SSBSE-2016 (Challenge track) Federica Sarro and Kalyanmoy Deb *Eds.*, LNCS 9962, Raleigh, North Carolina, USA, 8-10 Oct. Springer. *Preprint*

API-Constrained Genetic Improvement

William B. Langdon, David R. White, Mark Harman, Yue Jia, Justyna Petke

University College London, CREST, UK

Abstract. ACGI respects the Application Programming Interface whilst using genetic programming to optimise the implementation of the API. It reduces the scope for improvement but it may smooth the path to GI acceptance because the programmer's code remains unaffected; only library code is modified. We applied ACGI to C++ software for the state-of-the-art OpenCV SEEDS superPixels image segmentation algorithm, obtaining a speed-up of up to 13.2% (±1.3%) to the $50K Challenge winner announced at CVPR 2015.

1 Introduction and Background

Genetic improvement uses computational search to find improved versions of existing software systems [8,6,11,19]. It usually does this by searching for a set of edits that are performed on the software system to be improved, such that the desired functional behaviour of the original is retained, while some functional [10,5] and/or non-functional [15,11] aspects are improved. There has been a recent upsurge of activity in this area, with results demonstrating that genetic improvement is able to improve many different properties of systems, including dynamic memory use [20], speed of execution [9,17] and energy consumption [1,14], as well as augmenting and fixing broken functionality [10,5].

One of the advantages of genetic improvement is that it uses unconstrained modifications to software systems, more akin to genetic programming [13], than traditional program transformation. As a result, the programmers' original version of the system, although improved, is also syntactically (and possibly semantically [9,15]) altered, making it less familiar to the programmer than the original. This lack of familiarity may pose a barrier to acceptance of genetically improved programs, and adoption of genetic improvement as a technique; developers may be concerned about ongoing maintenance and comprehension of the genetically improved program.

Ultimately, these concerns may be overcome by the advantages offered by genetic improvement: that which we currently regard as source code may, in future, become 'the new object code', to be manipulated freely by genetic improvement [6]. However, even if this vision were to be realised, there will remain a necessary transition period, during which we will need to support a 'mixed economy of software systems'. Systems, part produced by machine and part produced by humans, will have to co-exist, symbiotically and seamlessly. This raises the fundamental question for genetic improvement of determining the best separation of concerns between human and machine: how they might collaboratively arrive at improved software systems that are acceptable to human developers?

We propose API-Constrained Genetic Improvement (ACGI), as a first attempt to identify such a suitable separation of concerns. The key insight underlying ACGI is that human programmers are *already* generally prepared to accept third-party software in the form of library code, accessed through API calls. Typical criteria for library code acceptance revolve around the performance of the library functions, and demonstration of acceptable behaviour with respect to a suite of test cases; exactly the criteria that are automatically and inherently assessed during the genetic improvement process. Using ACGI, we constrain genetic improvement to manipulate only the library's source code, leaving the API and application code unmodified.

Although library functions are inherently designed to be general solutions, the underlying implementation does not have to be the same for all client applications. Instead we suggest libraries offer opportunities for specialisation. With potentially multiple implementations, each tailored to the expected usage of the library by one or more applications. AGCI, we hope, can tailor library functions to each particular client application, providing evidence for improved performance and adequate testing.

In the next two sections we apply GI to just the C++ source code which implements the SEEDS picture segmentation [18]. This implementation won the State of the Art Vision Challenge (`http://code.opencv.org/projects/opencv/wiki/VisionChallenge`) last year at the 28$^{\text{th}}$ IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2015) and was subsequently incorporated into the Open Source Computer Vision (OpenCV) library. Just acting on this source, using real run time on a real computer for fitness, GI was able to find an almost identical class which was on average more than 13% faster on the images used in the State of the Art Challenge. (These images are 700 by 1000 full colour. None of them were used in training by our GI.)

2 Applying ACGI to OpenCV Image Segmentation

We used the new ACGI framework on the OpenCV C++ source code of SEEDS Superpixels. To identify the library methods used, we first profiled a simple client application of the SuperPixels library using valgrind. This highlighted the updatePixels() method of the `SuperpixelSEEDSImpl` class. Then we used ACGI (see Table 1) to apply mutations to just updatePixels() and fellow methods called by it. (I.e. update(), addPixel(), deletePixel(), probability(), threebyfour() fourbythree() and updateLabels(). In total 319 lines of code.)

3 Results

3.1 Best of First Generation

In the first generation (left Figure 1) all but five mutants compiled. (These five failed to compile due to a bug in the new swap mutation's use of scoping rules). Eight failed run time array bound checks. Four were aborted by the CPU limit of 5 seconds (all due to deleting the iteration increment part of `for` loops). 65 ran and terminated ok but at least one pixel (of 7 990 272) was different from the value calculated by the original code. Leaving 18 cases where the code was

Table 1: Evolve faster than state-of-the-art superPixel OpenCV segmentation

Representation:	list replacements, deletions, insertions and swaps (via BNF grammar)
Fitness:	Compile (gcc 4.8.5) modified code. Compare its segmentation of 2448 by 3264 colour training image with segmentation given by original code. If identical, fitness is nanoseconds to run SuperpixelSEEDS::iterate(pic,4) else mutant is killed. To reduce noise, run on local disk on otherwise idle networked Linux PC. For robustness to noise, fitness is 25^{th} percentile (i.e. 3^{rd}) of 11 sequential measurements.
Population:	Panmictic, non-elitist, generational. 100 members.
Parameters:	Initial population of random single mutants. 50% truncation selection. 50% two point crossover and 50% mutation. (Mutations chosen equally between insert, delete, replace and swap.) No size limit. Stop after 200 generations.

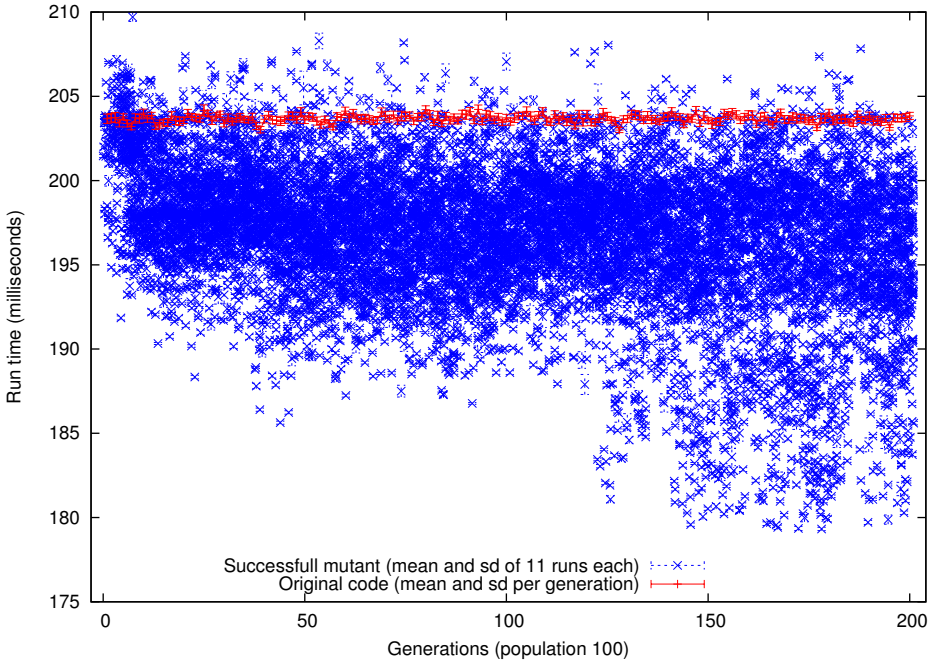

Fig. 1: Evolution of speed, on a 3.60GHz Intel i7-4790 32GB Centos7 desktop.

modified but gave exactly the same answer. It appears that the fastest of these improves the code by taking advantage of the fact that it is being run with its default settings. `<IF_updatePixels.cpp_267><IF_updatePixels.cpp_38>` replaces the condition of an `if` statement (`if( prior2 != 0 )` on line 267) with the `if` condition on line 38. As the compiler is now able to infer the condition will always be true, it can eliminate the `if` entirely. Whereas in the original code, although `prior2` is never zero, it is impossible for the compiler to know this.

3.2 Cleaning up the Best of Run Mutation

The best individual in generation 200 (right Figure 1) gives exactly the same answer on all $2448 \times 3264 = 7\,990\,272$ pixels as the prize winning code and yet runs on average 9.7% faster.

The evolved program contains 22 changes. To determine which are essential, each was removed one at a time to create an intermediate of 21 changes whose performance on the same training image was measured as before. In six cases this made the mutant significantly more than 0.1% worse. A new mutant was constructed from these six (in the same order as the best evolved program). (Notice we measured real runtime and so despite precautions some changes may still be included due to noise.) On the original training image it was 10.0% faster than the original code and produced exactly the same answer. It was run on 447 new images. In 424 cases the new code produced identical answers. In all but five of the remaining 23 images less than nine (median 3) pixels were changed. The biggest difference was 71 out of 7 990 272 pixels. Overall $< 0.000\,000\,1$ of validation pixels are different. On average across the 447 new images the new code is 10.3% (±1.4%) faster. On the six "bikes" images from the 2014 OpenCV challenge competition (which were not used for training), it always produces identical answers and is 13.2% (±1.3%) faster. Taking the mutant and recompiling (gcc 4.8.4) for a virtualized Ubuntu 14.04.1 cloud server we get the same speed up, i.e. 13.1% (±4.1%), however these savings did not carry over to a 1.6GHz Apple MacBook Air laptop with a LLVM compiler. (Some semantics-preserving changes are available via `https://github.com/Itseez/opencv_contrib/pull/687/`.)

3.3 The Six Improvements in the Best of Last Generation

The six line changes are described and partially explained next. They are grouped by which method of the `SuperpixelSEEDSImpl` class they were made to.

`updatePixels()` Lines 113 and 114 are swapped (by swap mutation). No semantic changes are expected. However, it will change the order in which data are read. (Notice the image exceeds our desktop PC's cache of 8 megabytes.)

A copy of line 59 is added to end of the first nested `for` loop which scans the whole image. Line 59 is in the nested `for` loop. It is a call to `update()`. It is difficult to see why this change is beneficial and perhaps it may change the program's output.

`probability()` Lines 279 and 281 are deleted. These are `case:` statements corresponding to values of `seeds_prior` which are never used *in these examples*. Reducing the number of cases in `switch( seeds_prior )` may make it faster for the cases that are used and in this code removing the unused options has no impact on the remaining cases.

`fourbythree()` Lines 338 and 345 are swapped. This has no impact on the output, but does change the order in which array elements are read.

A copy of line 199 (from `updatePixels()`) is inserted into `fourbythree()`. The line inverts global Boolean variable `forwardbackward`. However, `fourby`

`three()` is always called twice, so the second call immediately inverts `forward backward` a second time, restoring the original behaviour. However, it is difficult to see why this mutation would make the program go faster.

4 Related and Future Work

Concerns about the maintainability of genetically improved code have partly been addressed by work on automatically generating documentation for the improvements [3]. Human-written documentation may suffer from all sorts of inconsistencies and omissions, whereas machine-generated documentation could, in principle, be more systematic and thorough.

Nevertheless, our experience of genetic improvement [20,9,1], is that we are at once *delighted* by the surprise of seeing the unexpected improvements that can be found, yet at the same time *challenged* to understand, interpret and explain them. It is one of the advantages of computational search that it can confound and surpass human expectations. Indeed, this 'surprisal' is the underpinning of most human competitive results, some of which have already been reported for SBSE in general [16], and for genetic improvement in particular [12].

The ability to find unexpected solutions is both a strength and a weakness of genetic improvement: It is a strength because it finds improved software that no human would be likely to find, but it can become a weakness if it finds solutions that few humans can understand. Our approach to genetic improvement, ACGI, isolates and contains the modified code, in much the way that a surgeon might seek to isolate a wound [2]. While the modified parts of the code are the source of improvement, to the programmer they might more closely resemble a 'wound'.

In focusing on library functions, our work is similar to the work on deep parameter optimisation [20], which exposes additional parameters to facilitate better tuning at the application layer. However, inserting additional parameters inherently disrupts the API layer. By contrast, the goal of ACGI is to minimally disrupt the API layer, so that details of the modifications that lead to improvements become relatively unimportant to the software engineer. In this way, our approach partly resembles the goal of 'obliviousness' in aspect oriented programming [7]; client code performance is improved, yet it remains oblivious to the changes made in the library functions, since the same API is maintained.

In future work, we will seek to investigate human programmer tolerance to genetically improved code, addressing the fundamental question "how much disruption is a software engineer prepared to tolerate for a given level of performance improvement for a given software engineering domain/application?".

5 Conclusions

We have introduced API-Constrained Genetic Improvement (ACGI) with the aim of bridging the gap between machine and human, to allay concerns about genetic improvement maintainability. Our initial experiments indicate that, despite ACGI's tight constraints, improvements can still be found automatically,

in real-world software systems. E.g. compared to the winner of last year's image segmentation task in the OpenCV State of the Art Vision Challenge we find a speed up of 13% (with little change in functionality).

Acknowledgement: We would like to thank Bobby R. Bruce. This work is part supported by the GGGP and DAASE [4] projects.

References

1. Bruce, B., Petke, J., Harman, M.: Reducing energy consumption using genetic improvement. In: GECCO, pp. 1327–1334 (2015)
2. Cruse, P., Foord, R.: A five-year prospective study of 23,649 surgical wounds. Archives of Surgery, 107(2) pp. 206–210 (1973)
3. Fry, Z.P., Landau, B., Weimer, W.: A human study of patch maintainability. In: ISSTA, pp. 177–187 (2012)
4. Harman, M., Burke, E., Clark, J.A., Yao, Xin: Dynamic adaptive search based software engineering (keynote paper). In: ESEM. pp. 1–8 (2012)
5. Harman, M., Langdon, W.B., Jia, Yue: Babel Pidgin: SBSE can grow and graft entirely new functionality into a real world system. In: SSBSE, pp. 247–252 (2014)
6. Harman, M., Langdon, W.B., Jia, Yue, 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: ASE, pp. 1–14 (2012)
7. Kiczales, G.: Aspect oriented programming. ACM SIGPLAN Notices 32(10), 162–162 (Oct 1997), table of contents includes this invited talk.
8. Langdon, W.B.: Genetically improved software. In: Gandomi, A.H., et al. (eds.) Handbook of Genetic Programming Applications, pp. 181–220. Springer (2015)
9. Langdon, W.B., Harman, M.: Optimising existing software with genetic programming. IEEE TEVC 19(1), 118–135 (2015)
10. Le Goues, C., Forrest, S., Weimer, W.: Current challenges in automatic software repair. Software Quality Journal 21(3), 421–443 (2013)
11. Orlov, M., Sipper, M.: Flight of the FINCH through the java wilderness. IEEE TEVC 15(2), 166–182 (2011)
12. Petke, J., Harman, M., Langdon, W.B., Weimer, W.: Using genetic improvement & code transplants to specialise a C++ program to a problem class. EuroGP 2014
13. Poli, R., Langdon, W.B., McPhee, N.F.: A field guide to genetic programming. `http://www.gp-field-guide.org.uk` (2008)
14. Schulte, E., Dorn, J., Harding, S., Forrest, S., Weimer, W.: Post-compiler software optimization for reducing energy. In: ASPLOS, pp. 639–652 (2014)
15. Sitthi-amorn, P., Modly, N., Weimer, W., Lawrence, J.: Genetic programming for shader simplification. ACM TOG 30(6), 152:1–152:11 (2011)
16. de Souza, Jerffeson Teixeira, Maia, C.L., de Freitas, F.G., Coutinho, D.P.: The human competitiveness of search based software engineering. In: SSBSE (2010) pp. 143–152, IEEE
17. Swan, J., *et al.*: Gen-O-Fix: An embeddable framework for dynamic adaptive genetic improvement programming. Tech. Rep. CSM-195, Uni. Stirling (2014)
18. Van den Bergh, M., Boix, X., Roig, G., de Capitani, B., Van Gool, L.: SEEDS: Superpixels extracted via energy-driven sampling. In: ECCV 2012. LNCS, vol. 7578
19. White, D.R., Arcuri, A., Clark, J.A.: Evolutionary improvement of programs. IEEE TEVC 15(4), 515–538 (2011)
20. Wu, Fan, Harman, M., Jia, Yue, Krinke, J., Weimer, W.: Deep parameter optimisation. In: GECCO, pp. 1375–1382 (2015)