Fitness as Task-relevant Information Accumulation

Colin G. Johnson
University of Kent
Canterbury, Kent, UK
C.G.Johnson@kent.ac.uk

John R. Woodward
University of Stirling
Stirling, UK
jrw@stir.ac.uk

ABSTRACT

"If you cannot measure it, you cannot improve it."—Lord Kelvin

Fitness in GP/GI is usually a short-sighted greedy fitness function counting the number of satisfied test cases (or some other score based on error). If GP/GI is to be extended to successfully tackle "full software systems", which is the stated domain of Genetic Improvement, with loops, conditional statements and function calls, then this kind of fitness will fail to scale. One alternative approach is to measure the fitness gain in terms of the accumulated information at each executed step of the program. This paper discusses methods for measuring the way in which programs accumulate information relevant to their task as they run, by building measures of this information gain based on information theory and model complexity.

1. INTRODUCTION

Genetic improvement is a growing area of search-based software engineering [5]. In this brief paper we would like to argue that we can use a number of methods—information gain measures and model complexity—to evaluate the value of improvements in GI.

2. EVALUATION AS ACCUMULATION OF RELEVANT INFORMATION

Programs (or, in concurrent systems, threads within programs) execute one step at a time; as a program runs, each computation step executes one statement in the program. This is an unremarkable truism. However, this becomes interesting when we consider the question of quantifying *how much closer the program is to solving its task* after each of these steps occur. Clearly, for a correct and non-redundant program, each of these steps is doing something useful; we could say that it is *accumulating task-relevant information* (and, consequently, throwing away information that is irrelevant to the task).

GECCO'15 Companion, July 11–15, 2015, Madrid, Spain

ISBN 978-1-4503-3488-4/15/07…$15.00

DOI: http://dx.doi.org/10.1145/2739482.2768428

Consider the following simple program, which calculates 4-bit parity.

```
1. read v0
2. read v1
3. read v2
4. read v3
5.
6. f0 = XOR v0 v1
7. f1 = XOR v2 v3
8. f2 = XOR f0 f1
9.
10. output f2
```

The substantive lines of code—lines 6-8—each contribute some *information* that is relevant to the solution. For example, line 6 checks whether v_o and v_1 are the same or different. As humans we gain this understanding by reasoning in our minds about the code. Can we, however, come to this conclusion in a data-driven way? Looking at individual test cases doesn't help much—but if we look across a large set of inputs, perhaps this "adding information" comes out as a pattern? Consider Table 1. The first line in this table shows the inputs (in the form $v_3v_2v_1v_0$), and each subsequent line shows a bitwise measure of the difference (XNOR) between the program state generated on that line and the target state. For example, for the input set 0000, $v_0 = 0$, $v_1 = 0$, so line 6 (`f0 = XOR v0 v1`) gives $f_0 = 0$, and the target is 0, so the entry in the table is 1. Call this matrix of differences the *difference spectrum* for that execution.

If we look at the difference spectrum, we can see patterns in it. Line 6 is a list of 4 ones, follow by a list of 8 zeros, followed by a list of 4 ones. Such patterns can be captured by a pattern-finding measure such as the length of the bitstring when exposed to a compression algorithm. We are, in other terms, finding a *compression distance* [2] between the state of the program and the target (see [3] for more details).

Another way to do this is by applying a machine learning algorithm to learn a model of the difference between the state produced by the line, and the target. We use a measure of the complexity of that model as a way of quantifying the complexity of the mapping between that state at the target. This idea has been used—for a different reason—as a contribution towards a fitness measure in GP [4].

Whichever method is used, what we have at the end is a method of assigning a number to each executed step in the program, which (approximately) quantifies how much computation still needs to be done (i.e. the complexity of the "ghost" program [1]). Alternatively, we can think of this as how much task-relevant information is added by carrying out that step—an information gain measure. Importantly,

Line	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
6. XOR v0 v1	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1
7. XOR v2 v3	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1
8. XOR f0 f1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
8. TARGET	0	1	1	0	1	0	0	1	1	0	0	1	0	1	1	0

Table 1: The Difference Spectrum for the Parity Program

assigning this number *doesn't* depend on having the remainder of the program—we can assign this to a *partial* program.

3. FITNESS VIA INFORMATION GAIN

How do we evaluate fitness in genetic improvement? Clearly this depends on the kind of improvement we are seeking to carry out—it could be a measure of a functional property, or a measure of some non-functional property such as speed or readability. Most GI applications have at least two fitness criteria—to maximize the improvement criterion, whilst ensuring that the code changes do not change the correctness of the program on the test cases.

The traditional way to ensure the latter is by re-evaluating each run of the code on the test cases, and aggregating the total score across the test cases (e.g. by counting the number of test cases passed, or by adding together the error on each test case).

We propose that an information gain measure such as those discussed above is a valuable alternative measure for this. Assume that the code is to be modified at a particular point in the program in order to make the improvement. This gives us two points—the starting cut (StartC) and the ending cut (EndC)—between which the GI system improves the code.

We can then construct a *local target* vector at the EndC, by running the original program on its test set and measuring the value that it takes at the EndC for each test case. Then, we can apply an information gain measure such as the ones described above to evaluate the quality, in turn, of each *line* of an improvement, in terms of whether it generates information that is of use in getting towards that local target. This is summarized in Figure 1.

This has a number of advantages. Firstly, it does not rely on the correctness of the remainder of the program. Secondly, it puts pressure on the improvement to make improvements that solve coherent sub-problems. Thirdly, each line of the improvement can be learned one-by-one; we do not need to have a whole improvement before we can evaluate the quality of a line.

Clearly, this cannot be applied naively to programs with loops and conditionals. However, by aligning the step-by-step record of evaluation at appropriate breakpoints, these ideas can be extended to code with these features.

In summary: traditional GI has focused on the *overall* effect of improvements to the solution of the *whole* task. In this way, it is similar to most traditional software testing approaches (even white-box testing usually works *backwards* from end-of-program tests to estimate the location of faults). By contrast, in this approach, we quantify the accumulation of task-relevant information by looking across multiple runs of a program and finding patterns in that information. Because this information accumulation occurs locally in the program, we believe that this approach has a greater capacity to scale than traditional approaches.

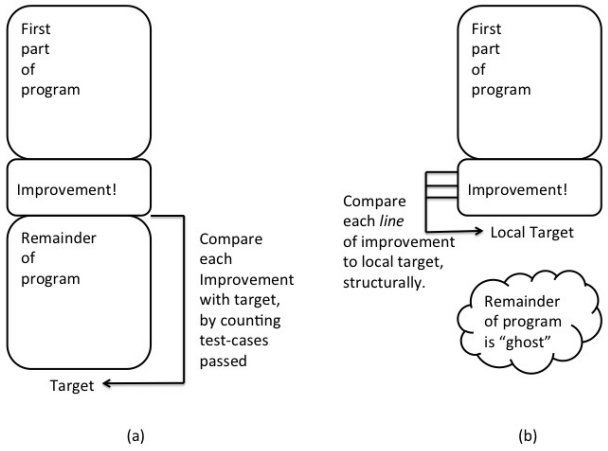

Figure 1: (a) traditional whole-program fitness measure; (b) structural matching to local target.

4. REFERENCES

[1] Kelly Androutsopoulos, David Clark, Haitao Dan, Robert M. Hierons, and Mark Harman. An analysis of the relationship between conditional entropy and failed error propagation in software testing. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 573–583, 2014.

[2] Charles H. Bennett, Péter Gács, Ming Li, Paul M.B. Vitányi, and Wojciech H. Zurek. Information distance. *IEEE Transactions on Information Theory*, 44(4), 1998.

[3] Colin G. Johnson and John R. Woodward. Information theory, fitness, and sampling semantics. In *PPSN 2014 Workshop on Semantic Genetic Programming*, 2014. http://www.cs.put.poznan.pl/kkrawiec/smgp/?n=Site.SMGP2014.

[4] Krzysztof Krawiec and Jerry Swan. Pattern-guided genetic programming. In Christian Blum et al., editors, *GECCO '13: Proceeding of the fifteenth annual conference on Genetic and evolutionary computation conference*, pages 949–956, Amsterdam, The Netherlands, 6-10 July 2013. ACM.

[5] Justyna Petke, Mark Harman, William B. Langdon, and Westley Weimer. Using genetic improvement and code transplants to specialise a C++ program to a problem class. In Miguel Nicolau et al., editors, *17th European Conference on Genetic Programming*, volume 8599 of *LNCS*, pages 137–149, Granada, Spain, 23-25 April 2014. Springer.