On the Uselessness of Finite Benchmarks to Assess Evolutionary and Swarm Methods [*]

Pablo Rabanal, Ismael Rodríguez and Fernando Rubio
Facultad de Informática. Universidad Complutense de Madrid. 28040 Madrid, Spain
prabanal@fdi.ucm.es, isrodrig@sip.ucm.es, fernando@sip.ucm.es

ABSTRACT

We argue against the usage of known finite benchmarks to compare the performance of swarm and evolutionary methods. The key of our criticism is that these methods support a huge set of parameter values and available sub-steps which can be selected and used, and this huge set of choices provides enough versatility to enable an ad-hoc tuning of the method to the particular inputs to be solved (which does not imply properly solving any input not considered in the benchmark). As an alternative, we propose using random input generators rather than known finite benchmarks.

Categories and Subject Descriptors

I.2.8 [**Computing Methodologies**]: Artificial Intelligence—*Search Methodologies*

Keywords

Over-Tuning, Random Benchmarks, Bio-Inspired Methods

1. DISCUSSION

The quality of evolutionary and swarm methods is usually assessed in terms of their behavior for some known sets of problem instances known as benchmarks. By running new metaheuristics for the same benchmark, the community can compare methods with each other in terms of the quality of constructed solutions, time performance, stability, etc.

In this paper we argue against the usage of finite benchmarks to compare metaheuristics with each other. Besides, we propose an alternative comparison method. Our main argument is that evolutionary and swarm methods are easily customizable to make them perfectly fit into *any* finite set of inputs. Note that these methods include a handful of variables and steps where many alternatives can be used. For instance, in a genetic algorithm, we can set variable values such as the mutation probability; we can choose among dozens of crossover methods proposed in the literature (or we can propose our own crossover method); or we can even choose our particular way to represent candidate solutions into chromosomes, among many other possibilities. In general, algorithm designers have an infinite set of parameter values and alternative sub-methods to vary, and these abundant choices can be used to convert any generic algorithm into a completely ad-hoc solution to handle a *specific* set of finite inputs. That is, the rich sets of algorithm parameters and customizable steps and sub-steps of many metaheuristics are expressive enough to let designers subtly convert (in an aware or unaware manner) any generic metaheuristic into an algorithm specifically tuned to efficiently deal with some given finite set of inputs –although it can be very inefficient to handle other inputs.

In other areas like machine learning, it is common to clearly separate the instances used during the training process from the instances used to asses the quality of the results. However, in the optimization area it is very common to use the same benchmark for both steps. This choice is known to increase the risk of over-fitting [2, 3] to a very specific input. Thus, a metaheuristic can be over specialized for a specific set of instances by over-tuning it [1]. We argue that over-tuning not only may break the performance balance among inputs by favoring some specific inputs over others. We think that it also invalidates comparisons with known benchmarks as a reliable scientific method for performance assessment.

Several researchers have warned against the risks of over-tuning. However, little has been said about how easy over-tuning can be if developers push hard to make their algorithms reach good results for specific benchmarks. For instance, rules could subtly make solutions evolve towards some predefined values which, in particular, coincide with the optimal solutions or are near to them. Let us consider an optimization problem defined in such a way that the optimal solution is at point $\bar{0} = (0, \ldots, 0)$. The arithmetic of transformation rules could make these rules be slightly biased towards approaching all components of solutions to 0. In this case, the performance of the method would be good for optimizing functions having the optimal point at $\bar{0}$ or near to it. Let us suppose that our space solutions is $[-d, d]^n$. If initial solutions are randomly generated in this space and all points are given the same likeliness to be chosen, then the center of mass of all points will very likely be near point $\bar{0}$. This could make a swarm algorithm such as Particle Swarm Optimization or an evolutionary algorithm such as GA be biased towards generating more subsequent

[*]Research supported in part by projects TIN2012-39391-C04-04 and S2013/ICE-2731.

GECCO '15 July 11-15, 2015, Madrid, Spain
© 2015 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-3488-4/15/07.
DOI: http://dx.doi.org/10.1145/2739482.2764672

solutions near $\bar{0}$ (regardless of any other additional consideration like the fitness). Over-fitting could also be due to tailoring the solutions representation or the algorithm parameters for the particular instances under consideration.

In order to fairly compare methods, we propose the following alternative: rather than using a given finite set of inputs, we consider constructing and using *random generators of inputs* for the problem under consideration, and we study their usage to compare metaheuristics with each other. Note that using a random input generator also has some disadvantages. In particular, any random input generator will necessarily be *biased*, and comparing the performance of two metaheuristics could be less straightforward. Despite these disadvantages, we argue that random input generators are probably a better choice to provide a fair assessment of swarm and evolutionary methods.

Let us suppose that we have designed a new method M_2 and we wish to compare its performance with that of some method M_1 previously known in the literature. Instead of assessing M_2 by running it for the finite benchmark that was used in the past to collect the reported results of M_1, we propose comparing them by considering their results for (different) sets of inputs independently generated by the same random input generator. That is, we do not assess the interest of M_2 by running it for the same fix benchmark as that originally used in the literature to illustrate the performance of M_1. Instead, we show the interest of M_2 by running the same *random input generator* as that originally used to construct the benchmark used to assess M_1 when it was firstly proposed, and then the *new* set of instances generated by that generator is used to evaluate M_2 and compare it with M_1. In this way, we could provide a fair comparison between M_1 and M_2 in such a way that the benchmark for M_2 is not known *in advance*.

We could argue that researchers claiming to have followed it could, in fact, firstly run the random input generator and *next* tune up their algorithms for the generated benchmark. Moreover, researchers could say that their benchmarks were randomly generated, but in fact they could be manually designed ones. In order to cope with these problems, we propose a simple protocol for random benchmark generation. For many popular problems (3-SAT, TSP, 0/1 Knapsack, etc), the corresponding random benchmarks generator should be hosted in a public web. Each time a researcher asks for a benchmark in the site, the researcher must provide his name and the executable program the researcher is intended to use later for solving the instances of the requested benchmark. Next, the generator produces the benchmark and records the following information in a public file: name of the researcher, the program file he is intended to run, the benchmark returned to the researcher, and the date and hour of the request. Note that the web does not need to *execute* the program of the researcher for the benchmark, it just needs to publicly store it.

Another difficulty of using random input generators is that any random input generator will be *biased*, in the sense that it will tend to construct inputs of some form with higher probability than others. Note that this problem is impossible to avoid, as it is not possible to give all possible inputs the *same* probability of being generated, since the set of possible inputs is generally infinite. However, handling some known bias over the *complete* set of possible inputs will provide a less biased comparison than if we just consider some finite set of inputs. Note that creating an ad-hoc solution to take advantage of some probabilistic bias over an infinite domain is harder than creating an ad-hoc solution for handling some finite set.

A last difficulty of using random input generators is that comparing the performance of two methods is less straightforward. Let us suppose that we want to compare our method M_2 against some previously known method M_1. If we can trust previous reported experimental results on M_1 (as it should ideally be), it would be preferable focusing on performing new experiments only for M_2 and comparing both results afterwards, as repeating experiments for M_1 could be a very time-consuming task. Note that we cannot use the same benchmark as that used in the past for M_1 because it would let us over-tune M_2. On the contrary, providing a fair comparison in this case requires running the same generator used in the past to generate the benchmark for M_1, this time to create the set of instances for M_2, and next comparing old results for M_1 and new results for M_2 in terms of their respective performance for these *different* sets of inputs. Note that both sets could include inputs of different sizes. We know that this comparison is *fair* because the same rule produces the inputs for both methods, although statistically removing the discriminating effect produced by using different sets could require both sets to be big. We may also consider this alternative: we run the generator for constructing the inputs of M_1, and next we require the generator to produce random inputs of the *same size* as those included in the former set, in order to construct the set of inputs of M_2.

We have performed some experiments where we solved MAX-3SAT instances generated by running different random input generators. The results obtained support our idea that, in practice, the performance of two metaheuristics M_1 and M_2 can be *fairly* compared by running them for two different sets of instances independently generated by the *same* generator. In order to assess the usefulness of using generators for fairly comparing the performance of different methods, results for different benchmarks constructed by the same generators should have similar performance when using the same configuration in the generator. In particular, if a new algorithm is to be compared with other algorithms by using the same generator and configuration, but with different instances, the performance results of each algorithm should not be significatively influenced by the specific instances under use. Otherwise, the comparison would not be fair. In our experiments, the deviation is quite small in most cases. Hence, even though the benchmarks are somehow random, the results are basically the same provided that the size of the benchmark is not too small.

2. REFERENCES

[1] M. Birattari. *Tuning metaheuristics: a machine learning perspective.* SCI 197. Springer, 2009.

[2] S. Geman, E. Bienenstock, and R. Doursat. Neural networks and the bias/variance dilemma. *Neural computation*, 4(1):1–58, 1992.

[3] V. N. Vapnik. *Statistical learning theory.* Wiley New York, 1998.