HOMI: Searching Higher Order Mutants for Software Improvement

Fan Wu(✉), Mark Harman, Yue Jia, and Jens Krinke

Department of Computer Science, UCL,
Gower Street, London WC1E 6BT, UK
{fan.wu.12,mark.harman,yue.jia,j.krinke}@ucl.ac.uk

Abstract. This paper introduces HOMI, a Higher Order Mutation based approach for Genetic Improvement of software, in which the code modification granularity is finer than in previous work while scalability remains. HOMI applies the NSGAII algorithm to search for higher order mutants that improve the non-functional properties of a program while passing all its regression tests. Experimental results on four real-world C programs shows that up to 14.7 % improvement on time and 19.7 % on memory are found using only First Order Mutants. By combining these First Order Mutants, HOMI found further improvement in Higher Order Mutants, giving an 18.2 % improvement on the time performance while keeping the memory improvement. A further manual analysis suggests that 88 % of the mutation changes cannot be generated using line based 'plastic surgery' Genetic Improvement approaches.

1 Introduction

Optimising software for better performance such as speed and memory consumption can be demanding, especially when the resources in the running environment are limited. Manually optimising such non-functional properties while keeping or even improving the functional behaviour of softwasre is challenging. This becomes an even harder task if the properties considered are competing with each other [14]. Search-Based Software Engineering (SBSE) [13] has demonstrated many potential solutions, for example, to speed up software systems [21,28], or to reduce memory consumption [29] and energy usage [7].

Previous studies have applied different search-based techniques to automate the optimisation process [3,6,15,22]. However, scalability of these approaches remains a challenge. To scale up and optimise real world programs, recent studies use a so-called 'plastic surgery' Genetic Programming (GP) approach. To reduce the search space, it represents solutions as a list of edits to the subject program instead of the program itself [7,27]. Each sequence of edits consists of inserting, deleting or swapping pieces of code. To ensure scalability, this approach usually modifies programs at the 'line' level of granularity (the smallest atomic unit is a line of code). As a results, it is challenging for 'plastic surgery' to optimise subject programs in finer granularity.

Mutation Testing [9,16] is an effective testing technique to test software. It automatically inserts artificial faults in the programs under test, to create a set

F. Sarro and K. Deb (Eds.): SSBSE 2016, LNCS 9962, pp. 18–33, 2016.
DOI: 10.1007/978-3-319-47106-8_2

of faulty programs that are called 'mutants'. These mutants are used to assess the quality of tests, to provide testing criteria for generating new tests [11], and to fix software bugs [22]. More recently they have also been suggested as a means to perform sensitivity analysis [29] and to optimise software [19].

We introduce the HOMI approach to improve non-functional properties of software while preserving the functionality. HOMI utilises search-based higher order mutation testing [12] to effectively explore the search space of varying versions of a program. Like other previous Genetic Improvement (GI) work [7,21,27], HOMI relies on high-quality regression tests to check the functionality of the program. Given a program p and its regression tests T. HOMI generates two types of mutants that can be used for performance improvement. A **GI-FOM** is constructed by making a single syntactic change to p, which improves some non-functional properties of p while passing all the regression tests T. Having the same characteristics as GI-FOMs, a **GI-HOM** is constructed from the combination of GI-FOMs.

By combining with Mutation Testing techniques, we specifically utilise equivalent mutants which are expressly avoided by mutation testers where possible [25]. We implemented a prototype tool to realise the HOMI approach. The tool is designed to focus on two aspects of software runtime performance: execution time and memory consumption. Time and space are important qualities for most software, especially on portable devices or embedded systems where the runtime resources are limited. Moreover, these two qualities are usually competing with each other, yielding an interesting multi-objective solution space. Our tool produces a set of non-dominated GI-HOMs (thus forming a Pareto front). We evaluate our tool using four open source benchmarks. Since the tool requires no prior knowledge about the subjects, it can be easily applied to other programs.

The paper presents evidence that using Higher Order Mutation is an effective, easy to adopt way to improve existing programs. The experimental results suggest that equivalent First Order Mutants (FOMs) can improve the subject programs by 14.7 % on execution time or 19.7 % on memory consumption. Further results show that by searching for GI-HOMs, we can achieve up to 18.2 % time reduction on extreme cases. Our static analysis suggests that 88 % of the changes in GI-HOMs cannot be achieved by 'plastic surgery' based approaches. The contributions of the paper are as follows:

1. We introduce an automatic approach to improve programs via Higher Order Mutation, which explores program search space at a fine granularity while maintaining good scalability.
2. We evaluate our approach on four open source programs with different sizes. We report the results and demonstrate that our approach is able to reduce the execution time by up to 18.2 % or to save the memory consumption by up to 19.7 %.
3. The results of a manual analysis are reported to show that our approach works on a smaller granularity, such that 88 % of the changes found by our approach cannot be achieved by line based 'plastic surgery' approaches.

4. We also show evidence that it is possible to combine the HOMI approach with Deep-Parameter-optimisation approach to further improve the performance.

2 The HOMI Approach

We propose the HOMI approach, a higher order mutation based solution to GI. Figure 1 shows the overall architecture of the HOMI approach. Given a subject program with a set of regression tests, and some optimisation goals, HOMI applies SBSE to evolve a set of GI-HOMs that improve the properties of interest while passing all the regression tests. To explore the search space efficiently, we follow the current practice of GI in separating our approach into two stages [20]. In the first stage, we apply first order mutation to find locations in the program at which making changes will lead to significant impact on the optimisation goals. In the second stage, we apply a multi-objective search algorithm at these program locations to construct a Pareto front of GI-HOMs.

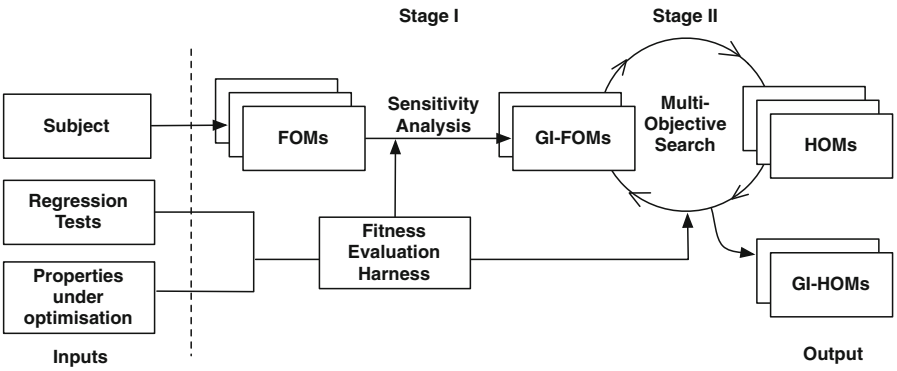


Fig. 1. The overall architecture of the HOMI approach.

2.1 Stage I: Sensitivity Analysis

Sensitivity analysis has been shown to be an effective way to reduce the search space in previous GI work [7,20,22]. Given a subject program under optimisation, some code pieces may have a greater impact on the properties of interest than others. Sensitivity analysis seeks to find small portions of code that have the greatest impact on the properties of interest. Thus, the subsequent optimisation can focus on a manageable amount of code, effectively reducing the search space. We use a first order mutation based sensitivity analysis approach to gather sensitivity information (See Sect. 2.3 for more details); this approach was introduced by Wu et al. [29]. We use this form of sensitivity analysis because it provides finer granularity than traditional statement or line-based sensitivity analysis.

As shown in Fig. 1, HOMI first generates a set of FOMs of the subject program and then evaluates them using a fitness evaluation harness. The evaluation harness is composed of regression tests and the measurement components for optimisation goals. It runs each FOM on all the tests and outputs the measurements of the optimisation goals as fitness values. After the fitness evaluation, HOMI removes FOMs that fail any regression tests and keeps only the survived ones. We do this because any mutants that pass all the regression tests are more likely to preserve the correctness of the subject. Finally, HOMI applies a non-dominated sorting [8] to rank all the survived FOMs by their fitness values.

The sensitivity analysis stage outputs a set of GI-FOMs. These FOMs are "potentially equivalent mutants" with respect to the regression test suites and have a positive impact on the properties of interest. We measure the sensitivity of code based on the FOMs' fitness values. A piece of code A is said to be more sensitive than another piece B, if a FOM generated from A dominates the FOM generated from B on the Pareto front. The range of a code piece can be measured at different granularity levels by aggregating the results of FOMs, such as at the syntactic symbol, the statement level, or the nesting code block level. The GI-FOMs generated and their sensitive information are passed to the next search stage as inputs.

2.2 Stage II: Searching for GI-HOMs

In the second stage, HOMI applies a multi-objective algorithm to search for a set of improved versions of the original program in the form of HOMs. We use an integer vector to represent HOMs, which is a commonly used data representation in search-based Higher Order Mutation Testing [18]. Each integer value in the vector encodes whether a mutable symbol is mutated and how it is mutated. For example, given a mutant generated from the arithmetic operator '`+`', a negative integer value means it is not mutated while the integer 0, 1, 2, 3 indicate that the code is mutated to '`-`', '`*`', '`/`', '`%`' respectively. In this way, each FOM is represented as a vector with only one non-negative number and HOMs can be easily constructed by the standard crossover and mutation search operators.

The algorithm takes the GI-FOMs as input and repeatedly evolves HOMs that inherit the strengths of the GI-FOMs from which they are constructed and yield better performance than any GI-FOMs alone. The fitness function that guides the search is defined as the sum of the measurement of each optimisation property over a given test suite. Given a set of N optimisation goals, for each mutant M, the fitness function $f_n(M)$ for the nth optimisation goal is formulated as follow:

$$\textit{Minimisation} \quad f_n(M) = \begin{cases} \sum C_i(M) & \text{if } M \text{ passes all test cases} \\ C_{\text{MAX}} & \text{if } M \text{ fails any test case} \end{cases}$$

The fitness function is a minimisation function where $C_i(M)$ is the measurement of the optimisation goal n when executing the test i. If the mutant M fails any regression tests, we consider it as a bad candidate and assign it with the

worst fitness values C_{MAX}. The algorithm produces a Pareto front of GI-HOMs. Each HOM on the front represents a modified version of the original program that passes all the regression tests while no property of interest can be further improved without compromising at least one other optimising goal.

2.3 Implementation

We implemented a prototype tool to realise the HOMI approach. The HOMI tool is designed to optimise two non-functional properties (running time and memory consumption) for C programs. In the fitness evaluation harness, we use *Glibc*'s *wait* system calls to gather the CPU time, and we instrument the memory management library to measure the 'high-water' mark of the memory consumption. We choose to measure virtual instead of physical memory consumption because the physical memory consumption is non-deterministic. This means it depends on the workload of the machine. By contrast, the virtual memory used is always an upper bound of the physical memory actually used.

HOMI uses the open source C mutation testing tool, Milu [17] to generate mutants. We chose Milu because it features search-based higher order mutation and can be used as an external mutant generator. By default, Milu supports only the traditional C mutation operators [1]. As memory consumption is one of the optimisation goals, we extended the original version of Milu to support Memory Mutation Operators proposed by Nanavati et al. [23]. Table 1 lists the Mutation Operators used in HOMI and their brief descriptions. During the search stage, HOMI transforms the internal integer vector representation of the candidate HOM to the data format recognisable by Milu, then invokes Milu to generate the HOM.

Table 1. Mutation Operators used by HOMI

Category	Name	Description
Selective Operators Mutation	ABS	Change an expression `expr` to `ABS(expr)` or `-ABS(expr)`
	OAAN	Change between `+, -, *, /, %`
	OLLN	Change between `&&, ||`
	ORRN	Change between >, >=, <, <=, ==
	OIDO	Change between ++x, −−x, x++, x−−
	CRCR	Change a constant `c` to `0, 1, -1, c+1, c-1, c*2, c/2`
Memory Mutation Operators	REC2M	Replace `malloc()` with `calloc()`
	RMNA	Remove `NULL` assignment
	REDAWN	Replace memory allocation calls to `NULL`
	REDAWZ	Replace allocation size with `0`
	RESOTPE	Replace `sizeof(T)` with `sizeof(*T)`
	REMSOTP	Replace `sizeof(*T)` with `sizeof(T)`
	REM2A	Replace `malloc()` with `alloca()`
	REC2A	Replace `calloc()` with `alloca()`
	RMFS	Remove `free()` statement

The HOMI tool employs a customised NSGA-II [8] to evolve GI-HOMs. During the search process, HOMI maintains a population of candidate HOMs. For each generation, the uniform crossover and mutation are performed to parent HOMs, generating offspring HOMs that are later evaluated using the fitness functions mentioned. A tournament selection is then performed to form the next generation. This process is repeated until a given budget of evaluation times is reached. Finally HOMI will generate a set of non-dominating GI-HOMs that perform better than the original program on time and/or memory consumption.

3 Empirical Study

This section first discusses the research questions we address in our empirical evaluation of the HOMI tool, followed by an explanation of the chosen subjects, tests and experiment settings.

3.1 Research Questions

Since the HOMI approach generates GI-HOMs from the combination of FOMs, a natural first question to ask is 'whether existing FOMs can be used to improve software'. This motivates our first research question.

RQ1: Can GI-FOMs improve program performance while passing all of its regression tests?

To answer this question, we run HOMI for sensitivity analysis only and report how much running time and memory can be saved by GI-FOMs. Of course, the answer also depends on the quality of the regression tests. All the tests used in our evaluation are regression tests generated by developers for real world systems. However, they may still not be sufficient to reveal the faults introduced by mutation. To make our experiment more rigorous and efficient, we carried out a pre-analysis in our evaluation. We analyse the function coverage of each subject using the GNU application *Gcov* and HOMI is set only to mutate the functions that are covered by regression tests.

RQ2: How much improvement can be achieved by GI-HOM in comparison with GI-FOMs?

If GI-FOMs alone can improve performance, we expect that GI-HOMs will inherit some strengths from the GI-FOMs and improve the performance further. To answer this question, we use HOMI to generate a GI-HOM Pareto front and investigate whether the GI-FOM solutions generated are on the Pareto front. Furthermore, it is interesting to see whether the new memory mutation operators help to improve the performance. This motivates our sub-question which studies the effect on mutation operators used.

RQ 2.1 How does the improvement achieved by applying the traditional mutation operators only compare to applying both of the traditional and memory mutation operators?

We answer this question by comparing the HyperVolume quality indicator of the Pareto fronts generated from HOMI using both sets of mutation operators. Given a Pareto front A and a reference Pareto front R, HyperVolume is the volume of objective space dominated by solutions in A. To take into account the stochastic nature of the search algorithms, we repeat both experiments 30 times. We use the non-parametric Mann-Whitney-Wilcoxon-signed rank tests to assess the statistical significance of the HyperVolume and untransformed [24] Vargha-Delaney effect size to further assess the magnitude of the differences [2].

RQ3: Can 'plastic surgery' GP based GI approach find edit sequences to construct the GI-HOMs found by HOMI?

We ask this question because we want to understand whether the granularity of mutation changes can be produced by the 'plastic surgery' GP approach. The 'plastic surgery' GP approach is a popular GI approach which searches for a list of edits from the existing source code. Typical changes generated by the GP approach are movements or replacements of different lines of code [20,26]. To answer this question, we carried out a sanity-check experiment manually using all the GI-HOMs found. For each GI-HOM, we search the entire program to see if the mutated statement exists in the program. If it does, the GI-HOM can be constructed by the patches generated from the GP approach easily. Otherwise, we consider the line/statement based 'plastic surgery' GP might not able to generate the GI-HOM directly.

RQ4 Can HOMI be combined with Deep Parameter Optimisation to achieve further improvement?

Finally, we want to investigate whether the HOMI approach can be combined with other types of GI techniques. Deep Parameter Optimisation is one of the state-of-the-art parameter tuning based GI techniques. It seeks to optimise library code used instead of the source code of the subjects [29]. We answer this research question by evaluating the GI-HOMs after linking them to Deep-Parameter-optimised libraries, then comparing them with their performance before the linking, and with the performance of the original program after linking to Deep-Parameter-optimised libraries.

3.2 Subject Programs and Tests

We optimise four subjects in our evaluation. Table 2 lists the subjects and their brief description. All tests used are regression tests, deemed to be useful and practical by their developers. *Espresso* is a fast application for simplifying complex digital electronic gate circuits. *Gawk* is the GNU *awk* implementation for string processing. *Flex* is a tool for generating scanners, programs which recognise

Table 2. Subject programs

Name	LoC	# of Tests	Description
espresso	13,256	19	Digital circuit simplification
gawk	45,241	334	String processing
flex	9,597	62	Fast lexical analyzer generator
sed	5,720	362	Special file editor

lexical patterns in text, and *sed* is an editor that automatically modifies files given a set of rules. We use the *espresso* version as well as its test cases from *DieHard* project [5]. Version 4.1.0 of *gawk* is used in this work. The source code and the test cases can be found in the GNU archives. We obtain the last two programs and corresponding test suites from the SIR repository [10].

3.3 Search Settings

In the sensitivity analysis stage, we pick the top 10 % most sensitive locations of the GI-FOMs and only search for GI-HOMs from these locations. We use a relative ratio instead of an absolute number because the search space can be adapted to the size of the subject. The choice of 10 % is based on our observation that the locations in the first 10 % are usually much more sensitive than the remaining locations since sensitivity seems to follow a power law, according to our informal observation. However, the ratio can be easily adapted as a parameter to our approach accordingly.

We repeat all HOMI experiments 30 times to cope with the non-deterministic nature of NSGAII and to facilitate inferential statistical analysis. The NSGAII performs a tournament selection of size 2 and uniform crossover with a probability of 0.8. There are 50 HOMs in each generation and the algorithm stops at 100th generation. These numbers were chosen after initial calibration experimentation to determine suitable parameters for our search process. All of the experiments are carried out on a desktop machine with a quad-core CPU and 7.7 GB RAM running 64-bit Ubuntu version 14.04. *Gcc* 4.8.4 with optimisation option -O3 was used to compile all the mutants. The source of this project is publicly available at https://github.com/FanWuUCL/HOMI.

4 Results and Discussion

4.1 Improvement by GI-FOMs

We begin by looking at the time and memory performance of the GI-FOMs generated from the sensitivity analysis stage to answer the RQ1. We calculated the improvement of GI-FOMs relative to the original program, and reported them in Columns 2 and 5 in Table 3. These values are averaged from 10 repeated evaluations. By applying selective and memory mutation operators to generate

FOMs, we found the improved versions of all four subjects, both on time and memory performance. More specifically, the improvement ranges from 0.9 % to 14.7 % on time and from 0.5 % to 19.7 % on memory performance. However, there might be a large gap between the memory and time improvement for some subject, for example, the GI-FOM of *sed* can run up to 14.7 % faster to only save 0.5 % memory. We conclude that even with the simplest changes introduced by first order mutation, the HOMI approach is able to improve the execution time and memory consumption.

4.2 Improvement by GI-HOMs

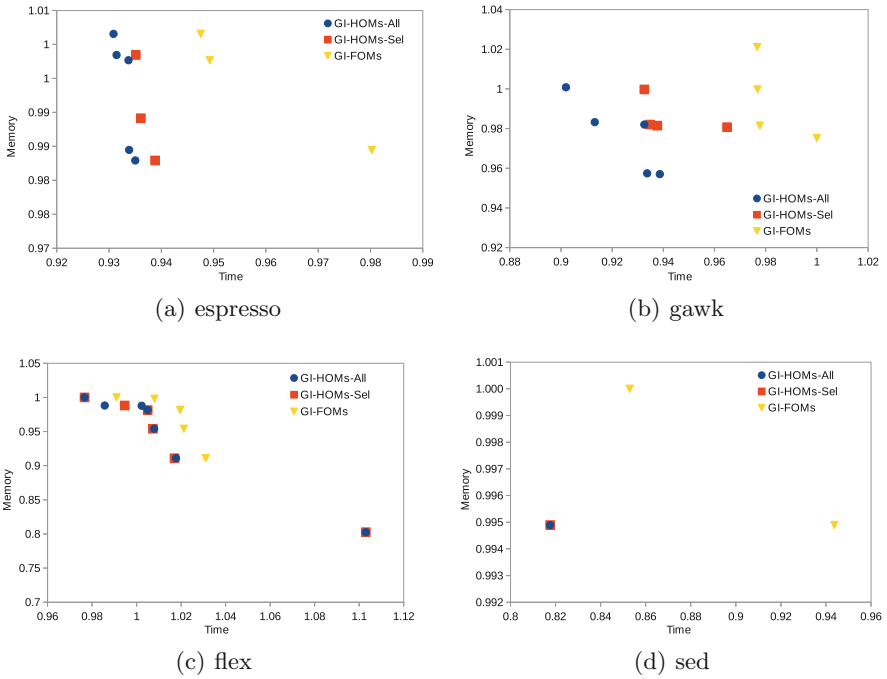
We now turn to the improvement found by GI-HOMs. Since improvement was found on GI-FOMs, it is interesting to investigate whether we can improve the performance further by combining them to form GI-HOMs. We applied NSGA-II [8] to search for better performance in HOMs using Selective Mutation Operators (GI-HOMs-Sel) and using both Selective and Memory Mutation Operators (GI-HOMs-All) respectively. Each experiment was repeated for 30 times and the best time/memory performance found for each subject is reported in Table 3.

Table 3. Improvement on time and memory by GI-FOMs and GI-HOMs. GI-HOMs-Sel are found using only Selective Mutation Operators while GI-HOMs-All and GI-FOMs are found using both Selective and Memory Mutation Operators

Subject	Time (%)			Memory (%)		
	GI-FOMs	GI-HOMs-Sel	GI-HOMs-All	GI-FOMs	GI-HOMs-Sel	GI-HOMs-All
espresso	5.2	6.5	**6.9**	1.6	**1.7**	**1.7**
gawk	2.3	6.7	**9.8**	2.5	1.9	**4.3**
flex	0.9	**2.3**	**2.3**	**19.7**	**19.7**	**19.7**
sed	14.7	**18.2**	**18.2**	**0.5**	**0.5**	**0.5**

The results of GI-HOMs-Sel are reported in Columns 3 and 6, and those of GI-HOMs-All are reported in Columns 4 and 7. We immediately observe that GI-HOMs achieve greater improvement than GI-FOMs on execution time for all subjects, also on memory consumption for two out of four subjects. The greatest time improvement found by GI-HOMs can be promoted to 18.2 %, while the improvement can be up to four times better (on *gawk*) than the improvement yielded from GI-FOMs. We also observe one case *gawk*), on which the GI-HOMs-Sel achieve less memory improvement compared with GI-FOMs, because they are lack of some memory-related changes that can only be achieved by Memory Mutation Operators.

We combine the results of 30 runs for each experiment and plot the Pareto fronts of GI-HOMs using all Mutation Operators, GI-HOMs using Selective Mutation Operators and GI-FOMs in Fig. 2. In the figure, time (x-axis) and memory (y-axis) are both normalised to the original performance. On all four



(a) espresso (b) gawk

(c) flex (d) sed

Fig. 2. Pareto fronts of GI-HOMs and GI-FOMs for each subject. Lower and lefthand solutions dominate high and righthand solutions.

subjects, we can see there is always an improvement from GI-FOMs to GI-HOMs, while the differences between GI-HOMs-Sel and GI-HOMs-All are less clear. To statistically demonstrate the difference, we calculated the HyperVolume [30] of the Pareto fronts of GI-HOMs-All and GI-HOMs-Sel over 30 runs, and applied Mann-Whitney-Wilcoxon U-test on the HyperVolume metric for these Pareto fronts. For subject *espresso* and *gawk*, the difference between HOMs (all) and HOMs (Selective) are significant ($p < 0.01$) with a large effect size ($A_{12} > 0.9$), while for the other two subjects, the difference is not significant.

In summary, the answer to RQ2 is that GI-HOMs can improve the time performance by up to 18.2 % or the memory performance by up to 19.7 %, compared with 14.7 % and 19.7 % in GI-FOMs. For the GI-HOMs using Selective Mutation Operators only, we found the same upper bound of the improvement, but only achieved sub-optimal solutions on two subjects. By including Memory Mutation Operators, further improvement on these subjects were found. Therefore, we can conclude that Memory Mutation Operators provide further improvement potentials for both time and memory optimisation.

4.3 HOMI vs 'Plastic Surgery' GP Based GI

This RQ investigates whether the granularity of mutation changes can also be generated by the 'plastic surgery' GP approach. To answer this question,

we investigated the GI-HOMs found in all experiments and manually reproduce them following the evolution rules used in the line/statement based 'plastic surgery' GP approach [20,26]. All together HOMI found 273 mutations in the improved GI-HOMs across all subjects. We first applied a simple hill-climbing algorithm to clean up the mutations that do not contribute to the improvement. This step narrowed the number of mutations down to 141. In total, there are 108 unique mutations identified (the same mutations may be found in several GI-HOMs).

For each of the unique mutation changes, we search the entire program to see if the mutated line/statement exists in the program. Because the typical changes generated by the 'plastic surgery' GP approach are movements or replacements of different lines of code [20,26], if a mutation does not appear somewhere else in the original source code, it cannot be generated directly from this form of GP approaches. The result shows that 95 (88 %) out of 108 mutations cannot be found in the original source code. Therefore, the answer to RQ3 is, there are 108 unique mutational changes found in the GI-HOMs, 88 % of which cannot be generated from the line-based 'plastic surgery' GP approach directly.

4.4 HOMI Combines with Deep Parameter Optimisation

In the last Research Question, we want to understand whether the improvement can be preserved or even promoted if we combine GI-HOMs with Deep-Parameter-optimised memory management library. To answer this question, we obtained a set of memory allocation libraries that were optimised for the time and memory performance for each subject from the authors of the Deep Parameter Optimisation work. We created four new optimised version of each subject by linking the most time/memory-saving GI-HOMs and libraries in pairwise.

The results are reported in Table 4. In the table, rows represent HOMI-improved programs and columns represent Deep-Parameter-optimised libraries, where 'Original' indicates the original program or library, 'T' indicates it is the most time-saving ones and 'M' indicates the most memory-saving ones. All of the numbers are the improvement in percentage compared with the original version. If in a combination (that does not involve the Original program/library), the time/memory performance is not worse than that of any of the 'ingredient' program/library, it is highlighted in bold font. On the other hand, all the underlined performances are the ones that are worse than both of the 'ingredient' program and library. For subject *sed*, there is only one GI-HOM on the Pareto front, thus it is both the most time and memory-saving program.

We observe that there are 10 out of 28 cases (bold numbers) when combining GI-HOMs with the Deep-Parameter-optimised library, the performance is at least the same as the best performance of the GI-HOM or library it is combined from, and is strictly better in four cases. However, there are three cases (underlined) that the combination makes their performance worse. In most of the cases, the performance lies between the performance of the GI-HOM and the library that it is combined from. In one extreme case (*flex*), we found that the most memory-saving library breaks the functionality of HOMs (indicated by '-Inf' in

Table 4. HOMI combines with Deep Parameter Optimisation. Each cell reports the time improvement followed by memory improvement in percentage. 'T' or 'M' indicates it is most time-saving or memory-saving GI-HOM/optimised library.

		Memory Management Library					Memory Management Library		
		Original	Deep(T)	Deep(M)			Original	Deep(T)	Deep(M)
espresso	Original	0/0	0.8/0.1	0.7/0.2	*gawk*	Original	0/0	5.4/1.6	-0.2/2.3
	GI-HOM(T)	6.9/-0.2	4.8/**0.1**	4.7/**0.2**		GI-HOM(T)	9.8/-0.1	5.6/**1.6**	5.4/**2.3**
	GI-HOM(M)	6.5/1.7	4.7/**1.8**	**6.7**/**1.7**		GI-HOM(M)	6.1/4.3	4.1/**5.8**	4.8/**5.5**
flex	Original	0/0	15.7/-2.6	-1.1/0.6	*sed*	Original	0/0	7.9/-1208	5.6/2.0
	GI-HOM(T)	2.3/0	14.4/-2.6	-Inf/-Inf		GI-HOM(TM)	18.2/0.5	5.8/-1208	4.1/0.9
	GI-HOM(M)	-10.3/19.7	-3.5/**19.7**	-Inf/-Inf					

the table). Therefore, the answer to RQ4 is, when combining the HOMI approach with Deep Parameter optimisation, the GI-HOM programs can be either improved or jeopardised. This result motivates a future study that searches and optimises HOMI and Deep Parameters altogether.

5 Threats to Validity

We discuss the threats to validity in this sections, where the threats to internal validity are discussed in Sect. 5.1 and those to external validity are discussed in Sect. 5.2.

5.1 Internal Validity

We used the regression tests that come with the subjects to evaluate the correctness of mutants. All the subject programs used in this paper were well tested in established works, and their tests used are regression tests, deemed to be useful and practical by their developers. However, passing the regression tests does not necessarily mean the semantics of the mutant is the same as the original program. This may pose a threat to the correctness of the GI-HOMs. To mitigated this threat, we set HOMI to apply mutation changes at the code that is covered by the regression tests.

After the sensitivity information is collected, we focus on 10 % most sensitive locations only. This is based on an assumption that less sensitive code is less likely to affect the performance of the program. However, there are still chances that the interactions between multiple less sensitive code may lead to some significant improvement. This possible synergy, if there is any, requires a much larger search space, thus, will make the approach much less scalable. To make the HOMI approach scalable, we confine the search on the most sensitive locations, making the search more effective. Furthermore, we make the ratio of sensitive locations a parameter of our approach, such that it can be adapted to trade between exploration and exploitation.

Another threat to validity comes from the measurement of time and memory performance. We applied the measurement approach proposed by Wu et al. [29]. To make the measurement accurate, we use CPU time and use the mean of 10 measurements to minimise the noise. For memory consumption, we instrument the memory management library to calculate the exact use of virtual memory. Therefore, the measurement noise is minimised.

5.2 External Validity

The approach can be easily applied to other subjects, but the conclusion may not generalise to larger scale systems. We use four subjects with varying sizes from 5,000 to 45,000 lines of code, and the results are consistent across all subjects. Therefore, we have confidence that the results may likely be generalised to larger scale systems, and the threat is thereby ameliorated.

We adopt Memory Mutation Operators in our approach because we are interested in time and memory performance. However, the same set of Mutation Operators does not necessarily lead to similar results when other software qualities are concerned. Since the selection of Mutation Operators is independent of the other parts of the approach, the choices of Mutation Operators can be easily adapted accordingly, thereby minimising this threat.

6 Related Work

One of the closely related GI work is the 'plastic surgery' GP approach proposed by Langdon et al. [20,21]. Their approach searches for sequences of edits at the granularity of statements. Due to the operations of statement swapping, the optimised code is hard for human developers to understand. Our approach uses simple syntactic mutations to improve the code, therefore the structure of the original code is always preserved. Since mutations can happen at the expression level, our approach works on a finer granularity.

Deep Parameter Optimisation is a similar work that also used First Order Mutants for sensitivity analysis [29]. After sensitivity analysis, they inserted and exposed additional parameters at most sensitive locations, which were later optimised using multi-objective search algorithms. Our approach follows a simpler procedure, combining FOMs to achieve better performance. Furthermore, our approach searches for code changes that happen at much more locations at the same time than the Deep Parameter approach does, while the scalability of the approach remains.

Other Genetic Improvement (GI) works also consider other qualities of software, such the correctness [22] or energy consumption [7]. In our work, execution time and memory consumption are concerned not only because they are important qualities to many benchmark programs, but also because unlike other software qualities, they are known to compete with each other. Therefore, it is interesting to study the trade-off between these two software qualities.

Barr et al. investigated the plastic surgery hypothesis: the content of new code can often be assembled from existing code base [4]. They found that 43 % of the code changes could be composed of the same software program at the line level. While they focused on human-written patches, we investigated how likely a machine-generated mutational change can be found somewhere else in the source code. Our result suggests that 88 % of those mutational changes cannot be composed of the same code base at the line level.

7 Conclusion

In this paper, we have introduced, HOMI, a search-based higher order mutation approach to GI. HOMI uses mutation operators to automatically modify subject programs at a finer granularity. Using a multi-objective search algorithm, HOMI found GI-HOMs that improve subject programs by 18.2 % on time performance or 19.7 % on memory consumption without breaking any regression tests. In our empirical study, we also find that by including Memory Mutation Operators, HOMI can find GI-HOMs that achieve better performance than using just traditional Selective Mutation Operators on two subjects. Furthermore, we find that 88 % of the mutational changes in our GI-HOMs cannot be generated from the currently widely-used line based 'plastic surgery' GP approach. Finally, by combining GI-HOMs with Deep-Parameter-optimised memory management libraries, we found further improvement than GI-HOMs or optimised libraries alone could achieve, which motivates a future research direction that searches and optimises GI-HOMs and Deep Parameters altogether.

References

1. Agrawal, H., DeMillo, R.A., Hathaway, B., Hsu, W., Hsu, W., Krauser, E.W., Martin, R.J., Mathur, A.P., Spafford, E.: Design of mutant operators for the C programming language. techreport SERC-TR-41-P, Purdue University, West Lafayette, Indiana, March 1989
2. Arcuri, A., Briand, L.: A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. Softw. Test. Verif. Reliab. **24**(3), 219–250 (2014)
3. Arcuri, A., Fraser, G.: On parameter tuning in search based software engineering. In: Cohen, M.B., Ó Cinnéide, M. (eds.) SSBSE 2011. LNCS, vol. 6956, pp. 33–47. Springer, Heidelberg (2011)
4. Barr, E.T., Brun, Y., Devanbu, P., Harman, M., Sarro, F.: The plastic surgery hypothesis. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014, pp. 306–317. ACM, New York (2014)
5. Berger, E.D., Zorn, B.G.: Diehard: probabilistic memory safety for unsafe languages. In: Programming Language Design and Implementation, PLDI 2006 (2006)
6. Brake, N., Cordy, J.R., Dan, Y.E., Litoiu, M., Popes U, V.: Automating discovery of software tuning parameters. In: Workshop on Software Engineering for Adaptive and Self-managing Systems, SEAMS 2008 (2008)

7. Bruce, B.R., Petke, J., Harman, M.: Reducing energy consumption using genetic improvement. In: Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation, GECCO 2015, pp. 1327–1334. ACM, New York (2015)
8. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Trans. Evol. Comput. **6**(2) (2002)
9. DeMillo, R.A., Lipton, R.J., Sayward, F.G.: Hints on test data selection: Help for the practicing programmer. Computer (4), 34–41 (1978)
10. Do, H., Elbaum, S., Rothermel, G.: Supporting controlled experimentation with testing techniques: an infrastructure and its potential impact. Empirical Softw. Eng. **10**(4), 405–435 (2005)
11. Harman, M., Jia, Y., Langdon, W.B.: Strong higher order mutation-based test data generation. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE 2011, pp. 212–222 (2011)
12. Harman, M., Jia, Y., Reales Mateo, P., Polo, M.: Angels and monsters: an empirical investigation of potential test effectiveness and efficiency improvement from strongly subsuming higher order mutation. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ASE 2014, pp. 397–408. ACM, New York (2014)
13. Harman, M., Jones, B.F.: Search-based software engineering. Inf. Softw. Technol. **43**(14), 833–839 (2001)
14. Harman, M., Langdon, W.B., Jia, Y., White, D.R., Arcuri, A., Clark, J.A.: The gismoe challenge: Constructing the pareto program surface using genetic programming to find better programs (keynote paper). In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, ASE 2012, pp. 1–14. ACM, New York (2012)
15. Hutter, F., Hoos, H.H., Leyton-Brown, K., Stützle, T.: ParamILS: an automatic algorithm configuration framework. J. Artif. Intell. Res. **36**(1), 267–306 (2009)
16. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Trans. Softw. Eng. **37**(5), 649–678 (2011)
17. Jia, Y., Harman, M.: MILU: a customizable, runtime-optimized higher order mutation testing tool for the full C language. In: Proceedings of the TAIC PART 2008, Windsor, UK, pp. 94–98, 29–31 August 2008
18. Jia, Y., Harman, M.: Higher order mutation testing. Inf. Softw. Technol. **51**(10), 1379–1393 (2009). Source Code Analysis and Manipulation
19. Jia, Y., Wu, F., Harman, M., Krinke, J.: Genetic improvement using higher order mutation. In: Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation, GECCO Companion 2015, pp. 803–804. ACM, New York (2015)
20. Langdon, W., Harman, M.: Optimizing existing software with genetic programming. IEEE Trans. Evol. Comput. **19**(1), 118–135 (2015)
21. Langdon, W.B., Modat, M., Petke, J., Harman, M.: Improving 3D medical image registration CUDA software with genetic programming. In: Conference on Genetic and Evolutionary Computation, GECCO 2014 (2014)
22. Le Goues, C., Nguyen, T., Forrest, S., Weimer, W.: Genprog: a generic method for automatic software repair. IEEE Trans. Softw. Eng. **38**(1), 54–72 (2012)
23. Nanavati, J., Wu, F., Harman, M., Jia, Y., Krinke, J.: Mutation testing of memory-related operators. In: 2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 1–10, April 2015

24. Neumann, G., Harman, M., Poulding, S.: Transformed Vargha-Delaney effect size. In: Barros, M., Labiche, Y. (eds.) SSBSE 2015. LNCS, vol. 9275, pp. 318–324. Springer, Heidelberg (2015)
25. Papadakis, M., Jia, Y., Harman, M., Traon, Y.L.: Trivial compiler equivalence: a large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol. 1, pp. 936–946, May 2015
26. Petke, J., Harman, M., Langdon, W.B., Weimer, W.: Using genetic improvement and code transplants to specialise a C++ program to a problem class. In: Nicolau, M., Krawiec, K., Heywood, M.I., Castelli, M., García-Sánchez, P., Merelo, J.J., Rivas Santos, V.M., Sim, K. (eds.) EuroGP 2014. LNCS, vol. 8599, pp. 137–149. Springer, Heidelberg (2014)
27. Petke, J., Langdon, W.B., Harman, M.: Applying genetic improvement to MiniSAT. In: Ruhe, G., Zhang, Y. (eds.) SSBSE 2013. LNCS, vol. 8084, pp. 257–262. Springer, Heidelberg (2013)
28. White, D.R., Arcuri, A., Clark, J.A.: Evolutionary improvement of programs. IEEE Trans. Evol. Comput. **15**(4), 515–538 (2011)
29. Wu, F., Weimer, W., Harman, M., Jia, Y., Krinke, J.: Deep parameter optimisation. In: Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation, GECCO 2015, pp. 1375–1382. ACM, New York (2015)
30. Zitzler, E., Thiele, L.: Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach. IEEE Trans. Evol. Comput. **3**(4), 257–271 (1999)

http://www.springer.com/978-3-319-47105-1

Search Based Software Engineering
8th International Symposium, SSBSE 2016, Raleigh, NC, USA, October 8-10, 2016, Proceedings
Sarro, F.; Deb, K. (Eds.)
2016, XXI, 318 p. 71 illus., Softcover
ISBN: 978-3-319-47105-1