

A Critical Note on Experimental Research Methodology in EC

A.E. Eiben and Márk Jelasity
Department of Artificial Intelligence, Free University of Amsterdam
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands

Abstract—In this paper we point to some essential shortcomings in contemporary practice in performing and documenting experimental research in EC. We identify some crucial problems and the limitations of this practice, and elaborate on research directions that should be pursued to improve the quality and relevance of experimental research.

I. INTRODUCTION

THE aim of this paper is to deliver a contribution to improving the commonly practiced experimental research methodology within evolutionary computing. Our main goal is to initiate a discussion about the means and ends of experimental EC research. In the long term we hope that this discussion will result in a widely accepted and practiced, sound research methodology supporting research results that are better founded and more relevant than today. Studying the literature it is clear that there are already some publications that can be related to various aspects of this subject. For the present treatment we deliberately chose a “light” citation policy in order to avoid the impression of an annotated bibliography and to emphasize the messages better, while acknowledging inspiring contributions from fellow researchers. Furthermore, let us explicitly state that much of our criticism on the present practice is applicable to some of our own work too.

We approach the methodology question here from the angles of generalizability, performance measures, and reproducibility. For an illustration of the nature of problems we wish to address here let us consider the following imaginary example of a paper representing common practice in experimental EC research. (Note however, that our discussion will not be restricted to this kind of papers.) The imaginary paper proposes a new feature in EAs and shows the merits of the new feature by experiments. The experimental section is structured as follows:

1. A testbed of functions, respectively problem instances, is specified.
2. The new EA and a “standard” EA are run on these functions (problem instances).
3. The outcomes of the experiments are reported by presenting some performance measures in graphical or table form.

Thereafter, conclusions are drawn on the merits of the new EA, arguing that the newly added feature – the main contribution of the paper – improves performance.

In *Proc. of the 2002 Congress on Evolutionary Computation (CEC 2002)*, IEEE, pp. 582–587.

A.E. Eiben: gusz@cs.vu.nl

M. Jelasity: jelasity@cs.vu.nl, also at RGAI, Univ. of Szeged, Hungary

The problem with this imaginary paper, and with all others it represents, is that:

1. The testbed of functions (problem instances) is chosen in an ad hoc manner, without motivating the choices. For instance, 15 functions are chosen from the literature with the implicit suggestion that the selection criterion was “others use these functions too”.
2. The outcomes are given in terms of some performance measures, averages, standard deviations, etc., but these measures are used without relating them to the research objectives and the expectations w.r.t. the investigated algorithms.
3. The conclusions are formulated in general, e.g., “the new feature improves EA performance”. The authors do not restrict the scope of their claims to *these test functions* and suggest that the claims hold for many other functions too, without specifying what kind of functions.
4. The results are hard to reproduce, because many details of the EAs are not explicitly specified and the source code is not made available.

In the remainder of this paper we discuss these and other, related problems. That is, we do not restrict the discussion to this illustrative example, but adopt a more general context. Section II concerns the issue of generalizability, corresponding to items 1 and 3 in the previous list. Performance measures and statistics are considered in section III, related to item 2 above, while section IV addresses related experimental objectives. The issue of reproducibility is handled in section V. We close the paper with a summary of the most important issues, also carrying hints on further research.

II. GENERALIZATION OF RESULTS

The essence of empirical science is generalization, i.e. generating scientific knowledge from empirical data through induction. The most important methodological issue is therefore to establish a framework in which this process becomes possible.

Although a large part of evolutionary computing is mainly empirical in nature, the present practice is more like fact collection without real generalization. Progress in the last decades into this direction seems to be only that the size of test suites increased from 5 (the famous De Jong functions) to twentysomething. The underlying hidden assumption was that if EA-A performs better on such a test suite than EA-B then EA-A is probably better in general. While it is true that more data gives a more reliable picture, this increase in the number of test functions is purely quantitative.

Probably the most straightforward idea towards a qualitative improvement is to divide test functions into classes such that functions within one class are similar and functions from different classes are different. Then the generalization could be performed w.r.t. these classes, e.g. one could state that EA-A works better on problem class P than EA-B. Clearly, if we could find a way that allows us to defend these kinds of statements, we could:

- say more than simply establishing that EA-A outperforms EA-B on 10 (arbitrarily) selected functions and
- clarify the scope of our results by making it explicit.

The second item is related to the wide-spread skepticism over general-scope or undefined-scope claims which is a result partly of the NFL theorem [1] debate and partly of the growing evidence supporting the superiority of approaches that deploy domain specific algorithmic components and/or representation.

The remaining part of this section elaborates on the issues raised by the above paragraphs, i.e. on the possibilities of classification of problem instances and the proper justification of inductive claims.

A. Problem Classes

Let us define an optimization problem instance as the problem of finding the minimum of a function $f : F \rightarrow \mathbb{R}$, where F is called the *search space* or *phenotype space*. A problem class is a set of problem instances with a common search space. For the sake of simplicity we exclude constrained problems and multicriteria problems but the discussion could have been extended to these cases too.

To be useful for our purposes, a problem class should be neither too general nor too specific. It should be general enough for reasonable induction but specific enough to be able to differentiate between algorithms or algorithm classes. The instances within a class can be quite diverse but they should have at least some common properties expressible by some form of knowledge. One possibility is to find a specific neighborhood relation that reflects the structure of the problem class, i.e. specific operators optimized for the problem class or special representations.

Obviously, quite some research is needed to draw borders between problem classes in a meaningful way. In the following, we want to express caveats about careless usage of notions.

A.1 Useless Classes

Unfortunately it is general practice to use inappropriate class labels. One mistake is to use terms that were defined for other purposes, like NP-hardness or the class of scheduling problems. Another mistake is to use a given vocabulary for composing classes that turn out to be inhomogeneous w.r.t. EA behavior. In the following we will discuss three features that prevent a class from being useful in EC research by the following examples:

- The class of NP-hardness is irrelevant.
- The class of “scheduling problems” is too general.
- Some classes defined by a selection of function properties are not distinguishing for EAs.

A.1.a NP-hardness. For a definition of NP-hardness see e.g. [2]. This label is often used in the research literature to suggest that a problem instance is hard. This is extremely misleading because NP-hardness is not even a property of a problem, it is a property of problem classes! Furthermore, only an infinite problem class can be NP-hard. We can think of this property as a characterization of the infinite problem instance size.

While useful in complexity theory this label is completely meaningless in heuristic optimization. Note that sometimes even the notion NP-complete is used which characterizes decision problems, not optimization problems. This adds yet another mistake. Furthermore, if we consider a problem instance from a problem class that is NP-hard we might very well have a problem in question that is easy to solve.

Instead of using the label NP-hard it is advisable to turn to theoretical results which exist for many NP-hard problem classes describing which instances are really hard. This might be a possible way to define meaningful and challenging problem classes. In Sections II-A.4 and II-A.3 we return to this question.

A.1.b Scheduling problems. Anyone seriously involved in solving real life scheduling problems would agree that there is no such thing as a scheduling problem in general. Any mathematical definition of a scheduling problem is too general and turns out to be equivalent with almost everything else (e.g. by resulting in an NP-hard class, see above) making induction practically impossible. Claims like “my scheduling algorithm is better in general” are therefore questionable. In other words, the main problem of this class is that it is too general to help meaningful induction.

Another well-known semi-practical problem to illustrate this matter is that of graph coloring. While it sounds narrow enough to make up a meaningful problem class it is well-known that specific features of the graphs to be colored have a great impact on the hardness of instances within the subclasses defined by these features. Think, for instance, of flat graphs, equipartite graphs, etc., [3].

A.1.c Commonly used test suites. The importance of composing proper test suites has been well noticed in EC [4]. Nevertheless, there has been not much effort in explicitly addressing the appropriateness of the classes of test functions resulting from the commonly used features, such as (multi)modality, separability, etc. The few attempts into this direction indicate that this vocabulary (and the problem classes definable with this vocabulary) is inappropriate. For instance, [5] explicitly aimed at relating EA behavior to problem classes defined along the traditional lines, but failed in achieving this because the experiments showed significantly different EA behavior within the classes in question. This, and similar other experiences, strongly indicate that a new vocabulary is needed to capture EC-relevant properties of test functions.

A.2 Natural Classes

Classes of problem instances that emerge from a specific real life situation we will call a *natural* class. For instance,

timetabling on a university or even on universities within a country specifies a natural problem class. In our perception, such a class conforms to our requirements to be useful (i.e. general enough) *because* it emerges from a real situation even if it consists of only a few instances. On the other hand sufficient specificity normally follows from the particular constraints and circumstances that are present in a typical real life setting.

A.3 Artificial Classes

Beside the natural classes it is a very interesting research field to find artificial classes that are general enough for reasonable induction but specific enough to be able to differentiate between algorithms or algorithm classes. We list directions and motivations for such research:

1. Find hard or easy problem classes for specific algorithms. This has been a flourishing field for many years. Finding hard or easy (royal road) problems helps us justify our existing theories of the behavior of algorithms and helps us develop new theories.
2. Find classes that somehow capture properties of natural classes (see Section II-A.2) while being easier to handle and experiment with, and also easier to describe theoretically. These classes would help develop and test algorithms that are likely to work well on the corresponding natural class.
3. Find classes that are simply interesting because they have the required balance between generality and specificity and are able to differentiate between algorithms. They might become useful later.

In case of artificial classes it must be shown that these classes are indeed appropriate. In order to do this one could develop a *problem instance generator* that generates instances from the class in question. Then it could be demonstrated that this class is relevant using any appropriate statistical and experimental method. We do not intend to prescribe how it should be done, but one example could be to show that an algorithm can be tuned to perform better *over the whole class* using statistical evidence.

An additional research field can aim at finding methods that *recognize* instances of a class from some kind of sample from the search space. Such methods (in the context of hard/easy problem classes) have already been suggested. This way properties of natural classes could be identified in terms of already existing and established artificial classes which would allow selection of appropriate optimizers or techniques in a well grounded way.

A.4 Problem instance generators

For some problems that have been the subjects of intensive study in the algorithmic community there exist random problem instance generators that create instances from the class in question. Typically such a generator needs a random seed and some problem specific parameters to be set by the user. A great advantage of such a generator is that it enables a systematic study: varying the given parameters algorithm behavior can be related to problem features. Well-known examples are graph coloring or 3SAT problems, where not only gener-

ators are available, but also the location of hard instances is known in terms of the parameters of the given generator. The collection of generators at [6] is a good initiative for example, although experimental and/or theoretical justification of the usefulness of the generators would be a desirable extension to the list.

B. The Relationship of Algorithms and Problem Classes

Once a set of well established problem classes is available (natural or artificial) scientific research has to clarify the relationship between these classes and the different algorithms. (Note that without *well established* problem classes it is not possible.) The outcome of such research projects can be very diverse. Some examples follow here: EA-A and EA-B are equivalent over problem classes $C_1 \dots C_n$; for problem class C_1 the best algorithm to date is EA-X; the best problem class for algorithm EA-Y is C_1 ; etc.

Such claims can make sense if the problem classes are well established and the research is based on sound experimental methodology and statistical methods. Again, it would be impossible to prescribe every way of doing proper experiments. We give one example in Section II-B.1 and we discuss some more theoretically oriented possibilities in Section II-B.2.

B.1 Learning the Best Algorithm

As an example we give a scenario for doing proper experiments for finding the best parameter set for a given problem class C_1 for a given algorithm. This scenario is used in the machine learning community for inductively find optimal models of databases (i.e. knowledge):

- choose a number of functions (problem instances) from C_1 for calibrating the algorithm: the training set,
- choose a number of *different* functions (problem instances) from C_1 for evaluating the algorithm: the test set,
- run the algorithm on functions (problem instances) from the training set to calibrate (tune) the algorithm,
- run the calibrated versions of the algorithm on functions (problem instances) from the test set,
- report the outcomes of the experiments on the test set,

This kind of techniques are widely used in machine learning and are called *wrapping* [7]. The point is that the evaluation is done on the test set and not the training set, which is not common practice in the EA community. As a result of the application of these techniques we gain much more reliable indicators of a given algorithm on a given class. This idea can be generalized to many more cases when we are interested in the relationship between algorithms and problem classes.

B.2 Fitness Landscapes

A more theoretical approach is possible with the tools of fitness landscape analysis. As we mentioned earlier the problem instances usually have landscapes as well. This is due to the fact that the search space usually has some structure in the form of a distance function or neighborhood relation. This way the problem classes become in fact fitness landscape classes. This offers a next level of abstraction when the actual details of a problem are ignored and only the landscape

is taken into account. The situation would be analogous to calculus where e.g. continuity of real functions can be generalized to any topological space. For an established problem class one could try to characterize it in terms of landscapes.

An EA defines another landscape through the applied *representation* and *operators* and maybe through other miscellaneous features like elitism, etc. This can be thought of as a transformation of the landscape. A promising research direction could be looking at successful algorithms for given problem classes and trying to reverse engineer the way the algorithm transforms the problem landscape.

These results might become directly applicable in practice when and if the fitness landscape of a natural problem class turns out to belong to a known landscape class. Then we could apply off-the-shelf landscape transformations (i.e. algorithms) with the appropriate operators and representation.

Of course a huge obstacle is the lack of powerful methods to characterize landscapes from sampling data (either algorithm trajectories or random samples). But it is not unlikely that meaningful landscape classes could be found which could be recognized using such methods with a sufficient reliability. Especially if we are thinking of a long term solution when a huge amount of data can be collected from a given landscape while using less advanced algorithms on it.

III. PERFORMANCE MEASURES AND STATISTICS

Experimental comparisons of two (or more) evolutionary algorithms – and the discussion in the foregoing sections – assume the usage of some algorithm performance measures. Claims on ranking algorithms are always meant as claims on ranking their performances. By the stochastic nature of EAs a number of experiments needs to be conducted to gain sufficient experimental data and performance measures within EC are based on some statistics. In this section we discuss performance measures and their usage, emphasizing that their usage and the interpretation of results should follow the goals pursued with the EAs in question.

A. Different performance measures

For problems where the (optimal) solution can be recognized one can easily define a success criterion: finding an (optimal) solution. For this type of problems the Success Rate (SR) measure can be defined as the percentage of runs terminating with success. For problems, where the optimal solutions cannot be recognized, the SR measure cannot be used. In general, this is the case if the optimum of the objective function is unknown, perhaps not even a lower/upper bound is available. As an example, think of a university timetabling problem.

The Mean Best Fitness measure (MBF) can be defined for any problem that is tackled with an EA – after all *any* EA is using a fitness measure. For each run of a given EA the best fitness can be defined as the fitness of the best individual at termination. The MBF is the average of the these best fitness values over all runs.

Note, that although SR and MBF are related, they are different and there is no general advice on which one to use for

algorithm comparison. The difference between the two measures is rather obvious, SR cannot be defined in a meaningful way for some problems, while the MBF is always a sound measure. Furthermore, all possible combinations of low/high SR/MBF values can occur. For example, low SR and high MBF is possible and it indicates a good approximizer algorithm: it gets close consistently, but seldomly really makes it. Such an outcome could motivate increasing the length of the runs hoping that this allows the algorithm to finish the search. An opposite combination of a high SR and low MBF is also possible indicating a “Murphy” algorithm: if it goes wrong it goes very wrong. That is, those few runs that terminate without an (optimal) solution end in a disaster, a very bad best fitness value deteriorating MBF. Clearly, whether the first or the second type of algorithm behavior is preferable, depends on the problem. As mentioned above, for a timetabling problem the SR measure is not even meaningful, so one is interested in a high MBF. To demonstrate the other situation, think of solving the 3SAT problem with the number of unsatisfied clauses as fitness measure. In this case a high SR is pursued, since the MBF measure – although formally correct – is useless because the number of unsatisfied clauses at termination says, in general, very little about how close the EA got to a solution. Notice, however, that the particular application objectives (coming from the original problem solving context) might necessitate a refinement of this picture. For instance, if the 3SAT problems to be solved represent practical problems with some tolerance for a solution, then measuring MBF and striving for a good MBF value might be appropriate.

For both aforementioned measures it is assumed that they are taken under an a priori specified limit of computational efforts. That is, SR and MBF always reflect performance within a fixed (maximum) amount of computing. The complementary approach is to specify when a candidate solution is satisfactory and measure the amount of computing needed to achieve this solution quality. Roughly speaking, this is the issue of algorithm speed. Within EC algorithm execution times are typically measured by the number of evaluations, rather than CPU times or system times. Clearly, this is meant to eliminate effects of particular implementations, software, and hardware, thus making comparisons independent from such practical details. The corresponding speed measure is the Average number of Evaluations to a Solution (AES), defined over those runs that terminate with a solution (a candidate with satisfactory quality). Sometimes the average number of evaluations to termination measure is used instead of the AES, but this has clear disadvantages. Namely, for runs finding no solutions the specified maximum number of evaluations will be used when calculating this average. Hereby, this measure mixes the AES and the SR measures and the outcome figures are hard to interpret. Using the AES measure generally gives a fair comparison of algorithm speed, assuming that all fitness evaluations cost the same amount of computing and that fitness evaluations consume most of the execution time of the algorithm. Hence, AES can be misleading in two cases. First, if some evaluations take longer than others. For instance, if a repair mechanism is applied, then evaluations invoking this

repair take (much) longer. An EA with good variation operators could then proceed by chromosomes that do not have to be repaired, while another EA may need much repair. The AES values of the two may be close, but the second EA would be much slower, and this is not an artifact of the implementation, etc. The second case is, when evaluations can be done very fast. Then the AES does not reflect algorithm speed correctly, for other components of the EA have a relatively large impact.

Algorithm speed, or rather the pace of progress, can be also measured in cases where one cannot specify satisfactory solution quality in advance. In these cases the best (worst/average) fitness value of the consecutive populations is plotted against a time axis – typically the number of generations or fitness evaluations. Clearly, such a plot provides much more information than the AES and therefore it can also be used when a clear success criterion is available. In particular, a progress plot can help ranking two algorithms that score the same on AES. For example, progress curves might disclose that algorithm A has achieved the desired quality already halfway of the run. Then the maximum number of evaluations might be lowered and the competition redone. The chance is high that algorithm A keeps its performance, e.g., its MBF, and algorithm B does not, thus a well-motivated preference can be formulated. Another possible difference between progress curves of algorithms can be the steepness towards the end of the run. If, for instance, curve A has already flattened out, but curve B did not, one might extend the runs. The chance is high that B will make further progress in the extra time, but A will not, thus again, the two algorithms can be distinguished.

B. Peak vs. average performance

For some performance measures (not for all of them) there is an additional question whether one is interested in peak performance or average performance considered over all these experiments. In EC it is typical to suggest that algorithm A is better than algorithm B if its average performance is better. In practical applications, however, one is often interested in the best solution found in X runs or within Y days (peak performance), and the average performance is not that relevant. To be more precise, assume that the algorithm under investigation is meant to solve real-life problems. The type of problems and the context these problems arise from have implications for the typical problem solving session where the algorithms are applied. Then the following cases can be distinguished.

1. In a problem solving session there is time for more runs on the given problem and the final solution can be selected from the best solutions of these runs. For instance, creating the timetables for a university might take a few weeks, allowing, say, 25 runs of the GA doing the job. In this case an algorithm setup with high peak performance is appropriate even if its average performance is not too good.
2. In a problem solving session there is only time for one run that must deliver the solution. This might be the case if a computationally expensive simulation is needed to calculate fitness values, or a real-time application. For this kind of problems an algorithm with high average performance is the best option, since this carries the lowest risk of missing the only

chance we have.

Notice that the first kind of problem solving seems to be the most frequently occurring one in practice. In this light, it is strange that the huge majority of experimental EC research is comparing average performances of algorithms.

B.1 Example

What we consider here is the interpretation of figures concerning averages and standard deviations. In EC it is common to express preferences for algorithms with better averages for a given performance measure, e.g., higher MBF or lower AES, especially if a better average is coupled to a lower standard deviation. This attitude is never discussed, but it is less self-evident than it might seem. The key is again to consider the objectives of applying the algorithms. Remaining with the timetabling example assume that two algorithms are compared based on 50 independent runs and the resulting MBF results that are given in figure 1. Then it is tempting to

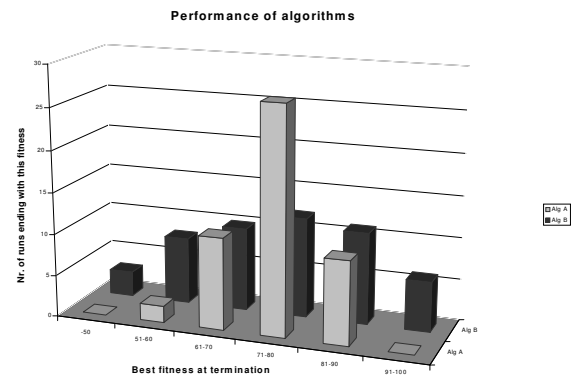


Fig. 1. Comparing algorithms by the best found fitness values

conclude that algorithm A is better because of the (slightly) higher MBF and the more consequent behavior (lower variation in best fitness values at termination). Notice, however, that 6 runs of algorithm B terminated with a solution quality algorithm A has never achieved. Therefore, if a typical problem solving session allows 50 runs of the given EAs, then algorithm B is preferable because of the higher chance of delivering a better time-table. Obviously, this reasoning can be applied to any problem or problem context, where peak performance is desirable.

This discussion of performance measures is far from exhausting. However, it is sufficient to illustrate the point that for a sound comparison it is necessary to specify the objectives of the algorithms in the light of some problem solving context and derive the performance measures used for comparison from these objectives. In other words, the choice of how we evaluate and compare EAs should be a consequence of what we want the EAs to do.

IV. EXPERIMENT OBJECTIVES

Throughout this paper we have not made an explicit distinction between possible goals of experimentation. Roughly speaking we can distinguish two such goals: optimizing and understanding algorithms.

Optimizing is the commonly practiced sport of designing an EA that beats others on a given problem or set of problems. This kind of experimental research finishes with establishing the fact of the superiority of a given EA. This is a fully legitimate research programme, our discussion so far implies two caveats:

1. The scope of claims regarding algorithm superiority must be established carefully – see section II.
2. The usage performance measures and the interpretation of statistics should not be done blindly, but according to how the algorithm will be used – see section III.

A research programme aiming at understanding goes further than fact finding. In particular, it is not limited to establishing *that* an EA is better than another, but also investigates *why*. Very often, developing a good understanding is also meant to serve for (later) optimization, but only implicitly. To name a simple example we can take one problem class and one EA variant, for instance a bit-string GA, still having many degrees of freedom in the exact algorithm setup by various operators, selection mechanisms, and population sizes. Trying to understand how different combinations of algorithm features influence algorithm behavior needs a careful design of experiments, because the brute force approach of trying all combinations is not practicable. We will not go into general principles of experiment design, good treatments of this subject can be found for instance in [8], [9]. Here we only note that in this case the weaponry of algorithm performance measures is extended with indicators of algorithm behavior. The leading question behind such indicators is “What happens during execution?”, rather than “How far can we get?”. Commonly used indicators include population distribution in the phenotype space, allele distribution in the genotype space, progress rates during execution and alike.

V. REPRODUCIBILITY OF RESULTS

Verifying results found in the literature is in practice almost impossible. That is, running a reportedly good algorithm on one’s own data is a difficult task. The details presented in a typical paper are insufficient to ensure that one would implement the same algorithm. To overcome this difficulty there are more possibilities. The one we advocate here is a rather idealistic solution: standardization of code. In particular, developing and using a standardized evolutionary algorithm library would be a double edged sword. On the one hand, individual researchers would be freed from implementing algorithms from scratch. Coding a new EA could be restricted to writing the code for the newly invented features with the guarantee that the rest of the code is thoroughly tested and works correctly. On the other hand, fellow researchers, thus the whole EC community, would have a simple way of reproducing the experiments. If they use the same library the new code can be fetched from the author’s electronic repository and tested on

one’s own problems in one’s own environment.

VI. CONCLUDING REMARKS

In this paper we have discussed various aspects of current practice in experimental research within evolutionary computing. Briefly summarizing we touched upon the following issues.

- Investigations are not targeted at acquiring generalizable knowledge. The “wrapping approach” from machine learning should be imported into EC to improve the inductive power of results.
- Well-defined and appropriate problem classes (that are necessary to follow the “wrapping approach”) are lacking. Intensive research efforts are needed to develop such classes for EC research.
- Objectives of the experimentation and the studied algorithms should be explicit. Performance measures and statistics should be used in line with these objectives.
- Reproducibility of experiments and verification of others’ results should be improved.

We are well aware of the fact that our treatment is not extensive. We do not consider all problems and we do not provide ready solutions to the problems we do discuss. However, we hope to initiate more awareness concerning these issues and in general about the lack of a solid experimental methodology in EC. The discussions and follow-up publications we wish to trigger will hopefully lead to a widely accepted and practiced, sound research methodology supporting research results that are better founded and more relevant than today.

ACKNOWLEDGMENTS

The authors are indebted to Ben Paechter for his valuable comments on earlier versions of this paper.

REFERENCES

- [1] David H. Wolpert and William G. Macready, “No free lunch theorems for optimization,” *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 1, pp. 67–82, Apr. 1997.
- [2] Martin Grötschel, László Lovász, and Alexander Schrijver, *Geometric Algorithms and Combinatorial Optimization*, Springer-Verlag, 2nd edition, 1993.
- [3] E.P.K. Tsang, *Foundations of Constraint Satisfaction*, Academic Press Limited, 1993.
- [4] Thomas Bäck and Zbigniew Michalewicz, “Test landscapes,” in *Handbook of Evolutionary Computation*, Thomas Bäck, David B. Fogel, and Zbigniew Michalewicz, Eds., pp. B.2.7:14–B.2.7:20. Institute of Physics Publishing and Oxford University Press, 1997.
- [5] Agoston E. Eiben and Thomas Bäck, “An empirical investigation of multi-parent recombination operators in evolution strategies,” *Evolutionary Computation*, vol. 5, no. 3, pp. 347–365, 1997.
- [6] William M. Spears, “Genetic algorithms (evolutionary algorithms): Repository of test problem generators,” <http://www.cs.uwo.edu/~wspears/generators.html>.
- [7] Ron Kohavi and George H. John, “The wrapper approach,” in *Feature Extraction, Construction and Selection: A Data Mining Perspective*, Huan Liu and Hiroshi Motoda, Eds., vol. 453 of *The Kluwer International Series in Engineering and Computer Science*. Kluwer, 1998.
- [8] Richard S. Barr, Bruce L. Golden, James P. Kelly, Mauricio G.C. Resende, and William R. Stewart, Jr, “Designing and reporting on computational experiments with heuristic methods,” *Journal of Heuristics*, vol. 1, no. 1, pp. 9–32, 1995.
- [9] J. N. Hooker, “Testing heuristics: We have it all wrong,” *Journal of Heuristics*, vol. 1, no. 1, pp. 33–42, 1995.