Using Genetic Improvement to Optimise Optimisation Algorithm Implementations

Aymeric Blot and Justyna Petke

University College London (UCL), London, United Kingdom

a.blot@cs.ucl.ac.uk   j.petke@ucl.ac.uk

Keywords: *genetic improvement, software engineering, algorithm design, optimisation.*

1 Introduction

Genetic improvement (GI) [3] uses automated search to improve existing software. It has been successfully used to fix software bugs or improve non-functional properties of software such as running time, memory usage, or energy consumption. Recently, it has been shown that genetic programming, the eponymous GI typical search algorithm, was outperformed by local search strategies [1]. One result of that work was that GI was able to find interesting algorithmic changes in the implementation [2] of two state-of-the-art evolutionary algorithms, MOEA-D and NSGA-II. Here, we reproduce and extend this result with a simple local search, obtaining 10% faster software variants with little to no impact on solution quality in 6/18 GI runs.

2 Genetic Improvement of MOEA/D and NSGA-II

We largely reuse the experimental setup of the original work [1]. However, we only focus on the two MOEA/D and NSGA-II scenarios and only use a single search strategy — a simple first improving local search ($First_1$) — allowing for a much larger training budget (ten times longer). Full details can be found in [1].

Representation. Both MOEA/D and NSGA-II C++ implementations [2] are converted into XML using SrcML and their resulting abstract syntax tree (AST) evolved using the PyGGI GI framework. Software variants are represented by a list of mutations — or in the GI context, *edits* — to apply to the original software.

Edits. Three types of edits are considered: (1) deletion, (2) replacement, and (3) insertion of statements in the initial AST. These are the most typical GI operations.

Dataset. The original C++ implementation provides nine hard-coded "complicated" instances.

Fitness. Fitness is obtained by computing the number of CPU instructions, as recorded by the `perf` Linux tool. It was shown to be much more reliable and less noisy than raw execution time. Software variants leading to solution quality (here, IGD: inverted generational distance) worse than 110% of the original software are automatically discarded.

Training. Starting from an empty list (i.e., the original software) GI edits are either added or removed uniformly at random, and the resulting software variant evaluated on a single instance. Fitness is averaged across three runs with different random seeds as a compromise between controlling for stochasticity and keeping execution time reasonable.

Validation. To avoid overfitting, the final software variant is simplified using a second different unseen instance. Each edit is first individually evaluated and ranked, then a new patch is constructed recombining every edit that doesn't have a negative impact on the fitness.

Testing. The final validated software variant is evaluated on a third and final unseen instance. We also report on the average fitness recorded for all nine instances (27 runs).

Cross-validation. Experiments are repeated nine times using different instance samples, with each of the nine provided instances used at least once during testing.

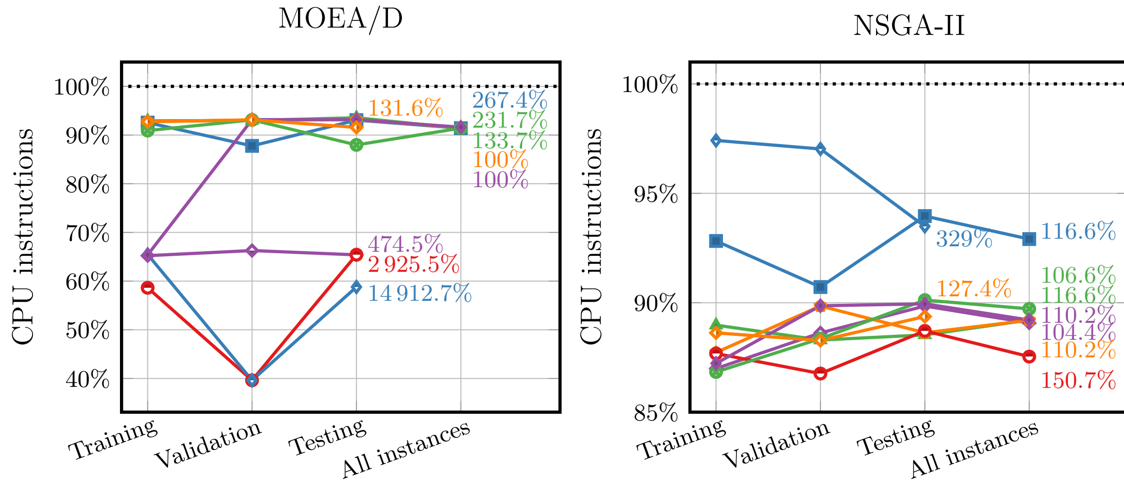

FIG. 1: CPU instruction count and solution quality of final software variants.

3 Results

Figure 1 shows for both MOEA/D (left) and NSGA-II (right) the fitness of the final software variants after training, validation, and testing steps, as well as when reassessed on all nine instances; solution quality is also provided. Results are given in percentage with regards to the original software. Percentages smaller than 100% mean faster or better variants. Variants with solution quality worse than 110% during testing were not evaluated on all instances.

Significant (>5%, up to 41%) improvements were found in all but one training steps. These improvements were almost always reproduced on an unseen instance during validation, and generalised most of the time (12/18) on the third unseen instance. Considering all instances together a few variants (6/18) still generalise with a 10% or less impact on solution quality.

Looking at individual edits, GI was able to remove unnecessary computations and intermediary output, as well as many minor algorithmic variations and simplifications. In particular, we note the relaxation of a uniqueness property during population merging for NSGA-II and an effective change in parameter for mating selection in MOEA/D.

4 Conclusions and Perspectives

Using only a simple local search, GI was able to produce significantly faster software variants for both MEAD/D and NSGA-II that generalised with little to no impact on solution quality in 6/18 GI runs. In the future, we plan to apply GI to all-purpose optimisation frameworks using more widespread and well known optimisation benchmarks. We are also interested in using multi-objective search to simultaneously co-evolve both execution time and solution quality.[1]

References

[1] Aymeric Blot and Justyna Petke. Empirical comparison of search heuristics for genetic improvement of software. *IEEE Trans. Evol. Comput.*, 2021.

[2] Hui Li and Qingfu Zhang. Multiobjective optimization problems with complicated Pareto sets, MOEA/D and NSGA-II. *IEEE Trans. Evol. Comput.*, 13(2):284–302, 2009.

[3] Justyna Petke, Saemundur O. Haraldsson, Mark Harman, William B. Langdon, David R. White, and John R. Woodward. Genetic improvement of software: A comprehensive survey. *IEEE Trans. Evol. Comput.*, 22(3):415–432, 2018.

[1] This work was supported by UK EPSRC Fellowship EP/P023991/1.