Evals is Not Enough: Why We Should Report Wall-clock Time

John R. Woodward
University of Stirling
Stirling
Scotland, United Kingdom
jrw@cs.stir.ac.uk

Alexander E.I. Brownlee
University of Stirling
Stirling
Scotland, United Kingdom
sbr@cs.stir.ac.uk

Colin G.Johnson
University of Kent
Kent
England, United Kingdom
C.G.Johnson@kent.ac.uk

ABSTRACT

Have you ever noticed that your car never achieves the fuel economy claimed by the manufacturer? Does this seem unfair? Unscientific? Would you like the same situation to occur in Genetic Improvement? Comparison will always be difficult [9], however, guidelines have been discussed [3, 5, 4]. With two GP [8] approaches, comparing the number of evaluations of the fitness function is reasonably fair. This means you are comparing the GP systems, and not how well they are implemented, how fast the language is. However, the situation with GI [6, 1] is unique. With GI we will typically compare systems which are applied to the same application written in the same language (i.e. a GI systems targeted at Java, may not even be applied to C). Thus, wall-clock time becomes more relevant. Thus, this paper asks if reporting number of evaluations is enough, or if wall-clock time is also important, particularly in the context of GI. It argues that reporting time is even more important when doing GI when compared to traditional GP.

Keywords

Genetic Improvement (GI), Genetic Programming (GP)

1. POSITION

The halting problem states that we cannot in general determine if a program will halt. This poses a deep issue for GI, but is particularly important if we only compare GI approaches using the number of evaluations. The easiest solution is to set a time-out parameter, after which, if a program has not halted, termination is forced on it. However, if this parameter is set too low, it will prevent correct programs from being produced. Conversely, if this parameter is set too high, valuable wall-clock time will be wasted executing programs which do not halt. Just counting the number of evaluations will not differentiate between this parameter being set a little too high, and much too high: this will make a difference to wall-clock time. We may be able to gather some valuable information as to how to set this parameter for different test cases, given runs of the existing program. An open question is, does setting the time-out parameter a little higher than the time needed for the existing program to run on a test case help GI discover better programs, even though they may not ultimately need the extra time to execute. In other words, as a program is manipulated, its descendants may walk along a path through programs which require a longer runtime, but eventually lead to programs with better properties.

Typically we compare two metaheuristics with a fixed number of evaluations on a given set of problem instances. This is the easiest (and reasonably fair) way to compare algorithms. However, it does not take account of the amount of time to generate the next program. This could be relatively simple and cheap to compute (e.g. randomly exchanging lines of code in a program). However, it could also be more computationally expensive (e.g. instrumenting the program to gather information). This difference is ignored if we are only counting evaluations of the target program.

A possible fair comparison would be to limit the number of clock cycles (a low level measure of time). When transferred to a different machine, GI may take a different number of clock cycles, but this may be a better indication of how good a GI system is than physical time or number of evaluations of the cost function. The number of clock ticks could then be used to estimate wall-clock time.

By measuring wall-clock time and the number of evaluations, we are presenting a fuller picture. Transferring the GI to a faster machine will result in speed up, but it is for the end user to take final responsibility for which machine they run the algorithms on. They may run GI on one hardware architecture, but later execute the genetically improved programs on a different machine with a different configuration.

We can classify programs into 4 execution types:

- *1-programs*, where all nodes in the syntax tree are executed once (e.g. programs constructed with a function set f_1 of arithmetic operators {+, -, *, %}).
- *0-1–programs*, where nodes are either executed once or not (e.g. programs constructed with a function set f_2 containing logical operators {AND, OR, NOT}, where short circuiting is used.)
- *bounded-programs* containing for loops with a determined number of iterations (bounded execution time).
- *unbounded-programs* with while loops with an unknown termination condition (unbounded execution time).

GECCO'16 Companion, July 20–24, 2016, Denver, CO, USA.

DOI: http://dx.doi.org/10.1145/2908961.2931695

Generally, most of GP is with the former two types of program ("1" and "0-1"), while most of GI is with the latter two types ("bounded" and "unbounded"). Adopting a GI approach, which deals with software, forces us to confront the fact that different programs can take vastly different amounts to time to execute. (Of course, GP work exists using Turing Complete instruction sets [7, 2], and there is no reason why GI could not be applied to programs consisting of instruction sets such as $f1$ or $f2$). With the first two types, programs will execute in a comparatively short amount of time (bound by the size of the program). While with the last two types, programs may take an extremely long time to terminate (possibly not halting). Therefore, reporting just the number of evaluations can be more misleading with GI when compared to traditional GP.

The number of evaluations in a GP/GI system, could be counted on at least 4 different levels: These being the number of: 1) programs evaluated (evaluated on a fixed set of test cases), 2) test cases evaluated (the number of test cases can vary over during the training), 3) the number of nodes when a program is executed (e.g. some programs take longer to execute than others), or 4) the number of nodes (weighted) when a program is executed (e.g. some instructions take longer to execute than others),
during a run. By choosing one of these methods to count evaluations over a different method, we may be able to demonstrate that one GI technique is superior to another. However, choosing a different method of evaluation, our claim could be invalidated.

When a program is executed in a GI framework, there are at least 4 possible outcomes; a program 1/ crashes, 2/ fails a test case, 3/ passes a test case, or 4/ is terminated as being possibly non-halting.
One GI system may produce syntactically incorrect programs, or avoid runtime errors, while another GI system may avoid incorrect programs, and have intelligent genetic operators which make used of white-box information obtained by instrumenting the program. A GI which uses clever test case prioritization will detect earlier when to bailout and stop testing a program. Just counting the number of evaluations will not distinguish between these scenarios and is therefore a crude measure of performance. For example, a GP evolving programs that computes polynomials will always succeed and therefore, in this case, it make more sense to count evaluations of the fitness function. Whereas, with a GI system, as the software being evolved is more complicated (taking different amounts of time to execute, failing, or not even halting), time becomes more of a pressing issue.

It may be the case that algorithm $A1$ runs faster than algorithm $A2$ on machine $M1$, but on machine $M2$ the opposite is true. If Computing Science were treated as a natural science, we would report behaviour over a number of machines (e.g. the most popular machines). As much of GI research targets non-functional properties, GI will be aimed at multi-objective optimization. However, as we compare GI systems themselves, we are making multi-objective comparisons, making comparisons even more difficult. Just as we can cherry pick which benchmark instances we select to showcase our the performance of an algorithm, (or equivalently over-tune our algorithm on those benchmark instances), we can cherry pick architectures for GI. We should be as open as possible when making comparisons.

In conclusion, making comparisons will always be problematic. However, the situation is more difficult with GI than with GP, as the scope as to what constitutes an evaluation is broader (involving a possible program crash), and can use more time (as there is possible non-termination of programs). This paper is not claiming we should abandon comparing programs based on the number of evaluations, but to promote the debate and raise awareness. Counting evaluations of the cost function is sensible with a black-box setting, when the evaluations of the cost function are similar. However, we can use white-box approach with GI, calling on expensive but useful instrumentation for example, and therefore the picture is more complex.

With the growth of online repositories, a GI system can be made available, along with the test cases. This will help to independently verify published results and allow for comparisons of different GI systems on different hardware.

2. REFERENCES

[1] S. O. Haraldsson and J. R. Woodward. Automated design of algorithms and genetic improvement: contrast and commonalities. In J. Woodward, J. Swan, and E. Barr, editors, *GECCO 2014 4th workshop on evolutionary computation for the automated design of algorithms*, pages 1373–1380, Vancouver, BC, Canada, 12-16 July 2014. ACM.

[2] B. Harvey, J. Foster, and D. Frincke. Towards byte code genetic programming. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, page 1234, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

[3] J. N. Hooker. Testing heuristics: We have it all wrong. *Journal of Heuristics*, 1(1):33–42, 1995.

[4] D. S. Johnson. A Theoretician's Guide to the Experimental Analysis of Algorithms. In *5th and 6th DIMACS Implementation Challenges*. American Mathematical Society, 2002.

[5] G. Kendall, R. Bai, J. Blazewicz, D. C. P., M. Gendreau, R. John, J. Li, B. McCollum, E. Pesch, R. Qu, N. Sabar, G. V. Berghe, and A. Yee. Good laboratory practice for optimization research. *J Oper Res Soc*, 67(4):676–689, Apr 2016.

[6] W. B. Langdon and G. Ochoa. Genetic improvement: A key challenge for evolutionary computation. In Y. Li, editor, *Key Challenges and Future Directions of Evolutionary Computation*, page Paper ID 17175, Vancouver, 25-29 July 2016. IEEE. Forthcoming.

[7] P. Nordin and W. Banzhaf. Evolving Turing-complete programs for a register machine with self-modifying code. In L. J. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 318–325, Pittsburgh, PA, USA, 15-19 July 1995. Morgan Kaufmann.

[8] R. Poli, W. B. Langdon, and N. F. McPhee. *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008. (With contributions by J. R. Koza).

[9] G. J. E. Rawlins. *Compared to What?: An Introduction to the Analysis of Algorithms*. Computer Science Press, Inc., New York, NY, USA, 1992.