5-1995

Testing Heuristics: We Have It All Wrong

John N. Hooker
Carnegie Mellon University, john@hooker.tepper.cmu.edu

Published In
Journal of Heuristics , 1, 33-42.

Testing Heuristics: We Have It All Wrong

J. N. Hooker
Graduate School of Industrial Administration
Carnegie Mellon University
Pittsburgh, PA 15213 USA

May 1995

Abstract

The competitive nature of most algorithmic experimentation is a source of problems that are all too familiar to the research community. It is hard to make fair comparisons between algorithms and to assemble realistic test problems. Competitive testing tells us which algorithm is faster but not why. Because it requires polished code, it consumes time and energy that could be spent doing more experiments. This paper argues that a more scientific approach of controlled experimentation, similar to that used in other empirical sciences, avoids or alleviates these problems. We have confused research and development; competitive testing is suited only for the latter.

Most experimental studies of heuristic algorithms resemble track meets more than scientific endeavors.

Typically an investigator has a bright idea for a new algorithm and wants to show that it works better, in some sense, than known algorithms. This requires computational tests, perhaps on a standard set of benchmark problems. If the new algorithm wins, the work is submitted for publication. Otherwise it is written off as a failure. In short, the whole affair is organized around an algorithmic race whose outcome determines the fame and fate of the contestants.

This *modus operandi* spawns a host of evils that have become depressingly familiar to the algorithmic research community. They are so many and pervasive that even a brief summary requires an entire section of this paper. Two, however, are particularly insidious. The emphasis on competition is fundamentally anti-intellectual and does not build the sort of insight that in the long run conduces to more effective algorithms. It tells us which algorithms are better but not why. The understanding we do accrue generally derives from initial tinkering that takes place in the design stages of the algorithm. Because only the results of the formal competition are exposed to the light of publication, the observations that are richest in information are too often conducted in an informal, uncontrolled manner.

Second, competition diverts time and resources from productive investigation. Countless hours are spent crafting the fastest possible code and finding the best possible parameter settings in order to obtain results that are suitable for publication. This is particularly unfortunate because it squanders a natural advantage of empirical algorithmic work. Most empirical work in other sciences tends to be slow and expensive, requiring well-appointed laboratories, massive equipment or carefully selected subjects. By contrast, much empirical work on algorithms can be carried out on a work station by a single investigator. This advantage should be exploited by conducting more experiments, rather than by implementing each one in the fastest possible code.

There is an alternative to competitive testing, one that has been practiced in empirical sciences at least since the days of Francis Bacon. It is controlled experimentation. Based on one's insight into an algorithm, for instance, one might expect good performance to depend on a certain problem characteristic. How to find out? Design a controlled experiment that checks how the presence or absence of this characteristic affects performance. Even better, build an explanatory mathematical model that captures the insight, as in done routinely in other empirical sciences, and deduce from it precise consequences that can be put to the test. I will give this sort of experimentation the deliberately honorific name 'scientific testing' to distinguish it from competitive testing.

I discuss elsewhere [9] how empirical models might be constructed and defend them as a viable and necessary alternative to a purely deductive science of algorithms. My main object in this paper is to show that scientific testing can avoid or substantially alleviate many of the evils that now stem from competitive testing.

This paper is written primarily with heuristic algorithms in mind, because it is for them that empirical investigation is generally most urgent, due to the frequent failure of purely analytical methods to predict performance. But its points apply equally well to exact algorithms that are tested experimentally. In fact a 'heuristic' algorithm may be more broadly conceived as any sort of search algorithm, as suggested by the historical sense of the word, rather than in its popular connotation of an algorithm that cannot be proved to find the right answer. The fact that some search algorithms will eventually explore the entire solution space and thereby find the right answer does not change their fundamentally heuristic nature.

I begin in the first section below with a description of the current state of affairs in computational testing. The description is a bit stark to make a point, and I hasten to acknowledge that the algorithmic community is already beginning to move in the direction I recommend in the two sections that follow. Perhaps a forthright indictment of the old way, however, can hasten our progress. The final section recounts how a more scientific approach to experimentation avoids the evils of competitive testing.

1 The Evils of Competitive Testing

The most obvious difficulty of competitive testing is making the competition fair. Differences between machines first come to mind, but they actually present the least serious impediment. They can be largely overcome by testing on identical machines or adjusting for machine speed. More difficult to defeat are differences in coding skill, tuning and effort invested.

With respect to coding skill, one might argue that competitive testing levels the playing field by its very competitiveness. If investigators are highly motivated to win the competition, they will go to great lengths to learn and use the best available coding techniques and will therefore use roughly the same techniques. But it is often unclear what coding technique is best for a given algorithm. In any event, one can scarcely imagine a more expensive and wasteful mechanism to ensure controlled testing—more on this later.

A particularly delicate issue is the degree to which one tunes one's implementation. Generally it is possible to adjust parameters so that an algorithm is more effective on a given set of problems. How much adjustment is legitimate? Should one also adjust the competing code? If so, how much tuning of the competing code can be regarded as commensurate with the tuning applied to the new code? One might fancy that these problems could be avoided if every algorithm developer provided a 'vanilla' version of the code with general-purpose parameter settings. But when a new code is written, one must decide what is 'vanilla' for it. No developer will see any rationale for deliberately picking parameter settings that result in poor performance on the currently accepted benchmark problems. So the question of how much tuning is legitimate recurs, with no answer in sight.

A related obstacle to fair testing is that a new implementation must often face off against established codes on which enormous labor has been invested, such as simplex codes for linear programming. Literally decades of development may be reposited in a commercial code, perhaps involving clever uses of registers, memory caches, and assembly language. A certain amount of incumbent advantage is probably acceptable or desirable. But publication and funding decisions are rather sensitive to initial computational results, and the technology of commercial codes can discourage the development of new approaches. Lustig, Marsten and Shanno [13] suggest, for example, that if interior point methods had come along a couple of years later than they did—after the recent upswing in simplex technology now embodied in such codes as CPLEX—they might have been judged too unpromising to pursue.

A second cluster of evils concern the choice of test problems, which are generally obtained in two ways. One is to generate a random sample of problems. There is no need to dwell on the well-known pitfalls of this approach, the most obvious of which is that that random problems generally do not resemble real problems.

The dangers of using benchmark problems are equally grave but perhaps less appreciated. Consider first how problems are collected. Generally they first appear in publications that report the performance of a new algorithm that is applied to them. But these publications would not have appeared unless the algorithm performed well on most of the problems introduced. Problems that existing algorithms are adept at solving therefore have a selective advantage.

A similar process leads to a biased evolution of algorithms as well as problems. Once a set of canonical problems has become accepted, new methods that have strengths complementary to those of the old ones are at a disadvantage on the accepted problem sets. They are less likely to be judged successful by their authors and less likely to be published. So algorithms that excel on the canon have a selective advantage. The tail wags the dog as problems begin to design algorithms.

There is not to impugn in the slightest the integrity of those who collect and use benchmark problems. Rather, we are all victims of a double-edged evolutionary process that favors a narrow selection of problems and algorithms.

Even if this tendency could be corrected, other difficulties would remain. Nearly every problem set inspires complaints about its bias and limited scope. Problems from certain applications are always favored and others are always neglected. Worse than this, it is unclear that we would even be able to recognize a representative problem set if we had one. It is rare that anyone has the range of access to problems, many of which are proprietary, that is necessary to make such a judgment, and new problems constantly emerge.

Perhaps the most damaging outcome of competitive testing was mentioned at the outset: its failure to yield insight into the performance of algorithms. When algorithms compete, they are packed with the cleverest devices their authors can concoct and therefore differ in many respects. It is usually impossible to discern which of these devices are responsible for differences in performance.

The problem is compounded when one compares performance with a commercial code, which is often necessary if one is to convince the research community of the viability of a new method. The commercial package may contain any number of features that improve performance, some of which are typically kept secret by the vendor. The scientific value of such comparisons is practically nil.

As already noted, the most informative testing usually takes place during the algorithm's initial design phase. There tend to be a number of implementation decisions that are not determined by analysis and must be made on an empirical basis. A few trial runs are made to decide the issue. If these trials were conducted with the same care as the competitive trials (which, admittedly, are themselves often inadequate), much more would be learned.

Finally, competitive testing diverts time and energy from more productive experimentation. Writing efficient code requires a substantial time investment because a low-level language such as C must be used, time profiles must repeatedly be run to identify inefficiencies, and the code must be polished again and again to root them out. The investigator must also train himself in the art of efficient coding or else spend his research money on assistants who know the art.

Not only does competitive testing sacrifice what would otherwise be the relative ease of algorithmic experimentation, it surrenders its potential independence. Experimental projects in other fields must typically await funding and therefore approval from funding agencies or industry sources. A lone experimenter in algorithms, by contrast, can try out his ideas at night on a work station when their value is evident only to him or her. This opens the door to a greater variety of creative investigation, provided of course that these nights are not spent shaving off machine cycles.

2 A More Scientific Alternative

None of the foregoing is meant to suggest that efficient code should not be written. On the contrary, fast code is one of the goals of computational testing. But this goal is better served if tests are first designed to develop the kind of knowledge that permits effective code to be engineered. It would be absurd to ground structural engineering, for instance, solely on a series of competitions in which, say, entire bridges are built, each incorporating everything the designer knows about how to obtain the strongest bridge for the least cost. This would allow for only a few experiments a year, and it would be hard to extract useful knowledge from the experiments. But this is not unlike the current situation in algorithmic experimentation. Structural engineers must rely at least partly on knowledge that is obtained in controlled laboratory experiments (regarding properties of materials, etc.), and it is no different with software engineers.

Scientific testing of algorithms can be illustrated by some recent work on the satisfiability problem of propositional logic. The satisfiability problem asks, for a given set of logical formulas, whether truth values can be assigned to the variables in them so as to make all of the formulas true. For instance, the set of formulas

$$\begin{array}{l} x_1 \text{ or } x_2 \\ x_1 \text{ or not-}x_2 \\ \text{not-}x_1 \text{ or } x_2 \\ \text{not-}x_1 \text{ or not-}x_2 \end{array}$$

is not satisfiable, because one of them is false no matter what truth values are assigned to the variables x_1 and x_2. (We assume that all formulas have the form shown; i.e., they consist of variables or their negations joined by or's.)

At the moment some of the most effective algorithms for checking satisfiability use a simple branching scheme [2, 8, 16]. A variable x_j is set to true and then to false to create subproblems at two successor nodes of the root node of a search tree. When the truth value of x_j is fixed, the problem can normally be simplified. For instance, if x_j is set to true, formulas containing the term x_j are deleted because they are satisfied, and occurrences of not-x_j are deleted from the remaining formulas. This may create single-term formulas that again fix variables, and if so the process is repeated. If the last term is removed from a formula, the formula is falsified and the search must backtrack. If all formulas are satisfied, the search stops with a solution. Otherwise the search branches on another variable and continues in depth-first fashion.

A key to the success of this algorithm appears to be the *branching rule* it uses—that is, the rule that selects which variable x_j to branch on at a node, and which branch to explore first. This is a hypothesis that can be tested empirically.

The most prevalent style of experimentation on satisfiability algorithms, however, does not test this or any other hypothesis in a definitive manner. The style is essentially competitive, perhaps best exemplified by an outright competition held in 1992 [2]. More typical are activities like the Second DIMACS Challenge [16], which invited participants to submit satisfiability and other codes to be tested on a suite of problems. The DIMACS challenges have been highly beneficial, not least because they have stimulated interest in responsible computational testing and helped to bring about some of the improvements we are beginning to see in this area. But the codes that are compared in this sort of activity differ in many respects, because each participant incorporates his or her own best ideas. Again it is hard to infer why some are better than others, and doubts about the benchmark problems further cloud the results.

The proper way to test the branching rule hypothesis is to test algorithms that are the same except for the branching rule, as was done to a limited extent in [8]. This raises the further question, however, as to why some branching rules are better than others. A later study [11] considered two hypotheses: a) that better branching rules try to maximize the probability that subproblems are satisfiable, and b) that better branching rules simplify the subproblems as much as possible (by deleting formulas and terms). Two models were constructed to estimate the probability of satisfiability for hypothesis (a). Neither issued in theorems but predicted that certain rules would perform better than others. The predictions were soundly refuted by experiment, and hypothesis (a) was rejected. A Markov chain model was built for hypothesis (b) to estimate the degree to which branching on a given variable would simplify the subproblem, and its predictions were consistent with experiment. This exercise seems to take a first step toward understanding why good branching rules work.

By conventional norms, this study makes no contribution, because its best computation times for branching rules are less than some reported in the literature. But this assessment misses the point. The rules were deliberately implemented in plain satisfiability codes so as to isolate their effect. Codes reported in the literature contain a number of devices that accelerate their performance but obscure the impact of branching rules. Beyond this, the study was not intended to put forward a state-of-the-art branching rule and demonstrate its superiority to others in the literature; it was intended to deepen our understanding of branching rule behavior in a way that might ultimately lead to better rules.

To illustrate the construction of a controlled experiment, suppose that we wish to investigate how problem characteristics influence the behavior of branching rules (an issue not addressed in [11]). Benchmark problems are inadequate, because they differ in so many respects that it is rarely evident why some are harder than others, and they may yet fail to vary over parameters that are key determinants of performance. It is better to generate problems in a controlled fashion.

One type of experimental design (a "factorial design") begins with a list of n *factors* that could affect performance—perhaps problem size, density, existence of a solution, closeness to 'renamable Horn' [1, 3, 4], etc. Each factor i has several *levels* $k_i = 1, \ldots, m_i$, corresponding to different problem sizes, densities, etc. The levels need not correspond to values on a scale, as for instance if the factor is 'problem structure' and the 'levels' denote various types of structure. A sizable problem set is generated for each cell $(k_1, \ldots, k_n)$ of an n-dimensional array, and average performance is measured for each set. Statistical analysis (such as analysis of variance or nonparametric tests) can now check whether factor 1, for instance, has a significant effect on performance when the remaining factors are held constant at any given set of levels $(k_2, \ldots, k_n)$. It is also possible to measure interactions among factors. See [11, 14] for details.

This scheme requires random generation of problems, but it bears scant resemblance to traditional random generation. The goal is not to generate realistic problems, which random generation cannot do, but to generate several problem sets, each of which is homogeneous with respect to characteristics that are likely to affect performance.

This principle is again illustrated by recent work on the satisfiability problem. Several investigators have noted that random problems tend to be hard when the ratio of the number of formulas to the number of variables is close to a critical value ([5, 6, 7, 10, 12, 15], etc.). But this observation scarcely implies that one can predict the difficulty of a given problem by computing the ratio of formulas to variables. Random problems with a given ratio may differ along other dimensions that determine difficulty in practice.

This example has an additional subtlety that teaches an important lesson. In many experiments, nearly all problems that have the critical ratio are hard. This may suggest that other factors are unimportant and that there is no need to control for them. But some of the problem structures that occur in practice, and that substantially affect performance, may occur only with very low probability among random problems. This in fact seems to be the case, because practical problems with the same formula/variable ratio vary wildly in difficulty. It is therefore doubly important to generate problem sets that control for characteristics other than a high or low formula/variable ratio—not only to ensure that their effect is noticed, but even to ensure that they occur in the problems generated.

How can one tell which factors are important? There is no easy answer to this question. Much of the creativity of empirical scientists is manifested in hunches or intuition as to what explains a phenomenon. Insight may emerge from theoretical analysis or examination of experimental data for patterns. McGeoch [14] discusses some techniques for doing the latter in an algorithmic context.

3 What To Measure

Most computational experiments measure solution quality or running time. The former is unproblematic. The latter, however, is better suited to competitive than scientific testing.

Consider the satisfiability algorithms discussed earlier. Two theories were proposed to explain the effect of branching rules on the performance of branching algorithms. Both theories were based on predictions of the search tree size. So if one is interested in confirming or refuting the theories, as one should be in a scientific context, it makes sense to count the nodes in the tree, not to measure physical running time.

It is true that the ultimate goal is to minimize running time. But if the connection between branching rules and tree size is understood, one can combine this with estimates of the amount of processing at each node to predict the running time. The latter estimates can be based on further empirical work.

One might object that the tree size and running time per node may be related. It is possible, for instance, that more time is spent at each node when the tree size is small (other things equal) in order to carry out the greater degree of problem simplification at each node that is necessary to produce a small tree. This might suggest that one should measure total running time to capture the combined effect. But if the goal is to explain the combined effect of tree size and per-node computation, one should formulate and test models that explain this combined effect! Then one should measure what is predicted by the models, which is more likely to be the number of terms and formulas deleted rather than computation time. Computation time can then predicted on the basis of the average time required to delete a term or a formula, which naturally depends on the data structure, machine, etc.—factors that can be investigated independently of the branching rule.

The principle is simple: measure what is predicted by the model, and nothing more. McGeoch [14] proposes an interesting framework for thinking about how to do this. She suggests that an algorithm be viewed as an abstraction of a code, rather than viewing a code as an implementation of an algorithm. In other words, the code is the phenomenon, and an algorithm is a simplified model of what happens in the code. It may omit any mention of data structures and machine architecture, for instance.

McGeoch goes on to suggest that an algorithm be *simulated* to explore its behavior. The code that simulates the algorithm is entirely distinct from the polished code (perhaps yet to be developed) that the algorithm models. For instance, if one is interested in a node count for satisfiability algorithms, one need only write enough code to generate the nodes of a search tree. The data structure, machine, etc., are irrelevant so long as they do not affect the node count. In particular, they may be slow and inefficient.

If I may elaborate somewhat on McGeoch's idea, it is reminiscent of an astronomer's simulation of, say, galactic evolution. The astronomer may believe that initial conditions A give rise to spiral galaxies, and initial conditions B give rise to spherical galaxies. One way to check this empirically is to simulate the motions of the stars that result from initial conditions A or B and gravitational attraction. The algorithm simulated is an abstraction or a simplified model of the phenomenon, because it omits the effect of interstellar matter, etc. Nonetheless it improves our understanding. The astronomer is hardly bothered by the fact that the running time is different from that of the real phenomenon!

In the algorithmic context, one simulates only as much detail as the algorithmic model specifies. If one really has a model that predicts physical running time and therefore needs to measure it, then one would simulate the machine itself and count machine cycles. Even in this case it makes no sense to measure the actual running time of the simulation, which might run on any number of machines, fast and slow.

In practice, one would begin with a high-level algorithm in which the operation of various subroutines is left unspecified. A simulation might measure only the number of subroutine calls. This would provide only a crude prediction of a how a finished code would behave. Then one would begin to flesh out the subroutines. Whenever possible, a subroutine should be modeled independently of the calling routine. As one approaches the level of data structures, the models predict more accurately the performance of a finished code. Furthermore, the insight gained along the way permits one to write better algorithms for the subroutines and eventually a fast commercial code.

4 The Benefits of Scientific Testing

Scientific testing solves or alleviates all of the problems associated with competitive testing that were mentioned earlier.

Consider the issue of fair comparison of algorithms. To begin with, machine speeds are completely irrelevant. If one simulates what is to be measured, as recommended above, the results are machine independent. Likewise it makes no difference which data structure is used, unless of course the model being tested actually specifies the data structure, so that its operation is explicitly simulated. In the latter case one would simply implement the data structure specified. Coding skill is irrelevant—provided one has enough skill to simulate the algorithm correctly! The issue of how much to tune an algorithm becomes moot, because the parameter settings are among the factors one would investigate experimentally. That is, rather than agonize over what are the best parameter settings, one runs controlled experiments in which many different parameter settings are used, precisely in order to understand their effect on performance. Finally, established algorithms implemented in highly developed codes have no advantage. The polished code is not even used. Instead one simulates the relevant aspects of the algorithm in a rough-and-ready implementation, perhaps using a high-level language such as Prolog, Mathematica, Maple, or a simulation language. In short, the problem of fair comparison becomes a nonproblem.

As already discussed, the issue of how to choose problem sets is completely transformed. Rather than try to assemble problems that are representative of reality, one concocts problems so as to control for parameters that may affect performance. The problems are not only likely to be atypical but deliberately so, in order to isolate the effect of various characteristics. Admittedly, the choice of which factors to control for is far from trivial and may demand considerable insight as well as trial and error. But it is a problem that creative scientists deal with successfully in other disciplines, whereas the task of choosing representative benchmark problems seems to confound all efforts. Furthermore, it is a problem that algorithmists *ought* to struggle with, because it goes to the heart of what empirical science is all about.

Once the necessity of relying on benchmark problems is obviated, the accompanying evils evaporate, including the unhealthy symbiosis between problems and algorithms described earlier. Benchmarks will continue to play a role; the temptation to match a finished algorithm against the benchmarks will be irresistible. But they should play precisely this benchmarking role for finished products and not an experimental role in the scientific study of algorithms. It is a matter of distinguishing research and development: benchmarks are appropriate for development, but controlled experimentation is needed for research.

This emphasis on scientific testing requires a new set of norms for research. It asks that experimental results be evaluated on the basis of whether they contribute to our understanding, rather than whether they show that the author's algorithm can win a race with the state of the art. It asks scholarly journals to publish studies of algorithms that are miserable failures when their failure enlightens us.

If this seems inappropriate, it is perhaps because we have in fact confused research and development. We have saddled algorithmic researchers with the burden of exhibiting faster and better algorithms in each paper, a charge more suited to software houses, while expecting them to advance our knowledge of algorithms at the same time. I believe that when researchers are relieved of this dual responsibility and freed to conduct experiments for the sake of science, research and development alike will benefit.

References

[1] Aspvall, B., Recognizing disguised NR(1) instances of the satisfiability problem, *Journal of Algorithms* **1** (1980) 97-103.

[2] Böhm, H., Report on a SAT competition, Technical report no. 110, Universität Paderborn, Germany (1992).

[3] Chandru, V., C. R. Coullard, P. L. Hammer, M. Montanez and X. Sun, On renamable Horn and generalized Horn functions. In *Annals*

of Mathematics and Artificial Intelligence **1**. J. C. Baltzer AG, Basel (1990).

[4] Chandru, V., and J. N. Hooker, Detecting extended Horn structure in propositional logic, *Information Processing Letters* **42** (1992) 109-111.

[5] Cheeseman, P., B. Kanefsky and W. M. Taylor, Where the really hard problems are. In *Proceedings of the International Joint Conference on Artificial Intelligence, ICAI91* (1991) 331-337.

[6] Crawford, J. M., and L. D. Auton, Experimental results on the crossover point in satisfiability problems. In *Proceedings of the Eleventh National Conference on Artificial Intelligence, AAAI93* (1993) 21-27.

[7] Gent, I. P., and T. Walsh, The SAT phase transition. In A. G. Cohn, ed., *Proceedings of the Eleventh European Conference on Artificial Intelligence, ECAI94*, Wiley (1994) 105-109.

[8] Harche, F., J. N. Hooker and G. L. Thompson, A computational study of satisfiability algorithms for propositional logic, to appear in *ORSA Journal on Computing.*

[9] Hooker, J. N., Needed: An empirical science of algorithms, *Operations Research* **42** (1994) 201-212.

[10] Hooker, J. N., and C. Fedjki, Branch-and-cut solution of inference problems in propositional logic, *Annals of Mathematics and Artificial Intelligence* **1** (1990) 123-139.

[11] Hooker, J. N., and V. Vinay, Branching rules for satisfiability, to appear in *Journal of Automated Reasoning.*

[12] Larrabee, T., and Y. Tsujii, Evidence for a satisfiability threshold for random 3cnf formulas. In Hirsh *et al.*, eds., *Proceedings of the Spring Symposium on Artificial Intelligence and NP-Hard Problems*, Stanford, CA (1993) 112-118.

[13] Lustig, I. J., R. E. Marsten and D. F. Shanno, Interior point methods for linear programming: Computational state of the art, *ORSA Journal on Computing* **6** (1994) 1-14.

[14] McGeoch, C. C., Toward an experimental method for algorithm simulation, to appear in *ORSA Journal on Computing.*

[15] Mitchell, D., B. Selman and H. Levesque, Hard and easy distributions of SAT problems. In *Proceedings, Tenth National Conference on Artificial Intelligence, AAAI92*, MIT Press (1992) 459-465.

[16] Trick, M., and D. S. Johnson, eds., *Second DIMACS Challenge: Cliques, Coloring and Satisfiability*, Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society (1995).