants in order to specialise it for a particular application.

The goal of our approach to automated specialisation is to reduce the reliance on human software engineers as the sole means by which different specialised versions are constructed. Automated specialisation can recommend interventions (small code changes, in our case one-line changes) in order to optimise for a specialised scenario. The software engineer can then decide which of these automatically recommended interventions to adopt.

If the software engineer has high confidence in the testing process [29], then they may simply trust the specialised version. However, provided the number of interventions is manageable, the programmer may find it reasonable to decide upon whether to accept each one on a case-by-case basis. It is therefore important that an automated specialisation approach does not produce too many interventions, yet can, nevertheless, produce nontrivial performance improvement. These observations motivate some of the questions we seek to answer in our empirical evaluation.

The *primary contribution of this paper* is an empirical evaluation that demonstrates that GI-based specialisation can produce multiple human-competitive specialised software versions specialised for various downstream software engineering application domains. We demonstrate this by specialising the 2009 incarnation of MiniSAT and the 2002 incarnation of ImageMagick.

We extend our previous conference paper [9] by providing further evidence for the effectiveness and efficiency of GI-based specialisation: Whereas the conference paper considered only one downstream application, the present paper extends this to three downstream applications for MiniSAT and reports the results of applying GI to specialise another piece of software, i.e., ImageMagick. Furthermore, while the conference version reported only upon the performance improvements due to GI (a subset of Research Question 1 in this paper), the present paper extends these results, reporting, in detail, on the performance of the GI process itself (RQs 2, 3, 4 and 5).

In summary, the 'delta' over previous (conference) version is as follows: MiniSAT has been specialised for two

6. http://www.imagemagick.org/discourse-server/viewtopic.php?t=27580

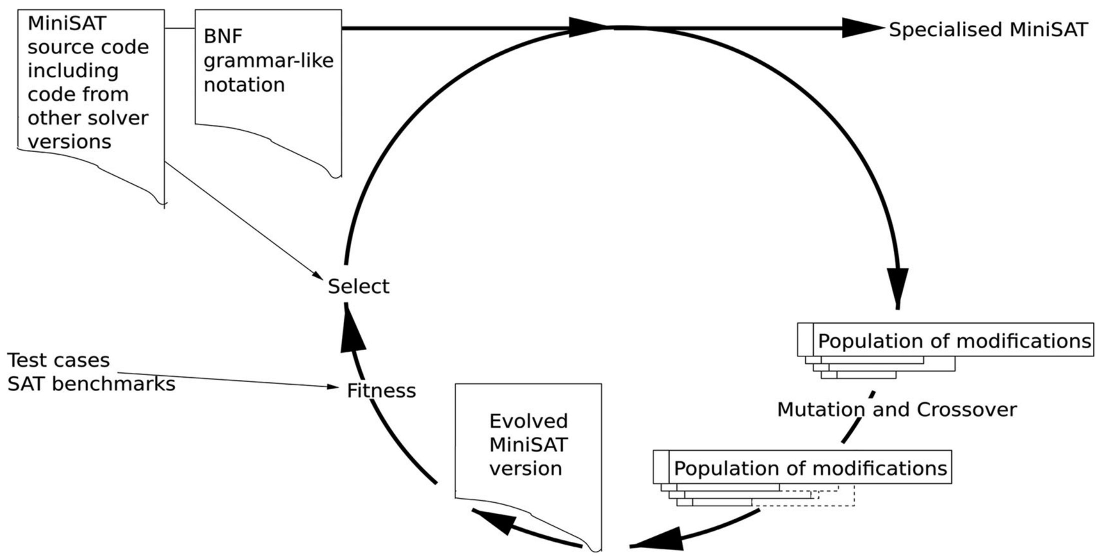
MiniSAT source code including code from other solver versions
BNF grammar-like notation
Specialised MiniSAT
Select
Population of modifications
Test cases SAT benchmarks
Fitness
Evolved MiniSAT version
Mutation and Crossover
Population of modifications

Fig. 1. Overview of the genetic improvement framework used for the MiniSAT solver.

additional application domains (Automated Termination Analysis of Term Re-write Systems and the problem of Ensemble Computation); each experiment has been re-run three times, varying the seed for GI; a detailed analysis of the GI process has been investigated in Section 9; an additional piece of software, called ImageMagick, has been specialised for two downstream applications.

The rest of the paper is organised as follows: Section 2 describes our approach to GI. Section 3 describes our chosen software for improvement, i.e., MiniSAT. Section 4 presents three problem classes to which we apply the proposed approach. Section 5 presents ImageMagick and the software specialisation scenarios. Section 6 describes the research questions posed. Section 7 presents our experiments, the results of which are given in Section 8. In Section 9 we analyse different aspects of our GI framework. In Section 10 we present threats to validity. Section 11 briefly outlines related work and Section 12 concludes.

2 Genetic Improvement with Multi-Donor Transplantation and Specialisation

We describe our approach to genetic improvement (GI), which uses code from multiple authors for transplantation and specialises the genetically improved software for a specific application domain by training GI on a specific set of test cases. We use a population-based genetic programming (GP) approach. Our work extends and adapts the genetic improvement framework introduced by Langdon and Harman [2]. Since we are using a different program, we update the fitness function. We also modify just one C++ or C file which contains the main solving algorithm. However, unlike Langdon and Harman [2], we use multiple donors and focus on specialising the program to improve it for a specific application domain. An overview of the approach for the MiniSAT solver is shown in Fig. 1.

```
<Solver_156> ::= "{\n"
<Solver_157> ::= "Clause* c = Clause_new(ps, false);\n"
<_Solver_158>::= "clauses.push(c);"
<_Solver_159>::= "attachClause(*c);"
<Solver_160> ::= "}\n"
```

Fig. 2. Lines 156-160 from the `Solver.C` MiniSAT program source file represented in our notation inspired by BNF grammars. Lines marked with `_Solver` can be modified.

Program Representation. GI modifies the code (in this case MiniSAT and ImageMagick) at the level of lines of source code. A notation (inspired by BNF grammars) is used to create a template containing all the lines from which new individuals are composed [2]. Such a template is created automatically and ensures that classes, types, functions and data structures are retained. For instance, opening and closing brackets in C++ programs are forced to stay in the same place, but the lines between them can be modified. Moreover, initialisation lines are also left untouched. An extract of a template for MiniSAT is shown in Fig. 2. Header files and comments are not included in our representation. The genome used in our GP is a list of mutations (see below).

Code Transplants. We evolve the host program by transplanting lines of code from other programs [30]. GI can also modify both original and transplanted code. Thus our GP has access to both the *host* program being evolved, as well as the *donor* program(s). We call all the lines of code which GP has access to the *code bank*. The donor code statements are then available for mutations of the *host* instance, but need not be used in the same order. For example, our search may combine the first half of an optimisation from one version of MiniSAT with the second half of an optimisation from another and then specialise the resulting code to problems from a particular application domain. This re-use and improvement of existing developer expertise is critical to the success of our technique.

Mutation Operator. A new version of a program (i.e., a new *individual*) is created by making multiple changes to the original program. Each such *mutation* or, in other words, update is either a DELETE, REPLACE or COPY operation. The changes are made at the level of lines of source code (with a special case for conditional and loop statements), which are picked at random from the code bank. A DELETE operation simply deletes a line of code, a REPLACE operation replaces a line of code with another line of code from the code bank

```
<_Solver_159>
              # Delete line 159
<for3_Solver_533><for3_Solver_772>
              # Replace the 3rd part of the 'for'
              # loop (i.e., loop variable increment)
              # in line 533 with the 3rd part of
              # the 'for' loop in line 772
<_Solver_806>+<_Solver_949>
              # Add line 949 in front of line 806
```

Fig. 3. Examples of the three types of mutations.

and COPY operation inserts a line of code from the code bank into the program. In the case of conditional and loop statements, we focus on and modify their predicate expressions.[7] For instance, the second part of a FOR loop (e.g., `i<0`) can only be replaced with the second part of another FOR loop (e.g., `i<10`) and any IF condition can be replaced with any other IF condition. Examples of the three mutation types are shown in Fig. 3.

Initial Population. The initial population is generated at random. Each individual in the initial population consists of a single mutation, i.e., either a DELETE, COPY or REPLACE operation, selected at random. Three examples of such single-mutation individuals are presented in Fig. 3.

Crossover Operator. We represent each individual as a list of mutations, which we call the *edit list*. This representation allows our technique to apply to programs of significant size [31], since we do not keep the whole of each version of the program in memory-just a list of changes. When creating individuals for the next generation, a *crossover* operation simply concatenates two individuals from the current population by appending one list to another. The first parent is chosen based on its fitness value while the other is chosen uniformly among those individuals from the breeding population, as in previous work [2].

Fitness Function. We evaluate the fitness of an individual in terms of a combination of functional properties (those related to software correctness) and non-functional properties (those related to performance, quality of service, etc.) by observing its performance on SAT instances. Before the GP starts, the training set of SAT instances is divided into five groups by difficulty, which we measure in required solving time and instance satisfiability.[8] In each generation one test case is sampled uniformly from each group (or 'bin' following terminology in [2]) and all individuals are run on the five selected test cases, following previous work. This sampling helps to avoid over-fitting. To evaluate an individual, the corresponding list of changes is applied to the original program and the resulting source code is compiled, producing a new SAT solver that can then be executed (individuals that fail to compile are never selected to be parents of the next generation).

To guide the GP search toward a more efficient version of the program, our fitness function takes into account both solution quality (in our case whether an instance is satisfiable or not) and program speed. We note that it will vary depending on the software application to be improved. For internal fitness calculations, efficiency is measured in terms of lines of code executed based on simple counter-based program instrumentation. The use of line counts (instead of CPU or wall-clock times) avoids environmental bias and provides a deterministic fitness signal. Therefore, we can use test cases that require a few seconds to compute. For the final presentation of our empirical results, timing measurements in seconds are also presented (see Section 8).

Selection. The GP process is run for a fixed number of generations with a fixed population size. After the fitness of each of the individuals is calculated, the fittest half of the population is chosen, filtered to include only those individuals that exceed a threshold fitness value. We focus on exploiting high-quality solutions, and thus our fitness threshold is set to select those individuals that either (1) return the correct answer in all cases, or (2) return the correct answer in all but one case and produce the correct answers at least twice as quickly as the original solver as measured in terms of the number of lines of code executed.

Next, a set of offspring individuals is created using crossover on those selected from the current population. Also a new mutation is added to each of the parent individuals selected to create offspring. Both crossover and mutation are applied with 50 percent probability. If mutation is chosen, one of the three operations (i.e., REPLACE, COPY and DELETE) is selected with equal probability. If mutation and crossover do not create a sufficient number of individuals for the next generation, new individuals are created consisting of one mutation (selected randomly). Finally, the fitness of the newly-created individuals is calculated, as described above, and the process continues until the generation limit is reached.

Filtering. We have observed that many program optimisations are independent and synergistic. As a result, we propose a final step that combines all mutations from the fittest individuals evolved and retains all synergistic edits. Exploring all subsets of edits is infeasible. Our prototype implementation uses a greedy algorithm. Each mutation from the best individuals from our experiments is considered separately. We apply each operation to the original program and evaluate its fitness. Next, we order the mutations by their fitness value[9] and iteratively consider these, adding only those edits that increase fitness. Other efficient techniques, such as constructing a 1-minimal subset of edits [32], are possible.

3 SAT SOLVING IN SOFTWARE ENGINEERING

The Boolean satisfiability problem (SAT) is the problem of deciding whether there is a variable assignment that satisfies a propositional formula. An example formula in conjunctive normal form (CNF) is: $(x \vee y) \wedge \neg z$, where x, y and z are Boolean variables; this formula is satisfiable, e.g., by the following assignment: $x = 1$, $y = 0$ and $z = 0$, while $z = 1$ makes the formula unsatisfiable. Many problems involving constraints can be encoded into CNF efficiently [33], thus allowing SAT solvers to be used on a wide range of problems.

7. In the case of a DELETE operation we replace the predicate expression with '0'.

8. The first group of test cases contains fast satisfiable SAT instances; the second group contains fast unsatisfiable instances; the third group contains satisfiable instances that require more time to solve; the fourth group contains unsatisfiable instances that require more time to solve; the fifth group contains a mixture of SAT instances requiring the most amount of time to solve.

9. Note that since each individual is represented by a list of edits (or mutations) and at the filtering stage we consider one mutation in turn, we use the word 'mutation' and 'individual' interchangeably.

Due to the developments in the early 2000s SAT solvers have become extremely efficient [34] and hence new application domains emerged, including problems in software engineering (SE). It would be infeasible to mention all the work in SE that uses SAT solvers, thus we will only mention a few problem domains. SAT solvers have been widely used in software and hardware verification. They improved the scalability of symbolic model checking, an important technique in verification [35], [36], by acting as backend solvers in the state-of-the-art model checkers. SAT solvers have also been used for finding optimal solutions for test suite minimisation [37] as well as optimal combinatorial interaction test suites [38], [39], [40] by translating the whole problem instance into SAT. Other applications involve test suite prioritisation [41] and software product line engineering [42]. Moreover, work on a SAT-based constraint solver won the ACM SIGSOFT Distinguished Paper award at the International Conference on Software Engineering 2015 [43].

MiniSAT is a well-known open-source C++ solver for SAT. It implements the core technologies of modern SAT solving, including: unit propagation, conflict-driven clause learning and watched literals [10]. The solver has been widely adopted due to its efficiency, small size and availability of ample documentation. It is used as a backend solver in several other tools, including Satisfiability Modulo Theory (SMT) solvers, constraint solvers (for solving Constraint Satisfaction Problems-CSP), Answer Set Programming (ASP) systems and solvers for deciding Quantified Boolean Formulae (QBF). MiniSAT has also served as a reference solver in SAT competitions.

In the last few years progress in SAT solving technologies involved only minor changes to the solvers' code. Thus in 2009 a new track has been introduced into the SAT competition, called MiniSAT-hack track. In order to enter this track one needs to modify the code of MiniSAT. This solver has been improved by many expert human programmers over the years, thus we wanted to see how well an automated approach scales. We used genetic improvement in order to find a more efficient version of the solver. In our experiments we used the version of the solver from the first MiniSAT-hack track competition-MiniSAT2-070721.

4 Problem Classes

SAT solving has a wide range of applications ranging from model checking through planning to automatic test-pattern generation [34], [44]. Moreover, over 1,000 benchmarks are available from SAT competitions.[10] These are divided into *application*, *random* and *crafted* categories.

We focus on three real-world problem domains to which SAT solving has been applied. Moreover, a wide range of benchmarks is available for each of the problem classes chosen. These include easy instances, solvable within seconds, that can be used within our GI framework during fitness evaluation. The three SAT problem classes have also been used by Bruce et al. [45] to optimise MiniSAT for energy consumption.

4.1 Combinatorial Interaction Testing

SAT solving has recently been successfully applied to Combinatorial Interaction Testing (CIT) [38], [39], [40], allowing us to experiment with GI for specialisation to that problem domain. CIT is an approach to software testing that produces tests to expose faults that occur when parameters or configurations to a system are combined [46]. CIT systematically considers all combinations of parameter inputs or configuration options to produce a test suite. However, CIT must also minimise the cost of that test suite. The problem of finding such minimal test suites is NP-hard and has attracted considerable attention [47], [48], [49], [50], [51].

SAT solvers have been applied to CIT problems [38], [39], [40] but the solution requires repeated execution of the solver with trial test suite sizes, making solver execution time a paramount concern. We follow the particular formulation of CIT as a SAT problem due to Banbara et al. [38], since it has been shown to be efficient.

4.2 Automated Termination Analysis

Program termination is one of the most important properties of software. Even though the problem is undecidable in general, there are techniques that can determine if certain programs will terminate automatically. There has been a lot of research in the area of termination analysis of term rewrite systems (TRS) [52]. Many programming languages can be translated into TRSs, thus making tools for termination analysis of TRSs very popular. From 2006 SAT solvers have been used to automate certain TRS termination techniques [53], and now they are a key technique in the field [54].

One of the systems for automated termination proofs of term rewrite systems is the Automated Program Verification Environment (AProVE).[11] We use SAT benchmarks obtained using this system that were also submitted to SAT competitions in 2007, 2009 and 2011.[12]

4.3 Ensemble Computation

SAT solving is used outside software engineering as well. Thus we include another application in order to also investigate wider applications of genetic improvement for SAT solving beyond software engineering.

SAT representation is a natural fit for modelling problems relating to logical circuits, for example, testing circuit equivalence. In real-world circuits a key issue is minimising the number of elementary computations for a given task. This problem generalises to finding the smallest Boolean circuit that computes multiple Boolean functions simultaneously. The Ensemble Computation problem is to decide whether a certain number of arithmetic gates suffices to evaluate all the computations on the required subsets of input variables [55], [56].

Recently a SAT encoding has been introduced to model the problem of deciding whether a given ensemble has a circuit of given size [56]. A generator of such instances is provided at: http://www.cs.helsinki.fi/u/jazkorho/sat2012/. The website also provides a set of challenge instances that are yet to be solved.

10. See http://www.satcompetition.org.

11. See:http://aprove.informatik.rwth-aachen.de/index.asp?subform =home.html.

12. SAT benchmarks obtained with AProVE that we used are available at: http://www.cs.ucl.ac.uk/staff/C.Fuhs/.

5 Image Processing

Image processing deals with transforming an image to create a new image, for instance, to increase the image quality or reduce it's size. There exist hundreds of different image file formats that offer different trade-offs. Image manipulation is very important, since images can be used for various tasks. These can range from posting holiday images on one's blog to storing medical images, such as MRI images.

ImageMagick, one of the most popular open source software suites for image processing, is able to process over 200 different file formats. It can animate an image sequence, compare mathematically and visually annotate the difference between an image and it's reconstruction. It allows for conversion between the various image formats as well as image resizing, blurring, flipping and other. It can display and report on the characteristics of the input image. It can also create a composition of images by combining several separate images.

GraphicsMagick, forked from ImageMagick at the end of 2002, is a popular alternative to ImageMagick. There are plenty of web articles and blogs comparing both tools. They have similar functionality, yet their efficiency might vary significantly depending on the type of images being processed. We envision that GI could combine the two programs to create a hybrid that would work best for all image conversion tasks. In our experiments we focus on a specific functionality, namely, conversion from jpg to png, that was reported to be slow by users of ImageMagick.

Our chosen version of ImageMagick, i.e., 5.5.2, is an incarnation of software that has been around for over 10 years. Even though the GraphicsMagick fork occurred partly to improve software's efficiency, the core functions have largely remained the same between the two projects at the time of the fork. Therefore, this version is already highly-optimised by expert human developers. As mentioned before, we will use GI to improve efficiency of the jpg to png file conversion. We use two sets of images for specialisation: greyscale and colour images.

6 Research Questions

The primary research questions for any technique that seeks to provide automated support to software engineers concern the efficiency and effectiveness of the proposed technique. Therefore, our first research question investigates this for GI-based specialisation.

RQ1, Effectiveness and Efficiency. What is the effectiveness and efficiency of GI-based specialisation?

This question consists of two subquestions, concerned with effectiveness and efficiency. The effectiveness of GI-based specialisation concerns the degree to which it can create improved specialised versions:

RQ1a, Effectiveness. Can genetic improvement find faster specialised software versions than any general version developed and optimised by expert human programmers?

The efficiency of GI-based specialisation is simply the computational cost of the specialisation process:

RQ1b, Efficiency. What is the computation cost of the specialisation process?

Note that a specialised version, once constructed, will be used multiple times. Therefore, the overall approach remains useful, even where efficiency gain for each execution of a specialised version is considerably smaller than the computational cost of producing it. However, the computational cost will, nevertheless, determine the ways in which GI-based specialisation can be used in practice. Given the complexity of the task in hand, it seems unreasonable to expect specialisation to be instantaneous, but it will need to be fast enough to incorporate into a development cycle (for example, taking no longer than an overnight build, in practice).

If our approach to GI-based specialisation proves to be sufficiently efficient and effective to be potentially useful, then this serves as a proof of concept. However, the nature of the specialisation process immediately raises a number of important subsidiary questions, concerning the factors which may influence the quality and performance of GI-based specialisation; questions to which we now turn.

Clearly, the larger the code bank, the larger is the potential search space of possible program mutations. We therefore investigate the relationship between the size of the code bank and the performance of genetic improvement: *RQ2, Code Bank Size.* How well does the genetic improvement approach perform depending on the size of the code bank?

Our GI-based specialisation uses a post-processing filter to find the best individual mutations from all GI runs for each problem class. The outcome of this filtering process is the final set of modifications to be recommended for the original program in order to improve it. This set of modifications needs to be small enough to be practical, if the genetic improvement technique is to be used as a recommendation system (which recommends 'specialisation interventions' to the software engineer). This motivates our next research question:

RQ3, Number of Modifications Required. Does our filtering technique produce the most efficient solver variants when compared with the ones evolved directly by genetic improvement and how many interventions are recommended?

One would expect the specialisation technique to behave differently for each downstream application. If the same intervention is required for each and every application, then the specialisation technique cannot truly be said to be specialising. This motivates our fourth research question:

RQ4a, Specificity. Are the changes produced by GI problem-specific?

RQ4b, Generality. Are the changes produced by GI general efficiency improvements?

Finally, since our approach uses computational search as the primary mechanism for identifying improvements, there are a number of natural questions that arise concerning the computational search strategy. These are addressed in our final set of related research questions:

RQ5a, Fitness Function. What is the impact of the trade-off between efficiency and effectiveness in fitness function on finding a specialised software version using GI?

RQ5b, Comparison to Random Search (*Sanity Check* [57]). How does the chosen search strategy compare with random search?

RQ5c, Genetic Operators. What is the impact of various mutation and crossover operator rates on GI efficacy?

In order to provide answers to the research questions posed we conduct several genetic improvement runs described in the next section.

7 Experimental Setup

We present details of the genetic improvement framework used in our experiments.

Host & Donor Programs. We evolve MiniSAT2-070721, in particular the C++ file containing its main solving algorithm (i.e., the Solver.C file). This version was used as a reference solver in the first MiniSAT-hack competition, organised in 2009. Unless otherwise noted, we use MiniSAT and MiniSAT2-070721 interchangeably. The main solver algorithm involves 478 of the 2,419 lines in MiniSAT. For our experiments we use two donor programs, which altogether provide 104 new lines of source code. The first donor is the winner of the MiniSAT-hack competition from 2009, called "MiniSAT 09z". We refer to this solver as MiniSAT-best09. The second donor program is the "MiniSat2hack" solver, the best performing solver from the competition when run on our CIT and AProVE specific benchmarks.[13] In order to conform with notation in our previous work [9] we refer to this solver as MiniSAT-bestCIT.

We also evolve ImageMagick-5.5.2.[14] We used callgrind[15] and gprof[16] to profile ImageMagick and find the most time consuming part of the code when converting from jpg to png file format. We found that majority of the time is spent in the jpeg.c file, in it's ReadJPEGImage function, hence we target it for specialisation. Unless otherwise stated, we use ImageMagick and ImageMagick-5.5.2 interchangeably. We use GraphicsMagick-1.0[17] as the donor program. This is the first version of the software based on a fork from ImageMagick. We use GraphicsMagick-1.0 and GraphicsMagick interchangeably. ImageMagick's ReadJPEGImage function contains 414 lines of code (of 1,426 in the jpeg.c file). By transplanting code from GraphicsMagick, the genetic improvement process has access to overall 439 lines of code.

Test Cases. Real-world SAT instances from the combinatorial interaction testing area can take hours to run. Thus we evaluate MiniSAT performance on a set of synthetic CIT benchmarks. Using the encoding of Banbara et al. [38], we translated 130 CIT benchmarks into SAT instances.[18] We kept the number of values for each of the parameters the same in every instance. This allows us to verify observed results against public catalogues of best known results [48]. We use about half of these CIT benchmarks in the training set (which is divided into five groups, as discussed in Section 2) and used the rest in the verification set.

We chose 56 real-world SAT benchmarks from the automated termination analysis field (based on runtime), 24 of which are used as our training set. Once again we use execution time to define instance difficulty and divide the training set into five groups, where the second and fourth group contain unsatisfiable instances only, while the first and third contain only satisfiable ones.

Instances for the problem of finding efficient circuits for ensemble computation have been produced using the instance generator provided (see Section 4.3). A subset of benchmarks from the 'smallest' category available on the website is also used. The test set contains altogether 50 test cases, half of which are used by GI.

In each of the three cases we use execution time to define instance difficulty and divide the training set into five groups based on that measure. The largest instances in the training sets contain over 1 million SAT clauses. Nevertheless, MiniSAT is able to produce an answer for each of these within two minutes on the desktop machine used.

We evaluate ImageMagick's performance on five sets of greyscale and colour images. Each set consists of 20 images coming from the following five sources: geometric shapes used in previous work by the authors,[19] face images taken from University of Massachusetts 'Labelled Faces In The wild' dataset,[20] 'Pasadena Houses 2000' dataset used by the Computational Vision Group at California Institute of Technology,[21] images of galaxies posted by the National Optical Astronomy Observatory[22] and a set of personal photographs showing scenes from everyday life.[23]

As in the case of MiniSAT, we divided the images based on their type into 5 groups. In each generation we randomly sample an image from each group to avoid over-fitting. We specialise ImageMagick for two cases: greyscale and colour images. In the first set of experiments with ImageMagick we use 100 randomly selected greyscale images, 20 for each image type (i.e., house, galaxy, face, geometric shape, personal) in the training set. We randomly select another 100 for the test set. We repeat this procedure for the second set of experiments were we use colour images.

Code Transplants. In our experiments the source code of high-level human optimisations targeting a generic benchmark set serve as donor code and are selected and recombined with novel changes to produce a specialised host SAT solver. Adding a donor statement X to the code bank is equivalent, in terms of the search space explored, to adding IF (0) X to the input program in a preprocessing step.

In our previous work [9] we obtained the best results when using the best version of MiniSAT for the CIT domain as the donor. Thus we first evaluate which version of the solver is best for the three problem classes. Next, we use that version of the solver as the donor and run the GI framework. In the second experiment we add all three MiniSAT versions to the code bank in each case. We repeat all GI runs three times.

In the case of ImageMagick, we use code from GraphicsMagick as the donor code. In all experiments code from GraphicsMagick is in the code bank along with the original

13. Both solvers are available from the SAT competitions' website: http://www.satcompetition.org/.
14. Software available at: https://sourceforge.net/projects/imagemagick/files/old-sources/5.x/5.5/
15. http://valgrind.org/docs/manual/cl-manual.html
16. https://sourceware.org/binutils/docs-2.16/gprof/
17. Software available at: http://78.108.103.11/MIRROR/ftp/GraphicsMagick/1.0/
18. Benchmarks as well as the different MiniSAT versions are available by e-mail from Justyna Petke at j.petke@ucl.ac.uk.
19. Available upon request.
20. http://vis-www.cs.umass.edu/lfw/lfw.tgz
21. http://www.vision.caltech.edu/archive.html
22. https://www.noao.edu/image_gallery/galaxies.html
23. Available upon request.

code from ImageMagick. We specialise the conversion from jpg to png by modifying the ReadJPEGImage function.

Software Comparison. SAT instances produce either a 'Satisfiable' or 'Unsatisfiable' result. Therefore, output comparison of the original MiniSAT and the modified one is easily comparable. We note that we also check that the satisfiable answer returned by the evolved solvers is also correct.

As far as ImageMagick is concerned, we first specialise it for the set of greyscale images without allowing for any variation in the quality of the png image produced. We use the Mean-Squared Error metric for our comparison, frequently used for image comparison.[24] The fitness function takes into account values for all the RGB dimensions produced by the metric. We also investigate the trade-off between image quality and efficiency gain. Therefore in the next experiments, we allow for 50 percent difference in each of the red, green and blue values according to the MSE metric. We repeat these experiments for colour images. Each of the four GI runs is repeated three times and the best results are reported.

We compare our evolved software with both the host and donor programs in each of the experiments. We call our evolved software MiniSAT-gp and ImageMagick-gp, respectively. Finally, we refer to the software that results from our post-processing filtering step (see Section 2) as MiniSAT-gp-combined and ImageMagick-combined, respectively.

8 Results

To evaluate the efficacy of our technique, we evolve improved and specialised versions of MiniSAT and ImageMagick and compare them to human-improved SAT solvers and ImageMagick, respectively, in terms of both runtime cost and solution quality.

When specialising MiniSAT we conduct two sets of experiments for each application described in Section 4, varying the donor code bank. When specialising ImageMagick, we vary the image quality threshold in the fitness function for both greyscale and colour images. Each GI run is repeated three times (with different seed for mutation and crossover). The results are reproducible if the same seed, GI framework, population size and number of generations are used. While internal fitness calculations are measured in terms of lines of code executed, all final results are presented in terms of CPU time data based on runs on a Dell OptiPlex 9010 desktop with 8 GB RAM and a single 3.40 GHz Intel Core i7-3770 CPU in the case of MiniSAT and a 1.6 GHZ Lenovo 3000 N200 laptop with an Intel Core 2 Duo processor and 2 GB of RAM in the case of ImageMagick.

The GI framework was run with a population size of 100 for 20 generations. In each generation in the MiniSAT set of experiments the top fitness value was shared by up to 75 individuals. Note that individuals with the highest fitness among all generations might not always be the best, since in each generation a set of test cases is picked *at random* from the training set. Therefore, we used the following strategy to identify the best evolved solver: pick the best solver in each generation based on the *total* number of lines executed; evaluate these evolved solvers on the *whole* training set; select the one that requires the least number of lines to be executed as the best overall evolved solver. As before, we used the lines of code execution measure to determine the best evolved solver to avoid environmental bias.

24. We use ImageMagick's '-metric MSE' and '-colorspace Lab' options.

TABLE 1
Normalised Runtime Comparison of MiniSAT Versions, Specialised for CIT, Based on Averages Over 10 Runs

Solver	Donor	Lines	Time
MiniSAT (original)	—	1.00	1.00
MiniSAT-gp	bestCIT	0.97	0.99
MiniSAT-gp	bestCIT+best09	0.73	0.85
MiniSAT-gp-combined	bestCIT+best09	**0.72**	**0.84**

The "Donor" column indicates the source of the donor code available in the code bank. "Lines" indicates lines of code executed, "Time" indicates CPU time executed. Left column contains the best MiniSAT versions from 3 runs of the GI framework. (Lower is better, all measurements normalized to original MiniSAT).

We note that the best individual in terms of runtime might still be missed. However, we will get deterministic results. The same approach was used to evaluate each mutation (and combinations of mutations) during the filtering stage. In all experiments the compilation rate (using MiniSAT's and ImageMagick's provided Makefile) was high, between 68 and 95 percent. This high compilation rate results from our use of a specialised notation for edits, which prevents syntax errors, which was previously used by Langdon and Harman [2]. An example of which is shown in Fig. 2.

8.1 MiniSAT: Combinatorial Interaction Testing

In our previous work we identified "MiniSat2hack" solver as the best performing solver from the MiniSAT-hack track 2009 competition when run on our CIT-specific benchmarks. In order to conform with notation in our previous work [9] we refer to this solver as MiniSAT-bestCIT.

8.1.1 Transplanting from MiniSAT-bestCIT

In this experiment the code bank contained source code both from the original MiniSAT solver as well as MiniSAT-bestCIT. It was re-run three times varying the random seeds for genetic operators. To pick the best solver from the three GI runs, each of the evolved versions was run 10 times on the whole training set. Average runtimes were used to establish the best solver.

Runtime comparison with the fastest evolved solvers (called MiniSAT-gp each) for all 130 benchmarks used for the CIT problem domain is shown in Table 1. The best evolved version of MiniSAT is, on average, 1 percent faster than the original solver. Given that the best solver in our previous experiment [9] provided 17 percent improvement, this shows the importance of producing repeated runs.

It is unclear what impact certain mutations have on the solving process. However, we identified that 3 out of 6 line deletions simply removed assertions, which are indeed not needed in the solving process. This optimisation is rather trivial. These can be easily removed from the GI process by removing assertions from the code bank, so that GI searches for efficiency gains elsewhere in the code.

TABLE 2
Mutations Occurring in the Best Genetically Improved Solver, Specialised for CIT, from Three GI Runs

solver	mutation	mutated code	changes
MiniSAT-gp			
	DELETE	IF statement condition	2
	REPLACE	IF statement condition	2
	DELETE	line of code	6
	COPY	line of code	1
	DELETE	FOR loop condition	4
	REPLACE	FOR loop condition	11
total			26

(Donor: MiniSAT-bestCIT.)

Moreover, 6 out of 11 loop condition replacements have not introduced any important changes to the code, for example, i++ was substituted with i++.

The performance of our evolved version and the human-written version are not much different. Changes made by the GI process are shown in Table 2.

8.1.2 Transplanting from MiniSAT-bestCIT & MiniSAT-best09

In the second experiment for the CIT domain we added the version of MiniSAT that won the 2009 MiniSAT-hack competition to the code bank. We repeated the GI run three times and used the same evaluation criteria as in the previous experiment.

In contrast to previous work [9] this led to a faster version of MiniSAT than when only MiniSAT-bestCIT donor was used. The evolved version improved MiniSAT runtime by 15 percent as shown in Table 1. Details of the changes in this best evolved solver are presented in Table 3. The IF statement condition replacement triggered 95 percent of the code from MiniSAT-bestCIT donor, modifying the conflict analysis stage of the SAT solving process. The same change was discovered by GI run in our previous work [9].

8.1.3 Combining Results

Many mutations in the best evolved individuals are independent, that is, different one-line mutations, as shown in Fig. 3, occur in various software versions. Thus our approach based on filtering holds out the promise of combining the best parts of all variants discovered.

In the previous experiment the GI identified a 'good change': a one-line modification that allowed 95 percent of the code of MiniSAT-bestCIT donor to be executed. Even though the GP process produced individuals containing such a change, other mutations within all such individuals caused slower runtime or compilation errors.

TABLE 3
Mutations Occurring in the Best Genetically Improved Solver, Specialised for CIT, from Three GI Runs

solver	mutation	mutated code	changes
MiniSAT-gp			
	REPLACE	IF statement condition	1
total			1

(Donor: MiniSAT-bestCIT & MiniSAT-best09.)

TABLE 4
Mutations Occurring in the Combination of the Fastest Genetically Improved Solvers, Specialised for CIT, Also Presented in Fig. 4

mutation	mutated code	number of changes
DELETE	IF statement condition	2
DELETE	line of code	4
REPLACE	IF statement condition	1
total		7

We use our filtering technique described in Section 2 to combine the best mutations. We started with the individual composed of one mutation with the best runtime performance in terms of lines of source code executed and iteratively added mutations from the next performant individual. Only changes that decrease the number of lines executed and preserve correctness (in terms of output validity on the set of test cases) are retained. We tried all 27 mutations from the best two solvers evolved in the previous experiments.

The resulting 'combined' solver is on average 16 percent faster than the original MiniSAT, as shown in Table 1. In total, this version involved 7 evolved mutations. Details of all the mutations selected are presented in Table 4 and in Fig. 4. Note that new donor code was instrumented by means of IF (0) statements and marked with /**/.

The one IF condition replacement is the one that led to 15 percent speed-up in the second experiment. Two line deletions were one-line assertion removals. The other two corresponded to: deletion of a subtraction operation on a variable used for statistics; and removal of clause optimisation which removes false and duplicate literals. The last two changes remove conditions that check if the solver is in a conflicting state.

By combining the synergistic optimisations found in the three best evolved individuals, our approach produced the fastest specialised SAT solver for CIT among all solvers developed by expert human programmers that were entered into the 2009 MiniSAT-hack competition.

Since small benchmarks were chosen for the training set, the evolved individual might not scale to larger problems. Manual inspection suggests that optimisations relevant to large instances may not be retained, but a systematic evaluation on separate instances is left to future work. However, we note that the evolved individual retained required functionality on all the instances that were held out for verification, even though it was not exposed to any of them during evolution.

Ideally, we would like genetic improvement to be used as part of the build cycle, to recommend improvements to the software engineer. In the Combinatorial Interaction Testing case, the one IF condition replacement (leading to 15 percent speed-up) would be put forward as a recommended software change given its significant impact on solver performance.

The execution times are those for a standard desktop computer. Practising software engineers will likely have more powerful computational resources available for their build process, and we can expect engineering improvements to

```
<IF_Solver_370><IF_Solver_400>
replace 'if (1)' in line 370
        with 'if (0)' from line 400

   370: if (1)  ->  if (0)
        {
        for (i = j = 1; i < out_learnt.size(); i++)
          {
          if (reason[var(out_learnt[i])] == NULL ||
          !litRedundant(out_learnt[i], abstract_level))
                {
                out_learnt[j++] = out_learnt[i];
                }
          }
        }
        else
        {
    /**/  i = out_learnt.size();
    /**/  int found_some = find_removable(out_learnt,
                                 i, abstract_level);
    /**/  if (found_some)
          {
    /**/     j = prune_removable(out_learnt);
          }
    /**/ else
          {
    /**/     j = i;
          }
        }

<_Solver_191>
remove line 191

    191:       learnts_literals -= c.size();

<_Solver_168>
remove line 168

    168: assert(c.size() > 1);

<_Solver_142>
remove line 142

    142: ps.shrink(i - j);

<_Solver_185>
remove line 185

    185: assert(find(watches[toInt(~c[0])], &c));

<IF_Solver_720>
replace if condition in line 720 with 0

    720:  if (!ok || propagate() != NULL)  ->  if (0)
        {
        return ok = false;
        }

<IF_Solver_989>
replace if condition in line 989 with 0

    989:  if (conflict.size() == 0)  ->  if (0)
              {
             ok = false;
                }
```

Fig. 4. Details of the 7 mutations of MiniSAT solver for the CIT domain.

improve the performance of our implementation, which is mearly a research prototype. Therefore, we believe these results provide encouraging initial evidence that the computational time required by GI-based specialisation would allow it to be incorporated in many practical software engineering build processes.

8.2 MiniSAT: Automated Termination Analysis

We ran all solvers from the MiniSAT-hack track competition on the benchmarks obtained from the AProVE termination analysis tool. We identified that the best solver for this set of instances is "MiniSat2hack", that is, the same solver that is the best human-developed solver for the CIT domain, hence we use it as the donor again.

TABLE 5
Normalised Runtime Comparison of MiniSAT Versions, Specialised for AProVE, Based on Averages Over 10 Runs

Solver	Donor	Lines	Time
MiniSAT (original)	—	1.00	1.00
MiniSAT-gp	bestCIT	0.87	0.98
MiniSAT-gp	bestCIT+best09	1.00	1.00
MiniSAT-gp-combined	bestCIT+best09	**0.87**	**0.96**

The "Donor" column indicates the source of the donor code available in the code bank. "Lines" indicates lines of code executed, "Time" indicates CPU time executed. Left column contains the best MiniSAT versions from 3 runs of the GI framework. (Lower is better, all measurements normalized to original MiniSAT).

8.2.1 Transplanting from MiniSAT-bestCIT

We conduct three GI runs, varying the random number seed. By using runtime averages over the whole training set we determine the fastest evolved solver out of the three runs. The best evolved version is 2 percent times faster than the original MiniSAT solver, as shown in Table 5.

Mutations present in the best individual are given in Table 6. Note that 49 percent of the mutations are DELETE operations. Given that only 2 percent runtime improvement was achieved, we suspect that a lot of these mutations might actually have no impact or even a negative impact on solver performance. These might also delete functionality not covered by test cases. This hypothesis will be tested at the filtering stage when we combine best mutations from evolved individuals.

8.2.2 Transplanting from MiniSAT-bestCIT & MiniSAT-best09

In the next experiment we added MiniSAT-best09 donor to the code bank. When specialising for CIT, addition of new code led to a version of MiniSAT that was best overall for the CIT domain. However, for the AProVE domain the reduction both in lines of source code executed as well as runtime has not been significant as shown in Table 5.

TABLE 6
Mutations Occurring in the Best Genetically Improved Solver, Specialised for AProVE, in Three GI Runs

solver	mutation	mutated code	changes
MiniSAT-gp			
	DELETE	IF statement condition	4
	REPLACE	IF statement condition	9
	DELETE	line of code	34
	REPLACE	line of code	5
	COPY	line of code	8
	DELETE	FOR loop condition	3
	REPLACE	FOR loop condition	20
total			83

(Donor: MiniSAT-bestCIT

TABLE 7
Mutations Occurring in the Best Genetically Improved Solver, Specialised for AProVE, in Three GI Runs

solver	mutation	mutated code	changes
MiniSAT-gp			
	DELETE	line of code	1
	REPLACE	FOR loop condition	1
total			2

(Donor: MiniSAT-bestCIT & MiniSAT-best09)

Mutations occurring in the best evolved version of the solver are presented in Table 7.

Only two mutations were present in the best evolved solver version. One assertion was removed. The FOR loop condition replacement actually had no impact on runtime, since the expression i++ was replaced with i++ from a different line of code.

8.2.3 Combining Results

We use our filtering process for greedily combining mutation from the best evolved solvers from the two experiments. This technique leads to a solver that is 4 percent faster than the original MiniSAT on the set of instances from Automated Termination Analysis of Term Re-write Systems problem class as shown in Table 5. The mutations retained are shown in Table 8.

Out of 85 mutations, 16 were retained. These were: 10 assertion removals; deletion of a variable assignment to 0; removal of three addition operations on variables used for statistics; and deletion of two if conditions checking if the solver is in a conflicting state. Each individual change led to less than 1 percent improvement with top 5 being assertion removals. Therefore, it would be up to software developers whether additional time gain is worth getting rid off these assertions.

8.3 MiniSAT: Ensemble Computation

A SAT encoding of the problem of whether a given circuit computes an ensemble consisting of only SUM or OR gates was only proposed as recently as 2012 [56]. In our experiments for this problem domain we used the winner of the MiniSAT-hack track from 2009, since it turned out to be the most efficient out of the human-developed versions of MiniSAT available in the competition. Results for the best evolved MiniSAT versions obtained in the experiments described below are presented in Table 9.

8.3.1 Transplanting from MiniSAT-best09

In the first experiment MiniSAT-best09 and the original solver were used as donors. We conducted three GI runs

TABLE 8
Mutations Occurring in the Combination of the Fastest Genetically Improved Solvers

mutation	mutated code	number of changes
DELETE	IF statement condition	2
DELETE	line of code	14
total		16

TABLE 9
Normalised Runtime Comparison of MiniSAT Versions, Specialised for Ensemble Computation, Based on Averages Over 10 Runs

Solver	Donor	Lines	Time
MiniSAT (original)	—	1.00	1.00
MiniSAT-gp	best09	0.54	0.67
MiniSAT-gp	bestCIT+best09	0.51	*0.64*
MiniSAT-gp-combined	bestCIT+best09	**0.62**	**0.70**

The "Donor" column indicates the source of the donor code available in the code bank. "Lines" indicates lines of code executed, "Time" indicates CPU time executed. Left column contains the best MiniSAT versions from 3 runs of the GI framework. (Lower is better, all measurements normalised to original MiniSAT).

with different pseudo random number seeds. Once again average runtimes on the whole training set were used to determine the best evolved solver from the three runs. The evolved MiniSAT version is 33 percent faster on the whole Ensemble Computation set as shown in Table 9. The evolved mutations are presented in Table 10.

The best versions of solvers evolved in the three experiments contained one mutation that was responsible for the biggest runtime improvement. It removed a line of code in a CASE statement that led to reversing the *polarity mode* of MiniSAT. When an unassigned variable is picked by the solver for assignment, MiniSAT sets it to *false* by default. The one line removal changed that decision to *true*. This change is non-trivial, given that only around 30 percent of literals in all instances are positive, i.e., satisfied by assigning value *true* to them.

Moreover, 55 percent of the mutations in the best individual are deletion operations, a lot of which remove dead code. Note that by instrumenting the solver with IF (0) statements, to introduce new code as described in Section 4, we introduce code that is never executed. The GI process is able to discover the unused lines of code.

8.3.2 Transplanting from MiniSAT-bestCIT & MiniSAT-best09

By allowing GI to have access to code from all three solver versions, a MiniSAT version was evolved that is 36 percent faster than the original as shown in Table 9. Details of

TABLE 10
Mutations Occurring in the Genetically Improved Solver from Three GI Runs, Specialised for Ensemble Computation

solver	mutation	mutated code	changes
MiniSAT-gp			
	DELETE	IF statement condition	7
	REPLACE	IF statement condition	4
	DELETE	line of code	25
	REPLACE	line of code	5
	COPY	line of code	7
	DELETE	FOR loop condition	5
	REPLACE	FOR loop condition	15
	DELETE	WHILE loop condition	1
total			69

(Donor: MiniSAT-best09)

TABLE 11
Mutations Occurring in the Genetically Improved Solvers in Three GI Runs, Specialised for Ensemble Computation

solver	mutation	mutated code	changes
MiniSAT-gp			
	DELETE	IF statement condition	10
	REPLACE	IF statement condition	9
	DELETE	line of code	28
	REPLACE	line of code	9
	COPY	line of code	7
	DELETE	FOR loop condition	8
	REPLACE	FOR loop condition	20
total			91

(Donors: MiniSAT-best09+MiniSAT-bestCIT)

mutations in this best individual are presented in Table 11. Again the best individual contained replacement of a line of code in the CASE statement that reversed polarity of MiniSAT. On about a third of the 50 test cases the best evolved version produced *worse* runtimes than the original solver, but the advantage of reversed polarity for the rest of the instances compensated for this loss.

8.3.3 Combining Results

Finally, we applied the filtering step to the two best solver versions evolved in our previous experiments. The resultant solver is 30 percent faster than the original, Mutations present in the combined version of the solver are shown in Table 8. From a total of 160 mutations only 39 decreased the number of lines executed. After adding each of them in turn (according to the minimum number of lines executed) and checking whether the number of lines decreases, 31 of the 39 mutations remained, shown in Table 12.

Interestingly the best evolved solver in this experiment is 6 percent slower than a previously evolved version, as shown in Table 9. Thus we suspect there exist a mutation or a set of mutations that executes fewer lines, but eventually leads to increase in runtime. Another explanation would be existence of a certain combination of mutations that provide better improvement when applied concurrently rather than individually.

We have looked at all the 31 mutations. Most of these were assertions or operations on statistical variables removals. We identified three mutations responsible for the slowdown: two removed a call to a method, REMOVESATISFIED, that removes clauses that are already satisfied, while the third caused the main loop in that method not to be executed. These three changes accounted for a slowdown, since

TABLE 12
Mutations Occurring in the Combination of the Fastest Genetically Improved Solvers

mutation	mutated code	number of changes
DELETE	IF statement condition	2
DELETE	line of code	26
REPLACE	line of code	2
DELETE	FOR loop condition	1
total		31

TABLE 13
Normalised Runtime Comparison of ImageMagick Versions, Specialised for Greyscale Images, Based on Averages Over 10 Runs

Software	Donor	Fitness threshold	Lines	Time
ImageMagick (original)	—	—	1.00	1.00
ImageMagick-gp	Graphicks-Magick	0%	0.75	1.00
ImageMagick-gp	Graphicks-Magick	50%	0.75	0.97

The "Donor" column indicates the source of the donor code available in the code bank. "Lines" indicates lines of code executed, "Time" indicates CPU time executed. Left column contains the best ImageMagick versions from 3 runs of the GI framework. (Lower is better, all measurements normalized to original ImageMagick).

information for the satisfied clauses is still maintained, thus they are unnecessarily processed during the search process. To verify our findings, we removed the three mutations from the evolved solver and run it on our test set. This version achieved 37 percent speed-up over the original solver.

This finding triggers the question whether there should be a human-in-the-loop in the GI process, so that they could identify parts of the program that should be left unchanged by evolution. Alternatively larger test cases could have been chosen, however, this might not guarantee that all optimisations for large instances will be retained and the overall runtime overhead would be worth it. Ultimately we can recommend individual changes to the code and leave it for the software engineer to decide which should be deployed. For example, polarity mode switch would be suggested for the Ensemble Computation problem class.

8.4 ImageMagick: Grayscale Images

We used two profiling tools, gprof and callgrind, to identify the most time consuming part of ImageMagick when used for converting jpg to png images. We identified the ReadJPEGImage function in the jpeg.c file to be the target for GI optimisation. In all experiments the code bank contains the original code from ImageMagick and additional code from GraphicsMagick from its jpeg.c file. Each GI run was re-run three times varying the random seeds for genetic operators.

8.4.1 Transplanting from GraphicsMagick

We ran two sets of experiments for greyscale images. In the first one we only allowed GI-modified versions of ImageMagick to move to the next generation, if the output png was the same as the one produced by the original software. We say that the fitness threshold is set at 0 percent. In the second set we allowed for 50 percent difference in the output image RGB values, using the mean-squared error metric. We say that the fitness threshold is set at 50 percent.

Runtime comparison with the fastest evolved software for all 100 greyscale testset benchmarks used is shown in Table 13. All versions of the software that are reported in Table 13 produce identical output as the original software. The best evolved version of ImageMagick is, on average, 3 percent faster than the original software. Interestingly, 25 percent reduction in the number of lines executed is

TABLE 14
Mutations Occurring in the Best Genetically Improved Image Processing Software, Specialised for Greyscale Images, from Three GI Runs

solver	mutation	mutated code	changes
ImageMagick-gp			
	DELETE	IF statement condition	4
	REPLACE	IF statement condition	7
	DELETE	line of code	12
	REPLACE	line of code	10
	COPY	line of code	6
	DELETE	FOR loop condition	2
	REPLACE	FOR loop condition	6
total			47

(Donor: GraphicksMagick. Fitness threshold: 0%)

achieved. This is because GI removes lines in a for loop that is used very frequently. In particular, if statement conditionals are removed from inside the loop. The condition evaluation might be quick, yet the if statement body is never actually executed, hence doesn't contribute much towards runtime.

The performance of our evolved version and the human-written version are not different in a statistically significant sense. Changes made by the GI process are shown in Tables 14 and 15.

Interestingly, only 2 of 12 line deletions in the first experiment (with fitness threshold 0 percent) and 1 out of 16 line deletions in the second experiment (with fitness threshold 50 percent) removed an assertion. Most of the code removed was composed of assignment statements. The impact of those changes is, however, unclear.

8.4.2 Combining Results

As in the case with our experiments with MiniSAT, we note that many mutations in the best evolved individuals are independent. We use a filtering strategy to identify the best mutations.

We started with the individual composed of one mutation with the best runtime performance in terms of lines of source code executed and iteratively added mutations from the next performant individual. Only changes that decrease the number of lines executed and preserve correctness are retained. We tried all 110 individual mutations from the best two ImageMagick versions evolved in the previous experiments (4 were the same). 20 of these caused a reduction in lines used. The biggest one was caused by one-line deletion that seems to update a pointer offset from an IndexPacket variable. We conjecture that either the update never happens or the action is immediate hence does not influence the overall runtime significantly.

TABLE 15
Mutations Occurring in the Best Genetically Improved Image Processing Software, Specialised for Greyscale Images, from Three GI Runs

	mutation	mutated code	changes
ImageMagick-gp			
	DELETE	IF statement condition	8
	REPLACE	IF statement condition	7
	DELETE	line of code	16
	REPLACE	line of code	11
	COPY	line of code	12
	DELETE	FOR loop condition	5
	REPLACE	FOR loop condition	8
total			67

(Donor: GraphicksMagick. Fitness threshold: 50%)

TABLE 16
Normalised Runtime Comparison of ImageMagick Versions, Specialised for Colour Images, Based on Averages Over 10 Runs

Software	Donor	Fitness threshold	Lines	Time
ImageMagick (original)	—	—	1.00	1.00
ImageMagick-gp	Graphicks-Magick	0%	0.97	1.00
ImageMagick-gp	Graphicks-Magick	50%	0.97	1.00

The "Donor" column indicates the source of the donor code available in the code bank. "Lines" indicates lines of code executed, "Time" indicates CPU time executed. Left column contains the best ImageMagick versions from 3 runs of the GI framework. (Lower is better, all measurements normalised to original software).

We also investigated runtimes of the individuals composed of single-line mutations. None of these produced a faster version of software when compared with the original. ImageMagick consists largely of if statements and for loops. The mutations that led to improvment in terms of lines used modified a statement within a for loop or disabled an if condition. Since the png images produced by mutated software did not vary from the one produced by original ImageMagick, we conjecture that those if conditions evaluate to *false* in the original software. GI-modified software simply avoids unnecessary checks. Therefore, shows potential for improving legacy software.

8.5 ImageMagick: Colour Images

We repeated the set of experiments outlined in the previous section with a different training set. This time we focused on colour images and the jpg to png conversion functionality of ImageMagcik as before. We also used two fitness functions. Only those individuals that produced a valid png image that was identical to the output of the original software were moved onto the next generation. To allow GI to explore a larger search space we relaxed this condition by allowing 50 percent difference in the RGB values when comparing using the MSE image comparison metric. In both experiments GI evolved versions of software that preserve image output characteristics. We report on the fastest of such ImageMagick software variants in Table 16.

8.5.1 Transplanting from GraphicsMagick

In both sets of experiments we use GraphicsMagick as a source of the donor code. Mutations produced by the two best evolved versions are presented in Tables 17 and 18.

Neither of the best evolved individuals, in terms of lines used, led to runtime improvements. It is worth mentioning, however, that an if statement that was previously in GraphicsMagick was transplanted into ImageMagick. However,

TABLE 17
Mutations Occurring in the Genetically Improved Solver from Three GI Runs, Specialised for Colour Images

	mutation	mutated code	changes
ImageMagick-gp			
	DELETE	IF statement condition	10
	REPLACE	IF statement condition	12
	DELETE	line of code	34
	REPLACE	line of code	14
	COPY	line of code	16
	DELETE	FOR loop condition	8
	REPLACE	FOR loop condition	12
total			106

(Donor: GraphicksMagick, Fitness threshold: 0%)

TABLE 18
Mutations Occurring in the Genetically Improved Solver from Three GI Runs, Specialised for Colour Images

	mutation	mutated code	changes
ImageMagick-gp			
	DELETE	IF statement condition	9
	REPLACE	IF statement condition	17
	DELETE	line of code	23
	REPLACE	line of code	15
	COPY	line of code	25
	DELETE	FOR loop condition	3
	REPLACE	FOR loop condition	11
total			103

(Donor: GraphicksMagick, Fitness threshold: 50%)

that mutation did not lead to any statistically significant runtime improvement.

8.5.2 Combining Results

Finally, we evaluated each mutation from the best individuals in turn. However, since none of them produced runtime improvments on the test set, we did not proceed with the filtering step as in the case of MiniSAT.

8.6 Summary

We summarise our findings and answer the following research question posed at the beginning of this paper.

RQ1, Effectiveness and Efficiency. What is the effectiveness and efficiency of GI-based specialisation?

RQ1a, Effectiveness. Can genetic improvement find faster specialised software versions than any general version developed and optimised by expert human programmers?

RQ1b, Efficiency. What is the computation cost of the specialisation process?

To sum up, we found MiniSAT solver versions achieving between 4 and 36 percent runtime improvement. We verified these results with uninstrumented versions of MiniSAT (without source code calculating lines of code, IF(0) statements etc.). We manually inserted changes produced in the final evolved solver versions into the original Solver.C file and noted only up to 2 percent discrepancy (based on averages over 20 runs).

Additionally, these automatically evolved solvers are faster than *any* of the general purpose human-optimised general solvers from the first edition of the MiniSAT-hack track competition for the three applications investigated.

Efficiency gain of ImageMagick was less significant. The best individual achieved up to 3 percent runtime improvement. However, the number of executed lines by the best evolved version of ImageMagick reduced the number of lines by 25 percent.

This thus provides an answer to *RQ1a*, that is, *genetic improvement can find faster specialised software versions*. The efficiency improvement can, however, vary.

Answering *RQ1b*, in the Combinatorial Interaction Testing case, *each run of the genetic framework took 9 hours, while for the other two problem classes each GI run took just under two days*. The reason for this runtime difference is that the instances from the Automated Termination Analysis and Ensemble Computation sets that were used for training simply take longer to run than the CIT instances. This shows that the efficiency of GI is highly dependant on the training set. Since significant improvement was achieved for CIT, we believe that using small instances is enough to apply the GI framework. However, evolution might target certain optimisations for large instances. This poses a challenge to the current GI framework. Evaluations of ImageMagick on 100 small images (less than 1 MB) took 5 hours each. This raises the question of trade-off between the GI effort and potential optimisation gains.*RQ2, Code Bank Size.* How well does the genetic improvement approach perform depending on the size of the code bank?

Furthermore, Tables 1, 5 and 9 provide an answer to *RQ2*, namely, *the size of the code bank has negligible impact on the performance of the genetic improvement framework*. In particular, a more efficient version of the solver was found when larger code bank was used in the Combinatorial Interaction Testing case, but the reverse was true for the Automated Termination Analysis problem class. By allowing GI access to a larger code bank, we enlarge the search space for possible changes. Therefore, efficient optimisations from other software variants might be harder to find. One could approach this problem by either adding weights to the code that's transplanted, so that it is mutated more frequently, or increase population and generation size. Further research needs to be done to address this issue.

RQ3, Number of Modifications Required. Does our filtering technique produce the most efficient solver variants when compared with the ones evolved directly by genetic improvement and how many interventions are recommended?

Experiments conducted for the Ensemble Computation problem class provide an answer to *RQ3*. *Our filtering technique does not always produce the most efficient solver variant when compared with the ones evolved directly by genetic improvement*. In that case, as shown in 9, the filtering step produced a less efficient program than the one evolved directly by the genetic improvement approach. In the case of ImageMagick, the runtimes of the best evolved individuals were almost identical with the run of the original software.

RQ4a, Specificity. Are the changes produced by GI problem-specific?

RQ4b, Generality. Are the changes produced by GI general efficiency improvements?

Overall, in all experiments there were certain generalist mutations such as assertion removals and deletion of

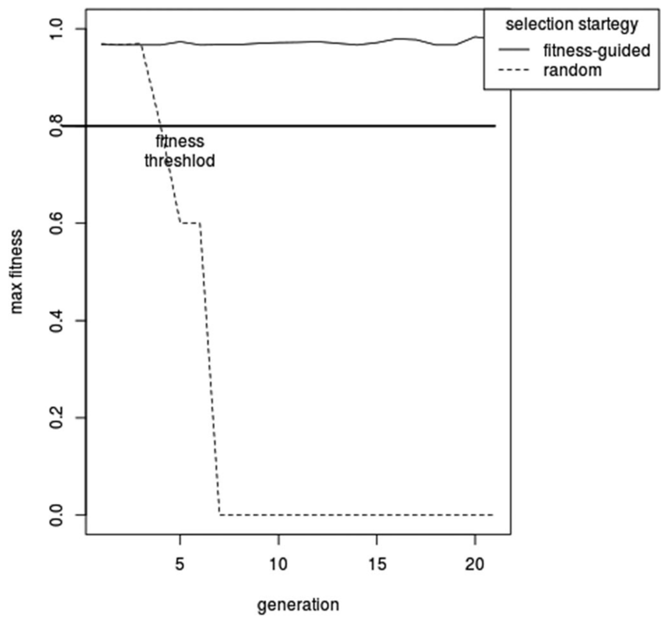

Fig. 5. Maximum fitness comparison with random search. GI framework was run on MiniSAT2-070721 with CIT test cases, varying individual selection strategy. After five generations all individuals had fitness value 0, i.e., did not compile or always returned an invalid output.

operations on variables used for statistics (such as learnt clauses counter). However, GI also found a few domain-specific changes. For the CIT domain, transplantation of functionality from another MiniSAT variant turned out the most fruitful, whilst polarity mode switch was the most effective for the Ensemble Computation domain, as shown in Section 8. This switch was not efficient for the CIT and Automated Termination Analysis as shown in previous work on energy optimisation of MiniSAT using genetic improvement [45]. In the ImageMagick experiments, different mutations were promoted, depending on whether grey-scale or colour images were targeted for conversion optimisation. Answering *RQ4*, *the genetic improvement approach produces mutations that are problem-specific, but is also able to evolve changes leading to general efficiency improvements*.

9 Analysis

Next we report results of modifying various aspects the genetic improvement framework. We focus on the experiments involving MiniSAT only, since the evolved solver versions showed a range of efficiency behaviour based on the seed to the GI process and the downstream application.

9.1 Tuning the Fitness Function

We investigated various fitness function parameters to answer the following research questions:

RQ5a, Fitness Function. What is the impact of the trade-off between efficiency and effectiveness in fitness function on finding a specialised software version using GI?

RQ5b, Comparison to Random Search (Sanity Check [57]). How does the chosen search strategy compare with random search?

Before we set the fitness in our experiments, we varied the trade-off between efficiency and correctness (and did not use donor transplantation technique) in order to identify a fitness function that would guide GP towards faster (but still correct) individuals. We performed GP runs with the following sets of fitness thresholds:

1) the evolved individual must be at least as quick as the original solver
2) the evolved individual must be correct in 2 out of 5 test cases and must be at least as quick as the original solver
3) the evolved individual must be correct in 3 out of 5 test cases and must be at least as quick as the original solver
4) the evolved individual must be correct in 4 out of 5 test cases and must be at least as quick as the original solver

The answer to *RQ5a* is as follows: in cases 1), 2) and 3) the individuals with highest fitness values were those that always produced a SATISFIABLE or UNSATISFIABLE answer. In those cases the DELETE mutation was used the most frequently. The functionality of the solver algorithm was being switched off, leaving just the 'return SATISFIABLE' or 'return UNSATISFIABLE' statements. Similar results were obtained in experiment 4), however, at least a few correct individuals were created, but their performance was statistically similar to the original, producing better runtime results for only 2 percent of instances.

We also performed a comparison with random search by switching off individual selection based on fitness value in our GI framework. After five generations the search produced entire populations of individuals that simply did not compile or produced errors and thus had zero fitness, which answers *RQ5b*. The result of the experiment conducted is shown in Fig. 5. By using a fitness-guided selection strategy we were able to evolve multiple individuals that met the fitness threshold and were faster than the original program.

9.2 Tuning the Genetic Operation Rates

We also asked:

RQ5c, Genetic Operators. What is the impact of various mutation and crossover operator rates on GI efficacy?We have noticed that frequently a few mutations have a huge impact on MiniSAT performance. Therefore, we tried the following mutation and crossover rates: {mutation: 75 percent, crossover: 25 percent} and {mutation: 25 percent, crossover: 75 percent}. However, none of these experiments led to a version of MiniSAT that was better than the one evolved in the work described in Section 8. This answers *RQ5c*. It is unclear when crossover would be helpful in GI work. Given the result by Gabel et al. [58] about software uniqueness, it is possible that at least 6 one-line changes need to be made to have a significant influence on runtime. In the current setup crossover plays a major role in the number of mutations in individual software variants. Further investigations into the impact of crossover and mutation in genetic improvement work needs to be undertaken. Another issue is the choice of genetic operators. Perhaps a more fine-grained mutation, that provided expression-level changes, would yield better results. Deep parameter tuning work would be suggested as a future direction [59].

9.3 Test Case Selection

In our previous work we produced a version of MiniSAT that was 17 percent faster on instances from the CIT application domain [9]. However, some mutations did not scale to large real-world CIT instances. Thus we re-ran the GI framework with larger instances. This experiment has taken proportionally longer to run and has not produced a faster version with more generalisable mutations. Best results were obtained efficiently with small test cases.

Moreover, mutations produced by our GI framework are at the source code level, hence are easily readable. Therefore, programmers can then decide which changes should be applied.

Current GI process relies heavily on the assumption that the test cases capture the desired software behaviour. Therefore, more work is needed to help the developer decide which ones to select for the GI process.

9.4 Output Validity

Since program correctness is measured by the pass rate of test cases, the evolved version might potentially not be valid on other, non-tested cases. However, we verified each solution produced by the final evolved solvers using the EDACC verifier tool available from SAT 2013 competition website: http://www.satcompetition.org/2013/downloads.shtml. Furthermore, for the CIT domain, for example, output validity can be checked in polynomial-time; it is the time to generate CIT test suites that is an issue.

Hence, depending on the application, it might be beneficial to have a program that is, say, 30 percent faster, but correct in 9 out of 10 cases (as long as output can be verified efficiently). Multi-objective optimisation could be applied to investigate the various trade-offs.

Furthermore, GI-generated changes have already been adapted into development. Langdon et al. [60] sped-up a DNA sequence mapping program using GI. The generated edits were submitted to the software development team, who incorporated these into the next software release. This shows that GI can already serve as a recommender system for software developers. Furthermore, since certain mutations produced by the GI framework are non-obvious to human developers, an automated approach that could verify such changes would overcome the issue of validity. Another idea would be to introduce an automated rollback functionality.

9.5 Masking Effect

In Section 8.3.2 we observed that if an individual contains a mutation that greatly increases it's fitness value, it may also contain mutations that actually hinder solver performance and still be selected in the next generation. One way to avoid this, would be to dynamically adapt the fitness function, based on fitness of the best individual found so far. Another idea would be to employ a hill-climbing algorithm instead of genetic algorithm in the search process. However, strict hill climbing could miss individuals where certain combinations of mutations lead to program speed-up.

9.6 Search Strategy

We used genetic programming within our GI framework to find specialised program versions. Plots of the mean fitness values in three of our six experiments (with 0-fitness value individuals excluded) are shown in Fig. 6. Given that there isn't an obvious increase in fitness value with the number of generations and that there exist individual mutations that lead to a significant runtime improvement, as shown in Section 8, a question arises whether the GI framework could benefit from another search strategy, such as hill climbing. Furthermore, the graphs show that there's little gain to be had in later generations. Perhaps deep learning strategies could be used to obtain a more fine-grained fitness function that would lead to better results with the genetic programming approach used. It would be good if the fitness function could be specified based on the application. As solution quality and speed should not have the same priority always for all applications.

9.7 Benchmark Structure

The best improvement was obtained for the Ensemble Computation problem class. For this set of benchmarks the winner of the 2009 MiniSAT-hack track competition was the best human-developed version of the solver, in contrast to the other two problem classes. Given that the biggest runtime improvement was obtained by simply switching the polarity mode of MiniSAT from *false* to *true*, one might argue that the instances from the Ensemble Computation class simply contain more positive literals than negative ones. This is, however, not the case.

Even though the general SAT problem is NP-complete, SAT solvers are generally extremely fast at solving industrial instances. It has been shown that such real-world benchmarks usually contain a small set of variables, that once set correctly, make the rest of the instance easily solvable. These are known as backdoors [61]. Perhaps reversing the polarity mode caused the backdoor variables to be set to correct values more quickly than when using the default settings.

More recently, a strong connection was found between MiniSAT's runtime and graph modularity of SAT instances, denoted by Q [62], [63]. In particular, SAT instances with $0.05 \leq Q \leq 0.13$ are much harder to solve for MiniSAT than others. Industrial instances were found to have high graph modularity, frequently above 0.3. We calculated the Q values for all instances used in our experiments using SATGraf.[25] For the CIT instances, the mean Q value is 0.24; for the AProVE instances, the mean Q value is 0.46; while for the Ensemble Computation class the mean Q value is the highest: 0.52. Given that several CIT instances had Q values in the 'hard-to-solve' range, the 16 percent improvement achieved by our GI approach (see Table 1) shows great potential for using transplantation as means of specialising software.

9.8 Code Bank

In our work we used code that has been developed by expert human programmers. Several software variants were available from the competition devoted to optimising MiniSAT. The question arises what is a good source for the code bank. As shown in previous work on bug fixing, the original software is a good source of such code (the 'plastic

25. SATGraf tool is available at https://ece.uwaterloo.ca/~vganesh/EvoGraph/Download.html. We used the 'cnm' algorithm.

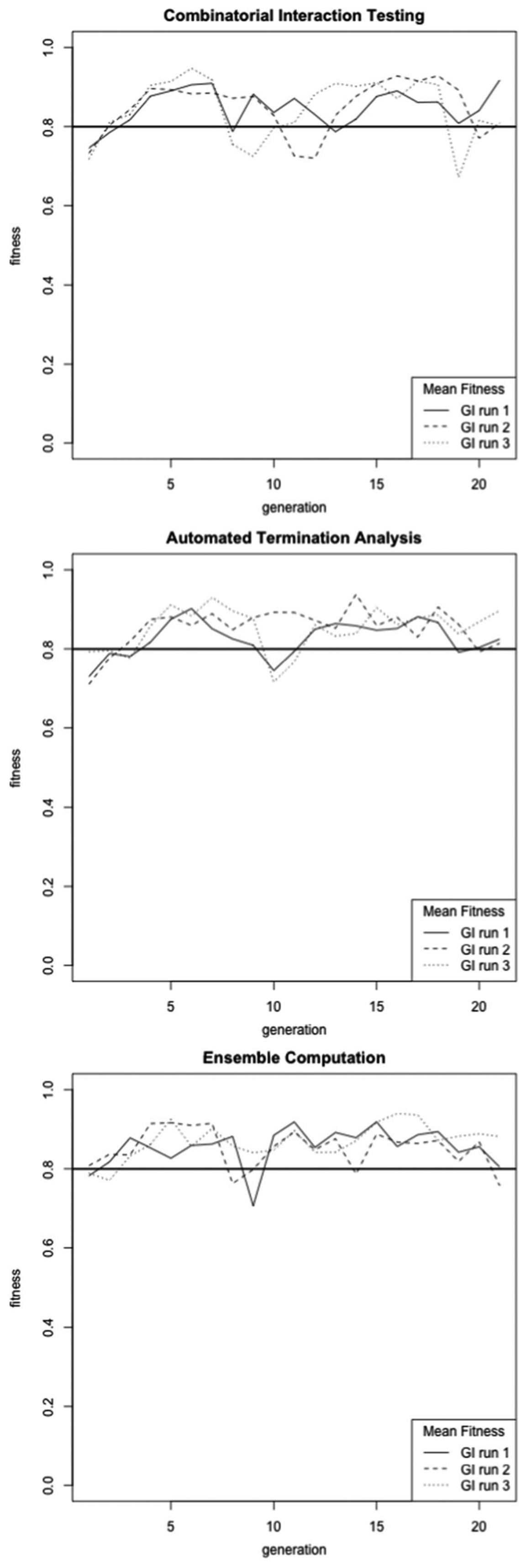

Fig. 6. Mean fitness values throughout 20 generations with 0-fitness individuals excluded. GI framework was run on MiniSAT2-070721.

surgery hypothesis' [64]). We advocate that our approach for software specialisation generalises to instances where one has access to multiple software variants. Where such variants do not exist, we cannot apply the transplantation approach. However, software variants can be found, for instance, in software revision histories or related projects (multiple pieces of software performing the same task, e.g., in open-source repositories). We picked an example of forking for the second set of experiments to show another area where the approach can be applied to.

10 Threats to Validity and Limitations

Our experiments show that, indeed, it is possible to automatically modify existing, highly-optimised code, in order to achieve runtime improvements. We used genetic improvement with software transplantation to achieve this. The range of results varies, depending on the downstream application and software of choice, of course. We have only reported results for two applications and a total of five downstream specialisation scenarios. More research is required to investiage the degree to which these generalise.

Genetic improvement is a new research area, hence there are not yet any guidelines in terms of how to setup a GI framework to evolve an optimised software version effectively and efficiently. In the previous section we mentioned several issues that might influence the success of genetic improvement techniques for runtime improvements. We presented empirical data, yet there is more work to be done.

The set of test cases that preserve software behaviour is not yet well defined. Furthermore, many programs do not come with test suites that provide good coverage. Therefore, the approach would not be immediately applicable.

One needs to consider each software improvement framework separately. In the case of MiniSAT, we were able to quickly verify the output of software. A SAT instance can be either satisfiable or not and we used benchmarks for which satisfiability is known. In the case of ImageMagick we chose a particular image comparison metric. However, one might argue that image quality might be sacrificed, for example, by 1 percent, if the software can process images much faster than the original software. The number of test cases needed for GI also requires further understanding of the genetic improvement process.

Traditionally, small mutations are applied to the code in the form of one-line copy, replace and delete operations. These have shown some success, including in this work, yet further research needs to be done into at which level of granularity the changes should be made. Another question relates to the crossover rate. In the MiniSAT experiments, the earlier generations produced the best results. The trend was reverse for the ImageMagick software.

Finally, output validity was measured by the number of test cases passed as a proxy for correctness in the fitness function. We argue that GI can draw here from the literature on genetic programming. However, we hope that the GI techniques will also investigate other search-based techniques in the quest of exploring the vast space of possible software mutations.

11 Related and Future Work

Our approach to program specialisation is based on Genetic Improvement. GI uses computational search to improve existing software while retaining its partial functionality. GI has been successfully used to automate the process of bug repair [28], in which the improved program has one or more bugs fixed that were present in the original program. The achievements of genetic programming to improve existing programs (by patching their bugs) has led to an explosion of interest in automated program repair [4]. It has also been successful in other areas of software engineering, such as reverse engineering [30].

GI described here is a specific instance of the application of evolutionary computation to software engineering problems [65], an area that has come to be known as Search Based Software Engineering (SBSE) [66], [67], [68], [69]. SBSE has been applied to many problems in software engineering such as project management [70], requirements engineering [71], design [72], [73], testing [29], [74], [75], maintenance [76], reverse engineering [30], refactoring and code smell detection [77], [78], [79]. Whereas many areas of SBSE target software engineering processes and non-code-based aspects of the overall software system, GI targets the source code itself, so it is a form of source code analysis and manipulation [80].

GI has been used to improve non-functional properties of relatively small laboratory programs [5], [7], [27], as well as larger real-world systems [2], [3], [81], [82], [83], [84]. It has also been used to automatically migrate a system from one platform to another [3], to improve energy consumption [45], [85] and to graft novel functionality into large existing systems [86], [87], [88].

Previous work on genetic improvement was concerned with a single program; the program to be improved. Code is extracted, perhaps modified and then reinserted back into the program at a different location. In most cases, the code to be inserted is taken from elsewhere in the original program [2], [3], [5], [6], [7], [27], [28], [81], but can also come from other programs written by human programmers [89], or be grown from scratch by genetic programming [86], [87].

Our focus is on transplantation from multiple programs for specialisation. This is an important departure from the previous literature in GI. As a result of multiple transplantation, GP is no longer concerned with a single program to be improved, but multiple donor programs, from which code can be extracted to help guide genetic improvement. The idea of code transplantation using GP was proposed by Harman et al. [30] and first implemented by Petke et al. [9]. In subsequent work, Barr et al. successfully transplanted a feature from one program into another using genetic improvement [89]. Program transplantation is a general approach (taking code from one human-written system and inserting it into another), but here we use it for GI-based specialisation.

The idea of transplanting code to improve its behaviour has also been investigated in the context of replacing legacy code with external components [90], [91], [92], [93]. Our approach could also be applied to this problem. Furthermore, it is more flexible in terms of granularity of the changes. For example, just one line of code from the donor could be transplanted using GI.

The goal of improvement adopted here also differs from that of most previous GI work, which focused on improving functional properties (by bug repair [4] and grafting [86]) or non-functional properties (such as execution time [2], [3], [5], [7], [27], [81], [82], [83] and energy consumption [6], [45], [85], [94], [95]).

In all of this previous GI work, the full functionality of the original program (notwithstanding any buggy behaviour) was to be retained in the genetically improved version of the program. By contrast, we explore GI's potential to specialise a program to a particular application domain. The specialised program need not retain the full functionality of the original. Therefore, it can optimise, outperforming the original program for the specific task for which it has been evolved.

In preliminary experiments with MiniSAT [96], optimisation through genetic improvement of general SAT competition instances was conducted. However, this approach led to only very modest runtime improvements of up to 2 percent. Using transplantation from various different versions of MiniSAT, we have been able to use GI-based specialisation to achieve a (human-competitive) improvement of 17 percent for the specialised application subdomain of Combinatorial Interaction Testing [9]. Here we extend our previous work [9] to show that MiniSAT can be specialised for multiple downstream application subdomains and add results for another piece of software, i.e., ImageMagick.

GI-based specialisation shares the goal of previous work on partial evaluation [12], [13], [14]. That is, both GI-based specialisation and partial evaluation seek to produce, from an original general program, multiple specialised versions that target some subset of the original's application domain.

However, the criterion for specialisation, the techniques used to specialise, and the specialised programs that result from each of the two approaches are all very different. That is, unlike partial evaluation, GI-based specialisation uses evolution, in the form of genetic programming, to search for specialised programs, whereas partial evaluation uses a sequence of meaning preserving transformations. The specialisation criterion for partial evaluation is a subset of inputs, or some predicate over the input space, where as for GI-based specialisation, the specialisation criterion is captured by a set of test cases (inputs and the corresponding desired output). Partial evaluation is also deterministic, whereas GI-based specialisation presented in this work is inherently stochastic, since it is based on evolutionary computation.

The vast majority of current genetic improvement work relies on a genetic programming algorithm. This has proven very successful in the automated program repair work [28].

However, GI can also use other SBSE approaches in order to search the space of different software variants. These, however, are yet to be tried.

We have chosen the original program to be the test oracle [97], determining the output corresponding to each input, and thereby constructing the test cases that guide genetic programming. This means that each specialisation targets some sub-problem within the overall problem space attacked by the original program. However, our formulation of specialisation as a problem for GI allows us to apply specialisation to problems where the specialised program must behave *differently* to the original program.

This could be useful in situations where the original program is not only too general for a particular problem (and therefore unnecessarily slow), but where it also fails to quite fit the specialised problem; the original program needs to be specialised *and* (slightly) adapted. By careful inclusion of a few additional test cases (that capture the desired new behaviour) we may be able to specialise the program and simultaneously adapt it, using the same GI-based specialisation process advocated here. It is worth noting that there is a time cost associated with setting up and running the GI framework. Investigation of this specialise-and-adapt problem and its efficiency remains an open challenge for future work.

12 Conclusions

We evolved specialised versions of the C++ program, MiniSAT, and ImageMagick, image processing software, using genetic improvement with transplants. Genetic improvement specialised MiniSAT for three particular hard problem classes and ImageMagick for two different types of images.

The MiniSAT-hack track of SAT competitions is specifically designed to encourage human scientists and engineers to adapt and develop MiniSAT code to find runtime improvements, and hence lead to new insights into SAT solving technology. The competition provides a natural source of genetic material for code transplants, as well as a natural baseline for assessing the competitiveness of the GI-specialised versions against the general versions optimised by expert humans.

We applied GI-based specialisation to three problem domains: Combinatorial Interaction Testing, Automated Termination Analysis of Term Re-write Systems and the problem of Ensemble Computation. The evolved MiniSAT versions achieved between 4 and 36 percent runtime improvement on our test set over the best general solver optimised by humans. For all three problem domains, the evolved solvers outperform *all* of the general human-written solvers entered into the 2009 MiniSAT-hack track competition, when applied to problems from the specialised domain.

We also applied the genetic improvement approach to ImageMagick. We targeted optimisation of the JPG to PNG conversion function. We used code from GraphicsMagick, that was forked from ImageMagick. That code was used as a pool of source code from which to draw candidates for transplantation. The best evolved individual achieved a 3 percent runtime improvement with respect to the original software. Future work may explore relaxed forms of equivalence that woud permit greater degrees of speed up.

We believe that these findings provide compelling evidence to support the claim that GI-based specialisation is a promising approach to automated program specialisation. We also believe that more research needs to be done into finding optimal GI setups for a given software application.

References

[1] M. Harman, W. B. Langdon, Y. Jia, D. R. White, A. Arcuri, and J. A. Clark, "The GISMOE challenge: Constructing the Pareto program surface using genetic programming to find better programs (keynote paper)," in *Proc. 27th IEEE/ACM Int. Conf. Automated Softw. Eng.*, Sep. 2012, pp. 1–14.

[2] W. B. Langdon and M. Harman, "Optimising existing software with genetic programming," *IEEE Trans. Evol. Comput.*, vol. 19, no. 1, pp. 118–135, Feb. 2015.

[3] W. B. Langdon and M. Harman, "Evolving a CUDA kernel from an nVidia template," in *Proc. IEEE World Congr. Comput. Intell.*, 18–23 Jul. 2010, pp. 2376–2383.

[4] C. Le Goues, S. Forrest, and W. Weimer, "Current challenges in automatic software repair," *Softw. Quality J.*, vol. 21, no. 3, pp. 421–443, 2013.

[5] M. Orlov and M. Sipper, "Flight of the FINCH through the Java wilderness," *IEEE Trans. Evol. Comput.*, vol. 15, no. 2, pp. 166–182, Apr. 2011.

[6] D. R. White, J. Clark, J. Jacob, and S. Poulding, "Searching for resource-efficient programs: Low-power pseudorandom number generators," in *Proc. Genetic Evol. Comput. Conf.*, Jul. 2008, pp. 1775–1782.

[7] D. R. White, A. Arcuri, and J. A. Clark, "Evolutionary improvement of programs," *IEEE Trans. Evol. Comput.*, vol. 15, no. 4, pp. 515–538, Aug. 2011.

[8] N. Eén and N. Sörensson, "An extensible SAT-solver," in *Theory and Applications of Satisfiability Testing*. Berlin, Germany: Springer, 2004, pp. 502–518.

[9] J. Petke, M. Harman, W. B. Langdon, and W. Weimer, "Using genetic improvement and code transplants to specialise a C++ program to a problem class," in *Proc. 17th Eur. Conf. Genetic Program.*, 2014, pp. 137–149.

[10] J. P. M. Silva, I. Lynce, and S. Malik, "Conflict-driven clause learning SAT solvers," in *Handbook of Satisfiability*, A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds. Amsterdam, The Netherlands: IOS Press, 2009, pp. 131–153.

[11] "MiniSAT-hack track of SAT competition," In 2009 this was part of the 12th International Conference on Theory and Applications of Satisfiability Testing, 2009. [Online]. Available: http://www.satcompetition.org/2009/

[12] D. Bjørner, A. P. Ershov, and N. D. Jones, *Partial Evaluation and Mixed Computation*. Amsterdam, The Netherlands: North-Holland, 1987.

[13] D. Binkley, S. Danicic, M. Harman, J. Howroyd, and L. Ouarbya, "A formal relationship between program slicing and partial evaluation," *Formal Aspects Comput.*, vol. 18, no. 2, pp. 103–119, 2006.

[14] N. D. Jones, C. K. Gomard, and P. Sestoft, *Partial Evaluation and Automatic Program Generation*. London, U.K.: Prentice-Hall, 1993.

[15] C. Consel, L. Hornof, R. Marlet, G. Muller, S. Thibault, and E.-N. Volanschi, "Partial evaluation for software engineering," *ACM Comput. Surveys*, vol. 30, no. 3, Sep. 1998, Art. no. 20. [Online]. Available: http://www.acm.org:80/pubs/citations/journals/surveys/1998–30-3es/a20-consel/

[16] S. Draves, "Partial evaluation for media processing," *ACM Comput. Surveys*, vol. 30, no. 3, Sep. 1998, Art. no. 21. [Online]. Available: http://www.acm.org:80/pubs/citations/journals/surveys/1998–30-3es/a21-draves/

[17] M. Dwyer, J. Hatcliff, and M. Nanda, "Using partial evaluation to enable verification of concurrent software," *ACM Comput. Surveys*, vol. 30, no. 3, Sep. 1998, Art. no. 22. [Online]. Available: http://www.acm.org:80/pubs/citations/journals/surveys/1998–30-3es/a22-dwyer/

[18] C. K. Gomard and N. D. Jones, "Compiler generation by partial evaluation," in *Proc. 11th IFIP World Comput. Congr. Inf. Process.*, 1989, pp. 1139–1144.

[19] C. Cadar, P. Pietzuch, and A. L. Wolf, "Multiplicity computing: A vision of software engineering for next-generation computing platform applications," in *Proc. Workshop Future Softw. Eng. Res.*, 2010, pp. 81–86.

[20] L. Beckman, A. Haraldson, O. Oskarsson, and E. Sandewall, "A partial evaluator, and its use as a programming tool," *Artif. Intell.*, vol. 7, no. 4, pp. 319–357, 1976.

[21] A. P. Ershov, *On the Essence of Computation*. Amsterdam, The Netherlands: North-Holland, 1978, pp. 391–420.

[22] Y. Futamura, "Partial evaluation of computation process-an approach to a compiler-compiler," *Syst. Comput. Controls*, vol. 2, no. 5, pp. 721–728, Aug. 1971.

[23] A. Haraldsson, "A partial evaluator, its use for compiling iterative statements in Lisp," in *Proc. Conf. Rec. 5th Annu. ACM Symp. Principles Program. Languages*, Jan. 1978, pp. 195–202.

[24] Y. Futamura and K. Nogi, "Generalized partial computation," in *Proc. IFIP TC2 Workshop Partial Eval. Mixed Comput.*, 1987, pp. 133–151.

[25] R. Poli, W. B. Langdon, and N. F. McPhee, *A Field Guide to Genetic Programming*. Egham, U.K.: Lulu Enterprises, 2008.

[26] J. R. Koza, *Genetic Programming-on the Programming of Computers by Means of Natural Selection*. Cambridge, MA, USA: MIT Press, 1993.

[27] A. Arcuri, D. R. White, J. A. Clark, and X. Yao, "Multi-objective improvement of software using co-evolution and smart seeding," in *Proc. 7th Int. Conf. Simulated Evol. Learn.*, Dec. 2008, pp. 61–70.

[28] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A generic method for automatic software repair," *IEEE Trans. Softw. Eng.*, vol. 38, no. 1, pp. 54–72, Jan./Feb. 2012.

[29] M. Harman, Y. Jia, and Y. Zhang, "Achievements, open problems and challenges for search based software testing," in *Proc. 8th IEEE Int. Conf. Softw. Testing Verification Validation*, 2015, pp. 1–12.

[30] M. Harman, W. B. Langdon, and W. Weimer, "Genetic programming for reverse engineering," in *Proc. 20th Working Conf. Reverse Eng.*, 14–17 Oct. 2013, pp. 1–10.

[31] C. L. Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *Proc. Int. Conf. Softw. Eng.*, 2012, pp. 3–13.

[32] A. Zeller, "Yesterday, my program worked. Today, it does not. Why?" in *Proc. 7th Eur. Softw. Eng. Conf. Held Jointly 7th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 1999, pp. 253–267.

[33] S. D. Prestwich, "CNF encodings," in *Handbook of Satisfiability*, A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds. Amsterdam, The Netherlands: IOS Press, 2009, pp. 75–97.

[34] A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds., *Handbook of Satisfiability*, vol. 185. Amsterdam, The Netherlands: IOS Press, 2009.

[35] Y. Vizel, G. Weissenbacher, and S. Malik, "Boolean satisfiability solvers and their applications in model checking," *Proc. IEEE*, vol. 103, no. 11, pp. 2021–2035, Nov. 2015.

[36] V. D'Silva, D. Kroening, and G. Weissenbacher, "A survey of automated techniques for formal software verification," *IEEE Trans. CAD Integr. Circuits Syst.*, vol. 27, no. 7, pp. 1165–1178, Jul. 2008. [Online]. Available: http://dx.doi.org/10.1109/TCAD.2008.923410

[37] F. Arito, F. Chicano, and E. Alba, "On the application of SAT solvers to the test suite minimization problem," in *Proc. 4th Int. Symp. Search Based Softw. Eng.*, 2012, pp. 45–59. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-33119-0_5

[38] M. Banbara, H. Matsunaka, N. Tamura, and K. Inoue, "Generating combinatorial test cases by efficient SAT encodings suitable for CDCL SAT solvers," in *Proc. 17th Int. Conf. Logic Program. Artif. Intell. Reasoning*, 2010, pp. 112–126.

[39] T. Nanba, T. Tsuchiya, and T. Kikuno, "Constructing test sets for pairwise testing: A SAT-based approach," in *Proc. Int. Conf. Netw. Comput.*, 2011, pp. 271–274.

[40] A. Yamada, T. Kitamura, C. Artho, E. Choi, Y. Oiwa, and A. Biere, "Optimization of combinatorial testing by incremental SAT solving," in *Proc. 8th IEEE Int. Conf. Softw. Testing Verification Validation*, 2015, pp. 1–10. [Online]. Available: http://dx.doi.org/10.1109/ICST.2015.7102599

[41] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, and Y. L. Traon, "Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test suites for large software product lines," *IEEE Trans. Softw. Eng.*, vol. 40, no. 7, pp. 650–670, 2014.

[42] D. Benavides, S. Segura, and A. R. Cortés, "Automated analysis of feature models 20 years later: A literature review," *Inf. Syst.*, vol. 35, no. 6, pp. 615–636, 2010. [Online]. Available: http://dx.doi.org/10.1016/j.is.2010.01.001

[43] A. Milicevic, J. P. Near, E. Kang, and D. Jackson, "Alloy*: A general-purpose higher-order relational constraint solver," in *Proc. 37th IEEE/ACM Int. Conf. Softw. Eng.*, 2015, pp. 609–619. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2015.77

[44] J. Marques-Silva , "Practical applications of Boolean satisfiability," in *Proc. 9th Int. Workshop Discrete Event Syst.*, May 2008, pp. 74–80.

[45] B. R. Bruce, J. Petke, and M. Harman, "Reducing energy consumption using genetic improvement," in *Proc. Genetic Evol. Comput. Conf.*, 2015, pp. 1327–1334.

[46] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Comput. Surveys*, vol. 43, no. 2, pp. 11:1–11:29, 2011.

[47] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The AETG system: An approach to testing based on combinatorial design," *IEEE Trans. Softw. Eng.*, vol. 23, no. 7, pp. 437–444, Jul. 1997.

[48] C. Colbourn, "Covering array tables," 2013. [Online]. Available: http://www.public.asu.edu/ ccolbou/src/tabby/catable.html

[49] B. J. Garvin, M. B. Cohen, and M. B. Dwyer, "Evaluating improvements to a meta-heuristic search for constrained interaction testing," *Empirical Softw. Eng.*, vol. 16, no. 1, pp. 61–102, 2011.

[50] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, "IPOG/IPOG-D: Efficient test generation for multi-way combinatorial testing," *Softw. Testing Verification Rel.*, vol. 18, no. 3, pp. 125–148, 2008.

[51] J. Petke, M. B. Cohen, M. Harman, and S. Yoo, "Efficiency and early fault detection with lower and higher strength combinatorial interaction testing," in *Proc. Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.*, Aug. 2013, pp. 26–36.

[52] F. Baader and T. Nipkow, *Term Rewriting and All That*. Cambridge, U.K.: Cambridge Univ. Press, 1998.

[53] M. Codish, J. Giesl, P. Schneider-Kamp, and R. Thiemann, "SAT solving for termination proofs with recursive path orders and dependency pairs," *J. Autom. Reasoning*, vol. 49, no. 1, pp. 53–93, 2012.

[54] C. Fuhs, "SAT-based termination analysis for Java bytecode with AProVE," (2011). [Online]. Available: http://www.dcs.bbk.ac.uk/~carsten/satcomp/AProVE11.pdf

[55] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco, CA, USA: Freeman, 1979.

[56] M. Järvisalo, P. Kaski, M. Koivisto, and J. H. Korhonen, "Finding efficient circuits for ensemble computation," in *Proc. 15th Int. Conf. Theory Appl. Satisfiability Testing*, 2012, pp. 369–382.

[57] M. Harman, P. McMinn, J. Souza, and S. Yoo, "Search based software engineering: Techniques, taxonomy, tutorial," in *Empirical Software Engineering and Verification: LASER 2009-2010*, B. Meyer and M. Nordio, Eds. Berlin, Germany: Springer, 2012, pp. 1–59.

[58] M. Gabel and Z. Su, "A study of the uniqueness of source code," in *Proc. 18th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2010, pp. 147–156.

[59] F. Wu, W. Weimer, M. Harman, Y. Jia, and J. Krinke, "Deep parameter optimisation," in *Proc. Genetic Evol. Comput. Conf.*, 2015, pp. 1375–1382.

[60] W. B. Langdon, B. Y. H. Lam, J. Petke, and M. Harman, "Improving CUDA DNA analysis software with genetic programming," in *Proc. Genetic Evol. Comput. Conf.*, 2015, pp. 1063–1070.

[61] R. Williams, C. P. Gomes, and B. Selman, "Backdoors to typical case complexity," in *Proc. 18th Int. Joint Conf. Artif. Intell.*, 2003, pp. 1173–1178.

[62] C. Ansótegui, J. Giráldez-Cru, and J. Levy, "The community structure of SAT formulas," in *Proc. 15th Int. Conf. Theory Appl. Satisfiability Testing*, 2012, pp. 410–423. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-31612-8_31

[63] Z. Newsham, V. Ganesh, S. Fischmeister, G. Audemard, and L. Simon, "Impact of community structure on SAT solver performance," in *Proc. 17th Int. Conf. Held Part Vienna Summer Logic Theory Appl. Satisfiability Testing*, 2014, pp. 252–268. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-09284-3_20

[64] E. T. Barr, Y. Brun, P. T. Devanbu, M. Harman, and F. Sarro, "The plastic surgery hypothesis," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2014, pp. 306–317.

[65] M. Harman, "Software engineering meets evolutionary computation," *IEEE Comput.*, vol. 44, no. 10, pp. 31–39, Oct. 2011.

[66] T. E. Colanzi, S. R. Vergilio, W. K. G. Assuncao, and A. Pozo, "Search based software engineering: Review and analysis of the field in Brazil," *J. Syst. Softw.*, vol. 86, no. 4, pp. 970–984, Apr. 2013.

[67] F. G. Freitas and J. T. Souza, "Ten years of search based software engineering: A bibliometric analysis," in *Proc. 3rd Int. Symp. Search Based Softw. Eng.*, Sep. 2011, pp. 18–32.

[68] M. Harman and B. F. Jones, "Search based software engineering," *Inf. Softw. Technol.*, vol. 43, no. 14, pp. 833–839, Dec. 2001.

[69] M. Harman, A. Mansouri, and Y. Zhang, "Search based software engineering: Trends, techniques and applications," *ACM Comput. Surveys*, vol. 45, no. 1, pp. 11:1–11:61, Nov. 2012.

[70] F. Ferrucci, M. Harman, and F. Sarro, "Search-based software project management," in *Software Project Management in a Changing World*. Berlin, Germany: Springer, 2014, pp. 373–399.

[71] Y. Zhang, A. Finkelstein, and M. Harman, "Search based requirements optimisation: Existing work and challenges," in *Proc. Int. Working Conf. Requirements Eng.: Found. Softw. Quality*, 2008, pp. 88–94.

[72] O. Räihä, "A survey on search–based software design," *Comput. Sci. Rev.*, vol. 4, no. 4, pp. 203–249, 2010.

[73] M. Harman, Y. Jia, J. Krinke, W. B. Langdon, J. Petke, and Y. Zhang, "Search based software engineering for software product line engineering: A survey and directions for future work," in *Proc. 18th Int. Softw. Product Line Conf.*, 2014, pp. 5–18.

[74] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege, "A systematic review of the application and empirical investigation of search-based test-case generation," *IEEE Trans. Softw. Eng.*, vol. 36, no. 6, pp. 742–762, Nov./Dec. 2010.

[75] P. McMinn, "Search-based software test data generation: A survey," *Softw. Testing Verification Rel.*, vol. 14, no. 2, pp. 105–156, Jun. 2004.

[76] M. D. Penta, "SBSE meets software maintenance: Achievements and open problems," in *Proc. 4th Int. Symp. Search Based Softw. Eng.*, 2012, pp. 27–28.

[77] U. Mansoor, M. Kessentini, B. Maxim, and K. Deb, "Multi-objective code-smells detection using good and bad design examples," *Softw. Quality J.*, vol. 25, no. 2, pp. 529–552, 2017.

[78] M. W. Mkaouer, M. Kessentini, S. Bechikh, K. Deb, and M. Ó. Cinnéide, "High dimensional search-based software engineering: Finding tradeoffs among 15 objectives for automating software refactoring using NSGA-III," in *Proc. Genetic Evol. Comput. Conf.*, 2014, pp. 1263–1270.

[79] A. Ouni, M. Kessentini, H. A. Sahraoui, K. Inoue, and K. Deb, "Multi-criteria code refactoring using search-based software engineering: An industrial case study," *ACM Trans. Softw. Eng. Methodology*, vol. 25, no. 3, pp. 23:1–23:53, 2016.

[80] M. Harman, "Why source code analysis and manipulation will always be important (keynote)," in *Proc. 10th IEEE Int. Work. Conf. Source Code Anal. Manipulation*, 2010, pp. 7–19.

[81] W. B. Langdon, M. Modat, J. Petke, and M. Harman, "Improving 3D medical image registration CUDA software with genetic programming," in *Proc. Genetic Evol. Comput. Conf.*, 2014, pp. 951–958. [Online]. Available: http://doi.acm.org/10.1145/2576768.2598244

[82] W. B. Langdon and M. Harman, "Genetically improved CUDA C ++ software," in *Proc. 17th Eur. Conf. Genetic Program.*, 2014, pp. 87–99.

[83] P. Sitthi-amorn, N. Modly, W. Weimer, and J. Lawrence, "Genetic programming for shader simplification," *ACM Trans. Graph.*, vol. 30, no. 6, pp. 152:1–152:11, 2011.

[84] W. B. Langdon, "Genetically improved software," in *Handbook of Genetic Programming Applications*, A. H. Gandomi, A. H. Alavi, and C. Ryan, Eds. Berlin, Germany: Springer, 2015, ch. 8, pp. 181–220. [Online]. Available: http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/langdon_2015_hbgpa.pdf

[85] E. Schulte, J. Dorn, S. Harding, S. Forrest, and W. Weimer, "Post-compiler software optimization for reducing energy," *SIGARCH Comput. Archit. News*, vol. 42, no. 1, pp. 639–652, Feb. 2014.

[86] M. Harman, Y. Jia, and W. B. Langdon, "Babel pidgin: SBSE can grow and graft entirely new functionality into a real world system," in *Proc. 6th Symp. Search Based Softw. Eng.*, 2014, pp. 247–252.

[87] W. B. Langdon and M. Harman, "Grow and graft a better CUDA pknotsrg for RNA pseudoknot free energy calculation," in *Proc. Genetic Evol. Comput. Conf.*, 2015, pp. 805–810. [Online]. Available: http://doi.acm.org/10.1145/2739482.2768418

[88] Y. Jia, M. Harman, W. B. Langdon, and A. Marginean, "Grow and serve: Growing Django citation services using SBSE," in *Proc. 7th Int. Symp. Search-Based Softw. Eng.*, 2015, pp. 269–275. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-22183-0_22

[89] E. T. Barr, M. Harman, Y. Jia, A. Marginean, and J. Petke, "Automated software transplantation," in *Proc. Int. Symp. Softw. Testing Anal.*, 2015, pp. 257–269.

[90] M. Schäfer, M. Sridharan, J. Dolby, and F. Tip, "Refactoring Java programs for flexible locking," in *Proc. 33rd Int. Conf. Softw. Eng.*, 2011, pp. 71–80.

[91] K. Ishizaki, S. Daijavad, and T. Nakatani, "Refactoring Java programs using concurrent libraries," in *Proc. 9th Workshop Parallel Distrib. Syst. Testing Anal. Debugging*, 2011, pp. 35–44.

[92] A. Ouni, R. G. Kula, M. Kessentini, T. Ishio, D. M. Germán, and K. Inoue, "Search-based software library recommendation using multi-objective optimization," *Inf. Softw. Technol.*, vol. 83, pp. 55–75, 2017.

[93] F. Thung, D. Lo, and J. L. Lawall, "Automated library recommendation," in *Proc. 20th Working Conf. Reverse Eng.*, 2013, pp. 182–191.

[94] D. Li, A. H. Tran, and W. G. J. Halfond, "Making web applications more energy efficient for OLED smartphones," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 527–538.

[95] I. L. Manotas-Gutiérrez, L. L. Pollock, and J. Clause, "SEEDS: A software engineer's energy-optimization decision support framework," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 503–514.

[96] J. Petke, W. B. Langdon, and M. Harman, "Applying genetic improvement to MiniSAT," in *Proc. 5th Int. Symp. Search Based Softw. Eng.*, 24–26 Aug. 2013, pp. 257–262.

[97] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The Oracle problem in software testing: A survey," *IEEE Trans. Softw. Eng.*, vol. 41, no. 5, pp. 507–525, May 2015.

Justyna Petke is a senior research associate in the Centre for Research on Evolution, Search and Testing (CREST), University College London. She has published articles on the applications of genetic improvement. Her paper work on GI won multiple awards. She is supported by the Dynamic Adaptive Automated Software Engineering grant from UK Engineering and Physical Sciences Research Council (EPSRC).

Mark Harman is professor of software engineering with UCL and an engineering manager with Facebook. He is widely known for work on source code analysis and testing and was instrumental in the founding of the field of search based software engineering (SBSE), an area of research to which this paper seeks to make a contribution. Since its inception in 2001, SBSE has rapidly grown to include more than 900 authors, from nearly 300 institutions. GGGP and DAASE projects partly support the presented work.

William B. Langdon is a professorial research fellow with UCL. He worked on distributed real time databases for control and monitoring of power stations in the Central Electricity Research Laboratories. He then joined Logica to work on distributed control of gas pipelines and later on computer and telecommunications networks. After returning to academia to gain a PhD in GP at UCL, he worked with the University of Birmingham, the CWI, UCL, Essex University, King's College, London and now for a third time at UCL.

Westley Weimer received the BA degree in computer science and mathematics from Cornell University and the MS and PhD degrees from the University of California, Berkeley. He is currently an associate professor with the University of Virginia. His main research interests include static and dynamic analyses to improve software quality, and fix defects.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.