The Logic of Benchmarking: A Case Against State-of-the-Art Performance

Wheeler Ruml
Department of Computer Science
University of New Hampshire
Durham, NH 03824 USA
ruml at cs.unh.edu

Abstract

This note marshals arguments for three points. First, it is better to test on small benchmark instances than to solve the largest possible ones. This eases replication and allows a more diverse set of instances to be tested. There are few conclusions that one can draw from running on large benchmarks that can't also be drawn from running on small benchmarks. Second, experimental evaluation should focus on understanding algorithm behavior and forming predictive models, rather than on achieving state-of-the-art performance on toy problems. Third, it is more important to develop search techniques that are robust across multiple domains than ones that only give state-of-the-art performance in a single domain. Robust techniques are more likely be useful to others.

Small Benchmarks Are Better

I have seen several reviewers claim that experimental evaluations must use problem instances that are large enough to take more than x time to solve, where x has varied from a few seconds to many minutes. The fact that the stated values for x have varied over two orders of magnitude is perhaps one indication that the criterion has no firm basis, but let's consider it in more depth. I believe that it represents a serious problem: I have seen a paper (not from my group) proposing a novel and interesting algorithm that soundly beat the state-of-the-art by orders of magnitude rejected (in part) because the benchmarks in the paper took only a few milliseconds to run. As Pedersen (2008) notes, reproducibility is fundamental to science, so why wouldn't we prefer small benchmarks that are easier to run? What could justify such a criterion?

The most obvious concern is that a small running time might be subject to significant measurement error—perhaps the hardware or OS might not be able to measure small times precisely. This is certainly not a problem under Linux, which reports time with sub-millisecond resolution, but timers under Windows can be limited to hundredths of seconds. If there is concern that the hardware or OS does not provide sufficient timing resolution, it is always possible to time algorithm execution over many small benchmarks and measure the sum, which is more likely to exceed the OS timer resolution. Furthermore, my experience with CPU-bound processes, like search algorithms, under Linux on x86 is that there is little difficulty in *repeatably* measuring sub-second times. That is to say, the same program run again on the same input reports a running time within a few percent.

Mytkowicz et al. (2009) report a study of the SPECint benchmarks in which they varied the size of the Unix shell environment (by increasing the length of a string assigned to a variable) and the link order of object files (by varying the order in which files were listed on the command line). This caused variations in the memory layout of the program, leading to variations in CPU time of almost $\pm 10\%$ for the same source code. To combat this variation, they recommend two practices. The first is to use a diverse set of benchmarks, to improve the odds that the benchmarks will respond differently to a particular bias introduced by the test environment and hence factor it out. The second is causal analysis: isolate what causes a performance improvement and leave everything else unchanged, hopefully leaving the uncontrolled biases the same between the two conditions. Note that nothing in this disturbing anecdote relates to benchmark size. Both short- and long-running benchmarks were affected, and the recommendation is to increase benchmark diversity, which requires more benchmarks, which must therefore be smaller (in order to fit into the same PhD program or yearly conference timescale). Large benchmarks therefore inhibit correction of measurement bias. (Of course, the easiest route to correct conclusions is to ensure that one's new technique results in more than a 20% speed-up!)

Sources of variation may have less to do with the operating system and more to do with the algorithm implementation or library routines. McGeoch (1996) relates multiple cases in which changing random number generators, fixing seemingly minor bugs, or innocent reimplementation resulted in statistically different results. She recommends that key results be obtained using a completely disjoint reimplementation. Note that this requires running the same benchmarks multiple times. This is infeasible when using large benchmarks (or when requiring extensive implementation tuning), giving a distinct advantage to small benchmarks.

Perhaps the best argument for running on large instances is that small problems might not reveal certain aspects of behavior, that asymptotic performance might not yet be clear or that some scaling effect might not be apparent at mod-

est problem sizes. Eppinger (1983) presents a case in which both theoretical analysis and initial empirical results agreed that a certain behavior would occur, but runs on much larger instances revealed evidence of the entirely opposite behavior (eventually resulting in a more refined theoretical analysis). (McGeoch (1996) cites two additional examples.) However, running on large instances can never preclude the possibility that running on even larger instances would uncover contradictory evidence. One can be no more logically certain when extrapolating from large benchmarks than from small ones. Both exercises must be treated as speculative, leaving large and small benchmarks equivalent in this respect. Overall, there seem to be few firm conclusions that one can draw from running on large benchmarks that couldn't also be drawn from running on small benchmarks. Given that smaller benchmarks allow a wider variety of problems and algorithm variations to be tested and a greater understanding of behavior to be developed, the choice is clear.

One final motivation for larger benchmarks is the concern that small problems might not be realistic or relevant, presumably with respect to commercial applications. The assumption seems to be that 'real-world' necessarily equals 'large'. I have personal experience with two industrial applications. In one case each problem instance had to be solved within 270 milliseconds and in the other each instance had to be solved within one minute. The two most celebrated recent uses of heuristic search, in game pathfinding (Nathan Sturtevant's work on Dragon Age: Origins) and in robotic motion planning (Max Likhachev's and David Ferguson's work on the DARPA Urban Challenge), also both require fast solving (in milliseconds and seconds, respectively). Such problems may have intricate complexities, but the instantiated portion of the search space certainly cannot be called large.

Benchmarking is for Understanding

As Hooker (1994, 1995) eloquently laments, it is easy to be drawn into treating benchmarks as a competition rather than as science (advancing understanding). I have seen a SoCS reviewer state "I believe that any publishable paper should demonstrate at least one domain on which the authors' algorithm outperforms the previous state of the art." But it is important to keep in mind that very few people really care about solving our toy puzzles. Real applications tend to be messy and complex, exactly the opposite of the simplicity we desire in a benchmark. Benchmarks are for helping us understand how algorithms behave. As Knuth (1996) argues, toy problems are important because they help us hone general techniques that will be broadly useful. Surpassing the current state-of-the-art in performance in a toy domain is certainly one indication that an idea has merit, but it is understanding rather than bragging rights that should be the ultimate goal of a paper. We seek algorithms or algorithmic techniques that are well-enough understood that we are comfortable recommending them to our colleagues in industry who are facing the real problems. Developing this predictive understanding is implicitly discouraged by emphasizing state-of-the-art performance.

General-purpose Techniques are Better

If we take as the goal of academic research on heuristic search to supply industry with a toolkit of well-understood algorithms and techniques, then perhaps the most valuable contribution would be methods that perform reliably across many different domains, rather than extremely well on some and extremely poorly on others. Achieving state-of-the-art performance on problem X through tricks that only apply to X is only interesting to those who face problem X. If X is a toy problem, that will be very few people. If the community concerned itself only with that style of work, it would risk, in the words of one journal reviewer "the impression that heuristic search has become a community that is only interested in peculiarities of the 15-puzzle, talks only to itself, and has no relevance to broader AI or CS research." A very generic method that will perform reasonably well on any problem that looks vaguely like X or Y or Z will find a much broader audience.

To test such a method, it is not necessary to implement the very latest state-of-the-art heuristic function for a single toy benchmark domain. It is critical that the method is tested in a wide variety of search spaces, but performance on any single particular problem is beside the point. Methods with special requirements, such as integer costs, real costs with a large lowest common denominator, or undirected graph edges would rate low in this endeavor.

Such a 'method of first resort' could become the Sawzall of search. My personal view is that the popularity of genetic algorithms and local search comes in large part from the broad applicability of those methods, rather than from superior performance. What of comparable value can heuristic search offer? We will have succeeded when we see multiple industrial sponsors at SoCS, eager for our ideas.

Acknowledgments

I am grateful for helpful comments from Jordan Thayer, Nathan Sturtevant and Ariel Felner and for support from the U.S. NSF (grant IIS-0812141) and the DARPA CSSG program (grant HR0011-09-1-0021).

References

Eppinger, J. L. 1983. An empirical study of insertion and deletion in binary search trees. *Comm. of the ACM* 26(9):663–669.

Hooker, J. N. 1994. Needed: An empirical science of algorithms. *Operations Research* 42(2):201–212.

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

Knuth, D. E. 1996. Are toy problems useful? In *Selected Papers on Computer Science*. CSLI. chapter 10, 169–183. Originally published in *Popular Computing*, 1977.

McGeoch, C. C. 1996. Toward an experimental method for algorithm simulation. *INFORMS Journal on Computing* 8(1):1–15.

Mytkowicz, T.; Diwan, A.; Hauswirth, M.; and Sweeney, P. F. 2009. Producing wrong data without doing anything obviously wrong! In *Proceedings of ASPLOS*, 265–276. ACM.

Pedersen, T. 2008. Empiricism is not a matter of faith. *Computational Linguistics* 34(3):465–470.