A Theoretician's Guide to the Experimental Analysis of Algorithms

David S. Johnson
AT&T Labs – Research
http://www.research.att.com/~dsj/

November 25, 2001

Abstract

This paper presents an informal discussion of issues that arise when one attempts to analyze algorithms experimentally. It is based on lessons learned by the author over the course of more than a decade of experimentation, survey paper writing, refereeing, and lively discussions with other experimentalists. Although written from the perspective of a theoretical computer scientist, it is intended to be of use to researchers from all fields who want to study algorithms experimentally. It has two goals: first, to provide a useful guide to new experimentalists about how such work can best be performed and written up, and second, to challenge current researchers to think about whether their own work might be improved from a scientific point of view. With the latter purpose in mind, the author hopes that at least a few of his recommendations will be considered controversial.

DRAFT of a paper that appeared in the Proceedings of the 5th and 6th DIMACS Implementation Challenges, Goldwasser, Johnson, and McGeoch, (eds), American Mathematical Society, 2002, 215-250.

Introduction

It is common these days to distinguish between three different approaches to analyzing algorithms: worst-case analysis, average-case analysis, and experimental analysis. The above ordering of the approaches is the one that most theoretical computer scientists would choose, with experimental analysis treated almost as an afterthought. Indeed, until recently experimental analysis of algorithms has been almost invisible in the theoretical computer science literature, although experimental work dominates algorithmic research in most other areas of computer science and related fields such as operations research.

The past emphasis among theoreticians on the more rigorous and theoretical modes of analysis is of course to be expected. Moreover, it has strong justifications. To this day it is difficult to draw useful extrapolations from experimental studies. Indeed, it was the lack of scientific rigor in early experimental work that led Knuth and other researchers in the 1960's to emphasize worst- and average-case analysis and the more general conclusions they could provide, especially with respect to asymptotic behavior. The benefits of this more mathematical approach have been many, not only in added understanding of old algorithms but in the invention of new algorithms and data structures, ones that have served us well as faster machines and larger memories have allowed us to attack larger problem instances.

Recently, however, there has been an upswing in interest in experimental work in the theoretical computer science community. This is not because theorists have suddenly discovered a love of programming (although some have) or because they are necessarily good at it (although some are), but because of a growing recognition that theoretical results cannot tell the full story about real-world algorithmic performance. Encouragement for experimentation has come both from those like myself who are already actively experimenting and from funding agencies who view experimentation as providing a pathway from theory into practice. As a result, we now have several relatively new forums that include both experimental and theoretical work or highlight hybrids of the two. These include the ACM Journal on Experimental Algorithmics, the ACM-SIAM Symposium on Discrete Algorithms (SODA), the European Symposium on Algorithms (ESA), the Workshop on Algorithm Engineering (WAE), and the Workshop on Algorithm Engineering and Experimentation (ALENEX).

These are in addition to traditional forums for experimental work such as those in the Operations Research, Mathematical Programming, Artificial Intelligence, and Scientific Computing communities. What, if any, is the value added by involving theoretical computer scientists with experimentation? One very real possibility is that experience with real-world computing may in turn reinvigorate our theoretical research agenda by suggesting new questions to study and establishing connections to applications. In this paper, however, I am interested in influences that go in the other direction. How can a background in theoretical analysis of algorithms help in doing experimentation? My hope, strongly supported by work of such researchers as Bentley, Goldberg, and others [Ben92, CGR94, FJMO95] is that the theoretician's concern for asymptotics, generality, and understanding can help drive a more scientific approach to experimentation. It is this scientific bias that I will be stressing in this paper, while also providing more general advice for would-be experimentalists.

Unfortunately, as many researchers have already discovered, the field of experimental analysis is fraught with pitfalls. In many ways, the implementation of an algorithm is the easy part. The hard part is successfully using that implementation to produce meaningful and valuable (and publishable!) research results. For the purposes of argument, there are perhaps four basic reasons why it might be worthwhile to implement an algorithm, each giving rise to a different type of paper (although many papers end up being hybrids of more than one type).

- To use the code in a particular application. This typically gives rise to an *application paper*, whose purpose is to describe the impact of the algorithm in that context. (This includes much of what is called *experimental mathematics*, where the "application" is the generation and proof of mathematical conjectures and we are primarily interested in the output of the algorithm, not its efficiency.)

- To provide evidence of the superiority of your algorithmic ideas. This often gives rise to what I will call a *horse race paper*, where results are published for standard benchmark instances to illustrate the desirability of this algorithm in comparison to previous competitors.

- To better understand the strengths, weaknesses, and operation of interesting algorithmic ideas in practice. This is the motivation behind what I will call an *experimental analysis paper*.

- To generate conjectures about the average-case behavior of algorithms under specific instance distributions where direct probabilistic analysis is too hard. This leads to what I will call an *experimental average-case paper*.

Although much of what I have to say will be applicable to all four types of paper, in what follows I will concentrate mainly on the third type, which is in a sense the most general and is also a type with which I have much experience, both as an author and as a reader, referee, editor, and scholar (i.e., survey paper writer). In these latter roles I have had the opportunity to read widely in the experimental literature, with special emphasis on papers about the Traveling Salesman Problem (TSP), which I surveyed in [JM97]. I have unfortunately seen many more papers that are seriously defective than ones that set good examples. Even fairly good papers typically receive referee's reports suggesting significant revisions, often involving substantial additional experiments. In this note I shall attempt to draw on my experience by articulating a set of general principles applicable to experimental papers and explaining how and how not to satisfy them. The presentation will be organized in terms of ten basic (and overlapping) principles that I believe should govern the writing of experimental papers:

1. Perform newsworthy experiments.
2. Tie your paper to the literature.
3. Use instance testbeds that can support general conclusions.
4. Use efficient and effective experimental designs.
5. Use reasonably efficient implementations.
6. Ensure reproducibility.
7. Ensure comparability.
8. Report the full story.
9. Draw well-justified conclusions and look for explanations.
10. Present your data in informative ways.

The remainder of this paper will discuss each of these principles in turn, explaining what I mean by them and elaborating on their implications. Along the way I will highlight "Pitfalls" (temptations and practices that can lead experimenters into substantial wastes of time) and my personal "Pet Peeves" (literally, "favorite annoyances" – common experimental and paper-writing practices that seem to me to be particularly misguided). Some of these practices may seem so obviously bad as to be not worth mentioning, but I have seen each of them more than once, and so they still seem worth an explicit discussion. Others of the practices I list as Pet Peeves are in such common use that my labeling them as "misguided" may come as a surprise, although I will argue that they violate one or more of the above principles. In some cases I will also highlight "Suggestions" for avoiding a given Pitfall or Pet Peeve, and cite papers where the suggested practices are exploited/explained. Specific citations of bad behavior will for the most part be omitted to spare the guilty, myself included. My view of these issues has evolved over the years. Even now it may be well-nigh impossible to write a paper that avoids all the Pet Peeves. An Appendix will collect all the Pitfalls, Pet Peeves, and Suggestions into separate lists for ease of reference.

Most of this paper concentrates on the study of algorithms for problems in the standard format where both instances and outputs are finite objects, and the key metrics are resource usage (typically time and memory) and (in the case of approximation algorithms) quality of the output solution. Many key computational problems do not fit this model, and in a short section at the end of the paper I will mention some of them and challenge the reader to think about how the ideas presented here might be adapted/extended to these other domains, as well as what other issues arise in them. I conclude with a final collection of warnings that I was unable to fit conveniently into the main body of the text.

As I have said, several of the principles enunciated in this paper may be somewhat controversial, at least as I elaborate them. Indeed, although there is much common agreement on what makes good experimental analysis of algorithms, certain aspects have been the subject of debate, such as the relevance of running time comparisons. For other points of view, see such recent papers as [BGK+95, CS00, Hoo93, McG96, MM99, RU01]. This particular paper was constructed using my earlier short draft [Joh96] as a template and adding in material I have developed in the process of giving talks expanding on that draft. The points of view expressed in this paper are my own, although they have been refined (and augmented) over the years in stimulating discussions with other researchers, including Richard Anderson, David Applegate, Jon Bentley, Andrew Goldberg, Cathy and Lyle McGeoch, Kurt Mehlhorn, Bernard Moret, Panos Pardalos, Mauricio Resende, and Gerhard Woeginger.

Section 1. Discussions of the Principles

In this, the main section of the paper, I discuss and illustrate each of the ten principles outlined in the introduction. Because these principles are so interlinked, the discussion of each will unavoidable refer to several of the others, but I have tried to avoid repetition as much as possible (although some of the points made are worth repeating).

Principle 1. Perform Newsworthy Experiments

This principle applies to all scientific papers: the results should be new and of interest and/or use to some reasonably-sized collection of readers. However, the standards for "interest" are more stringent for experimental algorithms papers.

For instance, it is much harder to justify experimental work on problems with no direct applications than it is to justify theoretical work on such problems. In theoretical work, researchers often choose to work on problems because of their amenability to analysis (and in the hope that they are representative enough that the lessons learned or proof techniques developed may be more widely applicable). In contrast, most problems are amenable to experimental analysis of some sort, and lessons learned from experimental analysis are typically highly problem- and instance-specific. Moreover, problems without applications do not have real-world instances, so the experimenter is left to invent (and justify) test data in a vacuum. And referees may well question the testing of code for algorithms that will never be used in practice.

A second dimension of "newsworthiness" has to do with the algorithms studied. A key question about any algorithm is whether it might be a useful alternative in practice. Since theoretical analysis (especially worst-case analysis) often cannot be relied on for an answer, one can justify studying a wide variety of algorithms, at least to the point where the question of practicality is answered (assuming there was at least some plausible argument in favor of practicality in the first place). When the answer is a definitive *No*, however, the value of writing papers about the algorithm is limited, although it is not necessarily eliminated. Consider for example the case of the *dominated* algorithm, i.e., one that is always slower than some already well-studied competitor (and, in the case of approximation algorithms, never produces better solutions).

PITFALL 1. *Dealing with dominated algorithms.*

Suppose you spend much time implementing and testing an algorithm that turns out to be dominated. All may not be lost. Indeed, much of the experimental literature is devoted to dominated algorithms. In many cases, however, the fact that the algorithm was dominated was not known at the time the paper about it was written (or at least was not known to the author or referees...)

PET PEEVE 1. *Authors and referees who don't do their homework.*

Particularly annoying are those authors who do cite the relevant references but clearly haven't read the parts of them that are relevant to their experiments.

What do you do if you *know* (or discover) that your algorithm is dominated? If the algorithm is newly invented and has not yet been written about, you are pretty much out of luck. Few people are likely to be interested in the fact that a previously unknown algorithm is bad. If it is a known algorithm, and assuming it has not been adequately studied before, there are several possible situations:

- The algorithm is already in widespread use or the algorithms that dominate it are so complicated that they are not soon likely to enter into widespread use. A detailed study of the algorithm's performance and how much it gives up over the more sophisticated approaches might well be of interest.

- The algorithm in question embodies a general approach applicable to many problem domains or provides a mechanism for studying the performance of data structures or other algorithmic constructs, so that studying how best to adapt the underlying approach to the current domain may have more general consequences. This argument can for example be viewed as a possible justification for some of the many papers that have been published about dominated metaheuristic algorithms for the TSP (see [JM97]).

- The algorithm is known but already expected to behave poorly, although this had not been verified (and the previous options do not apply). In this case you are again out of luck, although the negative result might be worth including in a more general paper that talked about undominated algorithms as well.
- The algorithm is of sufficient independent interest (either because of promising theoretical results or widespread publicity) that the fact that it *is* dominated is unexpected. This might be worth a note on its own, although again it would be better as part of a more comprehensive study.

In this last situation, the fact that the algorithm is dominated is itself newsworthy. Note, however, that domination is not always easy to establish conclusively, given that running times can depend strongly on implementations or machine/operating system interactions. In many cases simply getting within a factor of two of another algorithm's running time may be enough to claim competitiveness. Moreover, you are facing the difficult task of proving a negative experimentally: Even if you demonstrate great differences in performance, you still must convince the readers that you have not missed some important instance class for which the supposedly-dominated algorithm outperforms its competitors. You also must convince readers that you have not simply mis-implemented the algorithm. For example, several early papers on simulated annealing claimed that it was dominated by other approaches within the domains in question. The implementations described in these papers, however, used parameter settings that severely limited the computation time, whereas subsequent experiments showed that allowing the annealing process to take a (very) long time enabled it to perform very well in these domains, even when compared to other long-running algorithms. Thus the original papers did not show that simulated annealing was dominated, just that their particular (bad) implementations of it were. For a more precisely defined algorithm than simulated annealing, convincing readers that you have an efficient and correct implementation may be easier, but still is an added burden, requiring extra measurements to identify unavoidable bottlenecks in the algorithm's operation.

A third dimension of newsworthiness has to do with the generality, relevance, and credibility of the results obtained and the conclusions derivable from them. One is of necessity limited to a finite number of tests, but for a paper to be interesting, the results should yield broader insights, either in terms of asymptotics or presumptions of robustness, a point on which we shall elaborate when discussing Principle 3 below. (This is one way in which an experimental paper aspires to higher goals than a horse race paper, which can settle for the simple conclusion that the tested algorithm produces the stated results on the set of benchmark instances.) Also, the conclusion that algorithm A outperforms algorithm B becomes more interesting (and more believable) when supported by measurements (operation counts, subroutine profiling, etc.) that show in detail *why* this happens.

Given the above, let us talk a little about strategies for getting newsworthy results from your experiments, and the pitfalls along the way.

PITFALL 2. *Devoting too much computation to the wrong questions.*

With computational cycles becoming so cheap these days, it is tempting not to worry too much about wasted computation time. However, "computation time" includes your own time spent setting up, running, and evaluating the experiments, not just the "user time" reported by the operating system. If you want your paper production process to be as efficient as your algorithms it pays to be more systematic. Examples of potential wastes of time include

- Excessive study of results for just one or two instances. This is a natural temptation when one has a parameterized algorithm that needs tuning. Typically, however, your computing time will be better spent testing a variety of instances in a systematic way.

- Running full experimental suites before you finish figuring out how to make your implementation efficient. You'll no doubt have to rerun the experiments with the new code anyway, and the later runs will probably take less time.

- Running full experimental suites before you've decided what data to collect (and hence before you've instrumented the code to report that data). Again, you'll probably have to rerun the experiments.

SUGGESTION 1. *Think before you compute.*

Some of the issues that need to be resolved before you begin a major data generation project include the following. What algorithmic phenomena do you want to study, and what are the questions you want your experiments to address? Will anyone besides yourself care about the answers, given the current state of the literature in the area? Have you implemented the algorithm correctly, incorporating all the features/variants you want to study and providing for the production of all the output data you'll need? What is an adequate set of test instances and runs to yield convincing answers to the questions being asked? Given current computer speeds and memory capacity, which instances are either too small to yield meaningful distinctions or too large to yield feasible running times?

Of course, some of this thinking presupposes that you have already done at least a little computation. In order to resolve issues like those mentioned, one often will have to perform exploratory experiments.

SUGGESTION 2. *Use exploratory experimentation to find good questions.*

By "exploratory experimentation" I mean the testing one does before beginning the formal data collection process for a computational study. This includes experiments performed while debugging and optimizing the code as well as pilot studies of various sorts. In addition to trying to confirm that the algorithm is competitive with previous approaches, one can be examining how performance varies with instance type and size, and profiling the code to determine where the computational bottlenecks are and what sorts of tradeoffs are involved in various program design and data structure choices. Of special interest are unexpected anomalies in performance, which if reproducible may merit further study. Assuming that this further study does not simply reveal that there were bugs in your code, it may suggest surprising insights into the way the algorithm performs. In my own research, I tend to take an iterative approach:

1. Spend the first half of your proposed experimentation time generating lots of data and looking for patterns and anomalies.

2. Based on the above, finalize the algorithm implementation, decide what the interesting questions are, and design and perform comprehensive experiments to investigate these.

3. Analyze the resulting data. If it fails to answer the questions or raises new ones, go back to Step 2.

This tends to lead to newsworthy results, although it has its own dangers:

PITFALL 3. *Getting into an endless loop in your experimentation.*

Although most experimental papers could be improved by more extensive experimentation, at some point you have to draw the line and leave the remaining questions for the infamous "future research" section of the paper. (In this, you must be prepared to resist not only your own temptation to keep going, but also requests from referees for additional but non-essential experiments.) A newsworthy paper that never gets finished does not provide much news.

At this point, having discussed the dangers of working on the wrong questions and described techniques for finding the right questions, let me say a bit about what some of those good questions might be and which sorts of answers to them might be newsworthy, given that the problems and algorithms studied pass the newsworthiness test. The following list is by no means exhaustive, but will at least cover a range of possibilities. A typical paper might address several such questions. In the following list, I for simplicity follow common practice and use "algorithm" as a shorthand for "your implementation of the algorithm." Most descriptions of algorithms in technical papers and texts omit technical details that an implementer must fill in, and these can affect performance. Thus our experiments typically give us direct information only about the implementations used. With extensive study we may be able to deduce something about the implementation-independent properties of an algorithm or the specific implementation details that need to be included in the description of an algorithm if we are to promote its best performance. Hence our first question:

1. How do implementation details, parameter settings, heuristics, and data structure choices affect the running time of the algorithm?

2. How does the running time of the algorithm scale with instance size and how does this depend on instance structure?

3. What algorithmic operation counts best help to explain running time?

4. What in practice are the computational bottlenecks of the algorithm, and how do they depend on instance size and structure? How does this differ from the predictions of worst-case analysis?

5. How is running time (and the running time growth rate) affected by machine architecture, and can detailed profiling help explain this?

6. Given that one is running on the same or similar instances and on a fixed machine, how predictable are running times/operation counts? (If there is wide variability, this might affect the usability of the algorithm.)

7. How does the algorithm's running time compare to those of its top competitors, how are these comparisons affected by instance size and structure or by machine architecture, and can the differences be explained in terms of operation counts?

8. What are the answers to the above questions when "running time" is replaced by "memory usage" or the usage of some other computational resource, assuming the algorithm is such that this usage isn't easily predictable from theory?

9. What are the answers to questions 1,2,6,7 when one deals with approximation algorithms and "running time" is replaced by "solution quality"? (Standards of comparison that can be used to measure solution quality will be covered in the discussion of Principle 6.)

10. Given a substantially new class of instances that you have identified, does it cause significant changes in the algorithmic behavior for previously-studied algorithms?

Let me conclude the discussion of newsworthiness by mentioning one other temptation that can lead an experimentalist astray.

> PITFALL 4. *Start by using randomly generated instances to evaluate the behavior of algorithms, but end up using algorithms to investigate the properties of randomly-generated instances.*

One example of this kind of behavior is the recent flurry of activity around random instances of 3SAT. Early experimenters looking for hard instances to test discovered that if one randomly generates m clauses on n variables, one obtains a transition when $m \sim 4.24n$: Instances with fewer clauses are highly likely to be satisfiable, instances with more clauses are most probably unsatisfiable, and instances with just about that many clauses tend to be difficult for all known algorithms to solve. This observation was useful to algorithm evaluators, as it gave a nice (if unrealistic) class of challenging test examples. It also suggested interesting mathematical questions about the precise nature of the "phase transition" and its asymptotic properties. Unfortunately, it has also had the effect of hijacking much of the experimental research on SATISFIABILITY algorithms to examining just this class of instances and their variants, even though it is not at all clear that such experiments will tell us anything about performance on the much more structured instances that tend to occur in real-world applications.

A second, perhaps more benign example is the experimental work of [JMR96, PM96] devoted to estimating the expected optima for random Euclidean instances of the TSP. For the TSP, random geometric instances like these do seem to provide a barometer for performance in practice. Tight bounds on expected optima may thus give us a useful metric for evaluating the quality of the solutions produced by approximation algorithms, at least for instances so large that good instance-specific bounds are not easy to obtain.

Of course, performing experiments in order to deduce properties of random instances can be justified on straight mathematical grounds, and may be of more intrinsic interest than the performance of yet another heuristic. However, this is "experimental mathematics" rather than "experimental analysis of algorithms," and so in that sense represents a diversion from the task with which we are concerned here.

Principle 2. Tie Your Paper to the Literature

One key component in establishing the newsworthiness of a paper is placing it in proper context with respect to the literature on the problem being studied. Indeed, before undertaking any experimental project, you should do your best to discover and thoroughly study the prior literature if it exists. This should go without saying, but surprisingly many papers have failed because of lack of diligence on this point. Moreover, knowing what has been done can not only save you from performing "uninteresting" experiments, it can also suggest what the interesting questions should be. What behavior needs explaining? What algorithms seem open to further improvement? What types of test instances have not yet been adequately studied?

When your problem does have a literature, there are obligations as well as opportunities, however. Assuming an earlier paper studied instances that are large enough to be considered interesting today, you should provide comparisons between your results and those in the earlier paper. Here is another case where the experimenter's job may be more difficult than the theoretician's. The potential difficulty is that direct comparisons may be problematic. Although the earlier paper may have dealt with the same algorithm and similar test instances, it may not have dealt with precisely the same test instances or with the same implementation of the algorithm, and it most likely did not provide running times on the same machine/operating system combination you are using today.

Ideally, if your goal is to provide comparisons to a previously-studied algorithm, you should obtain the code for the implementation of the algorithm that was used in the previous experiments, and report results for that code on your own machine (while verifying that those results are consistent with those reported for the code previously). If that is not possible, the next best option is to develop a comparable implementation of the previous algorithm which you can run on your own machine and which, based on comparisons with the previously-reported results, seems to be of comparable effectiveness. (This would have the added benefit of helping to confirm the algorithmic claims of the earlier paper.) In the unfortunate event that the two implementations do not appear to be of comparable effectiveness, for example when one appears to run in quadratic time and the other in linear time, this point should be noted and explanations provided if possible (and your implementation should be the linear time one; see Principle 5).

The least desirable option, but one to which one must occasionally resort when attempting to provide comparisons to a hard-to-implement or incompletely-specified algorithm from the literature, is simply to compare your results on your own machine and test instances to those reported in the previous paper. In some cases, where the previous experiments were performed on some unfamiliar architecture, this may be impossible. Typically, however, you can provide significantly accurate ballpark bounds on relative machine speeds to at least answer the question of whether running times are possibly competitive or whether one implementation is clearly faster than the other. Even such rough comparisons are better than nothing. Readers need to see how your paper fits into the literature, even if the fit is at best only approximate.

Principle 3. Use Instance Testbeds that Can Support General Conclusions

There are basically two types of test instances available to experimenters: particular instances from real-world applications (or pseudo-applications) and randomly generated instances. The former are typically found in standard repositories, such as `TSPLIB` [Rei81], but might come from private sources proprietary to the author. The latter are provided by instance generators that, given a seed and a list of instance parameters, such as size and perhaps other details of instance structure, produce a random instance consistent with those parameters. Note that random instances need not be structureless (like random graphs and random distance matrices), and indeed should preferably be structured in ways that reflect some aspects of real-world instances.

A key property for random instance generators is that they be able to generate instances of arbitrarily large size. This allows you to get a partial handle on the types of asymptotic questions typically addressed by theoretical analysis, although machine dependent effects such as cache limitations will typically make precise asymptotics impossible. What you *can* determine, however, is the biggest instance you can reasonably run on your machine, and some idea about how much larger the runnable instances could be on a next-generation machine with greater speed, memory, etc. (These can be key questions in practice.) This gives random instances an advantage in experimental studies, although one that can be substantially dissipated if there is no evidence that the random instance class says anything about what will happen for real-world instances.

PET PEEVE 2. *Concentration on unstructured random instances.*

Not only may unstructured random instances tell us little about real-world performance, they may actively mislead us as to the difficulty of the problem. For example, many papers on optimization algorithms for the asymmetric TSP concentrate on asymmetric distance matrices with entries chosen independently and randomly from small ranges such as $\{1, 2, \ldots, N\}$, where N is the number of cities. These instances are particularly easy since as N increases the optimal tour length is likely to equal the easily computed "Assignment Problem" lower bound. Thus there are authors who proudly proclaim the ability of their codes to find optimal solutions for instances of this sort with many thousands of cities, while the same codes have great difficulty with structured instances from `TSPLIB` containing 53 cities or less.

One can also get in trouble by concentrating solely on real-world instances, especially outdated ones:

PET PEEVE 3. *The millisecond testbed.*

Computer speeds have grown at such a rate that in many problem areas modern running times for old testbeds have shrunk to negligibility. I have seen more than one paper in which the maximum running time reported for any of the algorithms tested on any of the instances studied is a second or less. Despite this fact, the papers devote much effort to determining which algorithm is fastest. I would maintain that in most applications, if an algorithm takes a second or less, running time is probably irrelevant, and an algorithm that takes 0.01 seconds on your machine doesn't provide a significant advantage over one that takes the 0.1, even though it is 10 times faster. The exception might be real-time computing applications (usually not relevant to the applications in question) or the need to solve thousands of instances quickly (also usually not relevant). One might also argue that that factor of 10 *would* make a difference for larger instances, but in that case one should test such larger instances to confirm that the advantage persists as instance size grows.

PET PEEVE 4. *The already-solved testbed.*

This is a related, perhaps more philosophical objection. When you are evaluating approximation algorithms, you need some standard against which to measure the solutions found, and the most desirable standard is of course the optimal solution value. Thus a natural temptation is to restrict your testing to instances for which the optimal value is known. However, one of the major reasons for resorting to approximation algorithms is to handle cases where it is too hard to find the optimal values, so restricting your tests to instances for which optima are known essentially ignores the domain for which the algorithms were intended. One might argue that the approximation algorithm is much faster than the optimization algorithm, and hence you are examining the trade-off between running time and solution quality. However, most papers that restrict attention to instances for which optima are known simply take the optimum values from the literature and do not seriously address the question of relative running times. Moreover, typically the only nontrivial instances for which optimum solutions are known are relatively small. Thus, as with the previous Pet Peeve, restricting to such instances doesn't allow you to address questions about how well an algorithm's performance scales with instance size. An alternative that is occasionally tried is to construct large artificial instances in such a way that the optimum solution value is known in advance. Unfortunately, restricting your tests solely to such instances is likely to yield unconvincing results because of the narrow and artificial nature of the instances. As to how you can evaluate instance quality for instances with unknown optima, we will discuss this in detail when we get to the Principle 6.

Returning to our general discussion about real-world test beds, note that even when such a test bed contains a reasonable range of instance sizes, papers that study only instances from the testbed can still have a hard task in drawing general conclusions about how algorithms operate and why they do so, as one can't design experiments that isolate instance properties and test how they affect performance. If the set of real-world instances is broad and varied enough, one may however at least be able to draw conclusions about the robustness of various performance measures across instance types.

Nevertheless, a hybrid approach is clearly to be recommended. The idea is to study one or more structured random instance classes, combined with results for real-world instances that enable the reader to evaluate the predictive quality of the random results. Especially useful are parameterized random instance generators such as the `washington` network generator of R. Anderson (available from DIMACS at `ftp://dimacs.rutgers.edu/pub/netflow/generators/`), which allow you to investigate the effect of structure as well as size on performance.

Principle 4. Use Efficient and Effective Experimental Designs

I earlier warned about the danger of spending too much computation time on the wrong questions, but even when you have identified the right questions, or at least the ones you want to answer, there are still computational savings to be had. In particular, there exist important techniques that will not only help you reduce the amount of computation you need to perform but also allow you to get broader and more accurate answers. Let me mention just a few specific ones.

SUGGESTION 3. *Use variance reduction techniques.*

When one is dealing with randomized algorithms (or randomly generated instances), the variability between runs of the algorithms (and between instances) may obscure your results and make it hard to draw conclusions. Thus one may have to perform large numbers of runs or test large numbers of instances to obtain reliable estimates of average performance. Variance reduction techniques help reduce these numbers. For an introduction to and broad-ranging survey of such techniques, see [McG92]. Here I will mention one that I have found useful for reducing the effects of instance variability.

Suppose you are attempting to compare average performance for several approximation algorithms over a class of randomly generated instances. Rather than independently estimate the average performance of each algorithm over the instance class and then comparing the estimates, use the *same* set of randomly generated instances for all your tests. Since each algorithm sees the same instances, the variability between instances will be factored out of the comparison. Thus, even though one may not have tight enough confidence intervals on the average solution qualities for the two algorithms to distinguish them statistically, the estimate of the average *difference* between solution values may be statistically significant and allow one to claim that one is better than the other.

I have used a variant of this technique in studying algorithms for the TSP, where I evaluate solution quality (tour length) in terms of its excess over the Held-Karp bound for the given instance. For the popular model in which cities correspond to points uniformly distributed in the unit square and distances are Euclidean, one can obtain very good estimates of what the expected value of that bound is for any number of cities N [JMR96], and so could compare algorithmic results to these estimates. However, one gets a more precise performance metric (with fewer samples) if one focuses on the difference between the algorithm's tour length and the Held-Karp bound for each instance.

SUGGESTION 4. *Use bootstrapping to evaluate multiple-run heuristics.*

For randomized approximation algorithms, one in practice may want to exploit the variability of the results by performing multiple runs and taking the best solution found. The naive way to estimate the expected behavior of the algorithm "Perform k runs of algorithm A and take the best" would be to run that algorithm itself some number m times, and compute the average of the m "best of k"s. This requires mk total runs of A, but those runs can be better spent using the technique known as "bootstrapping" First sort the mk results in order of solution value. It is now a relatively simply matter to compute the expected best value of a random sample of k of these results, chosen independently (with replacement), and this estimate makes better use of all the data than simply taking the average of the m bests. Moreover, using the same data set, one can also estimate the expected "best of k'" results for values of k' other than k.

Note that if we are going to take this approach, special attention must be paid to obtaining correct estimations of running time, especially for those "best of k" algorithms that only need to read the instance and perform any preprocessing steps once, thus amortizing these operations over all k runs. Since we may have performed these steps just once for all our km runs, we must be sure to measure the time for them separately in order to account for it properly in our final running time figures. This however is only a minor addition to the total work involved. It also leads to my final suggestion on experimental design.

SUGGESTION 5. *Use self-documenting programs.*

This suggestion is designed to reduce the experimenter's own time, rather than that needed for computation. When you generate large quantities of data, you must deal with the problem of organizing and accessing that data. And note that the accesses may take place years in the future, for example in response to a referee's report or your own desire to do a follow-up study. Thus it is dangerous to rely simply on personal memory. Saving data in files and directories with descriptive names can help, as can constructing `README` files providing directory road maps and other information. However, data that cannot be accurately interpreted after the fact is useless data. Thus it is desirable to have output files that contain (or are linked to) all the information one would want to know about the experiment that generated them. This includes not only performance metrics such as running time and solution quality, but also the name (and version) of the algorithm used, the machine on which it was run (and date, to help in case that machine gets upgraded), the name of the instance to which it was applied, the settings of all adjustable parameters, and any subsidiary measurements (operation times, counts, intermediate solution values) that you think may be of interest later. Moreover, to minimize the possibility of later misinterpretation, one should not rely on the syntax of the output file to tell you which data item is which, but should include the name of the item ("algorithm version," etc.) along with each item. For more detailed discussions of these issues, see [Ben91, McG01].

Principle 5. Use Reasonably Efficient Implementations

This is surprisingly a somewhat controversial principle. Although it would at first glance seem obvious that we should want to use efficient implementations (especially in a field like algorithm design where efficiency is one of our main goals), efficiency does come at a cost in programming effort and there are several situations in which researchers have argued that they should be allowed to settle for less.

For example, in some problem domains (the Traveling Salesman Problem is again an example) the best algorithms gain much of their speed from sophisticated data structures and speed-up tricks. This places a fairly high barrier to entry for researchers with new algorithmic ideas they want to promote: in order to get competitive codes that incorporate the new ideas, one may have to implement all those complicated speed-up mechanisms as well. Thus, many researchers have attempted to argue that it's permissible to forgo the speed-ups and still get meaningful results.

PET PEEVE 5. *Claiming inadequate programming time/ability as an excuse.*

In some cases, researchers simply assert that their implementations would likely be competitive with those of previous algorithms had they only had the time or skill to use the same speed-up mechanisms. A slightly more sophisticated variant on this involves comparing the results for a speed-up-free implementation of your algorithm to an implementation of its competitor with the speed-up mechanisms disabled. Here the implicit claim is that running time comparisons made for the speed-up-free case would translate directly to the situation in which both codes included the speed-up mechanisms.

This is a highly suspect argument. First of all, it is hard to quantify how much of a speed-up one would have obtained by implementing the additional mechanisms unless one actually does implement and test them. Second, there is no guarantee that a given speed-up mechanism will be equally effective for different algorithmic approaches. And third, with the slower implementations one cannot perform as many tests or handle as large instances, and so crucial comparative data will be missed.

A second reason why one might argue for saving programming time at the expense of additional computation time, and one that I myself have been guilty of using, arises in the case where one is primarily studying some other metric, such as the quality of solutions produced by an approximation algorithm, and the speed-up mechanisms would affect only the running time, not the solutions. If your machine is sufficiently fast to enable your inefficient implementation to generate all the data you need, this would seem to be acceptable (at least until the referees start asking for more data on larger instances).

In general, however, the advantages of efficiency are many:

1. Efficient implementations make it easier for you to support claims of practicality and competitiveness.

2. Results for implementations significantly slower than those one would want to use in practice may yield distorted pictures.

3. Faster implementations allow one to perform experiments on more and/or larger instances, or to finish the study more quickly.

Note that this principle does not imply you need go overboard on efficiency issues and fine tune your code mercilessly to save the last 10% in running time. Nor does it mean you must implement all the theoretical refinements that yield only small improvements in worst-case running time, especially if there is no indication that such tweaking will have a major practical impact. This is why the current principle refers only to *reasonable* efficiency. Unless your specific experimental goal is to evaluate the impact on running time of such theoretical refinements and speed-up tricks, you are advised to avoid the following pitfall.

PITFALL 5. *Too much code tuning.*

You should certainly make efficient use of data structures when this is called for and when real-world implementers would be likely to do so, e.g., $\Theta(N^2)$ sorting algorithms should typically be avoided. However, this is another situation where advance thinking and experimental exploration can save you effort. By understanding where the computational bottlenecks in the algorithm lie, and by profiling early implementations, you can determine where code optimizations and more sophisticated data structures are likely to have a major effect, and thus concentrate your programming effort where it will do the most good. Ideally the efficiency of the code you test should be (at least to within a constant factor in practice) as good as that of a code you would expect people to use in the real world.

Principle 6. Ensure Reproducibility

As with all scientific studies, a key part of an experimental paper is the reproducibility of the results. But what does "reproducibility" mean in the context of the experimental analysis of algorithms? In the strictest sense it means that if you ran the same code on the same instances on the same machine/compiler/operating system/system load combination you would get the same running time, operation counts, solution quality (or the same averages, in the case of a randomized algorithm).

This, however, would not be very informative, nor does it correspond to what is meant by "reproducibility" in classical scientific studies. In reproducing such a study, a scientist must use the same basic methods, but will typically use different apparatus, similar but distinct materials, and possibly different measuring techniques. He will be said to have reproduced the original results if the data he obtains is consistent with that of the original experiments and supports the same conclusions. In this way he helps confirm that the results of the original experiment (and the conclusions drawn from them) are independent of the precise details of the experiment itself.

It is this broader notion of reproducibility that I consider to be the most important. It has implications both for how you do your experiments and how you report them. Your experiments need to be extensive enough to give you confidence that your conclusions are true and not artifacts of your experimental setup (the machines, compilers, and random number generators you use, the particular instances you test, etc.) In reporting your results you must describe the algorithms, test instances, computing environment, results, etc. in enough detail so that a reader could at least in principle perform similar experiments that would lead to the same basic conclusions.

It is not typically necessary, however, to provide the precise instances that were tested, except perhaps those with unique properties that would be difficult to reproduce. If the results of the original experiment are reproducible in the broader sense, then we should be able to obtain similar results for similar but distinct instances. By the same token, if the original conclusions were claimed to hold for an algorithm rather than a particular implementation of one, then the broader notion of reproducibility would require that we test an independent implementation of the algorithm described in the paper, rather than the original author's code, even if the latter is available. Not only does this enhance our confidence in the conclusions, but it avoids a significant danger.

PET PEEVE 6. *Supplied code that doesn't match a paper's description of it.*

All too often I have found that the code an author supplies does not precisely implement the algorithm described in the corresponding paper. Differences can range from incorrect input specifications to more serious issues such as missing or added steps. Typically this is an honest mistake, due to poor record keeping on the author's part, but it is no less frustrating.

Of course, if the conclusions of the original paper refer to the performance of a specific code rather than the algorithm it implements, then reproducibility does require that the author provide access to his code. This might happen for instance in the evaluation of software packages, where papers often do not provide complete description of the details of the algorithms involved. Similarly, if we are dealing with a "horse race" paper where the main conclusion is that algorithm/implementation A beats algorithm/implementation B on a given testbed of instances, reproducibility requires that the author provide access to the testbed used. Both these cases, however, are more like product testing than science and are not our main focus here.

Complicating the whole issue of reproducibility is the fact that one key algorithmic measurement, the one we are often most interested in, is inherently irreproducible in the broad sense I have been advocating. This is running time. Broadly reproducible results should not be machine-dependent, but even if one fixes an implementation and an instance, running times are dependent on machine, compiler, operating system, system load, and many other factors. The best one can hope for is to conclude that the running times on the different combinations of machines, operating system, etc. are in some sense consistent with other measurements of the relative speeds of the combinations. We will have more to say about the inexact science of making such comparison in our discussion of the next Principle (Ensure Comparability).

The Principle of Reproducibility is perhaps the most violated one in the literature today. Besides simple failures to report key information and when appropriate provide access to code and/or instances, there are less obvious infractions:

PET PEEVE 7. *Irreproducible standards of comparison.*

Suppose you are experimentally evaluating approximation algorithms and have followed my advice and not restricted attention to instances for which the optimal solution value is known. You are then confronted with the question of how to measure the relative goodness of a solution. Here is a list of the answers to that question that I have encountered, several of which have associated reproducibility drawbacks, some of them fatal (see also [McG96].)

- Simply report the solution value.

  This certainly is reproducible in the narrow sense, but not in the broader sense that one can perform experiments on similar instances and determine whether one is getting similar results. It also provides no direct insight into the quality of the algorithm.

- Report the percentage excess over best solution currently known.

  This is narrowly reproducible if you explicitly state the current best and also make the instance available to other researchers. Unfortunately, many authors omit at least one of these two actions. Moreover, even if you provide both instance and value, this approach does not yield reproducible results in the broader sense. Furthermore, "current bests" represent a moving target and one usually can't tell whether all the "current bests" for the overall testbed are equally good. Thus we again may be left without a clear picture of the algorithm's true quality.

- For randomly generated instances, report the percentage excess over an estimate of the expected optimal.

This approach leads to reproducible results only if one explicitly states the estimate or completely specifies the method for computing it. It leads to *meaningful* results only if (i) the estimate is in fact consistently close to the expected optimal, and (ii) the values of the true optima under the distribution have relatively low variance. In most applications of this approach that I have seen, at least one of these two requirements was not satisfied.

- Report the percentage excess over a well-defined lower bound.

  I have already given an example of this approach in the discussion of Principle 4, when I described the use of the Held-Karp TSP lower bound [HK70, HK71, JMR96] for variance reduction. Assuming the lower bound is usually close to the optimal solution, as is the case for the Held-Karp bound, this can be a useful approach. It is only reproducible, however, if the lower bound can be feasibly computed or reliably approximated. When I first started using the Held-Karp lower bound as a standard of comparison for the TSP [Joh90], it probably did not completely meet these criteria, as we could only compute estimates whose quality we could not guarantee. Indeed, subsequent validation experiments showed that our estimates were only reliably close to the true value when we were dealing with randomly generated instances. Now however, the bound can be calculated exactly for instances as large as 1,000,000 cities using the publicly available `Concorde` code of [ABCC98]. Unfortunately, not all problems have such good and computable bounds.

- Report the percentage excess/improvement over some other heuristic.

  This appears to be the ultimate fallback option. It is reproducible so long as the "other heuristic" is completely specified. Unfortunately, it can be difficult to specify that other heuristic if it is at all complicated, and too often authors have settled for simply naming the heuristic, simply saying "2-Opt," "Lin-Kernighan," or (my favorite example of gross underspecification) "simulated annealing." For all these algorithms, there are variants having a wide range of behaviors, and the name itself does not provide enough detail to distinguish between them (nor do most references). If one is to apply this approach successfully, it is best to use a very simple algorithm as the standard, one that *can* be specified precisely in a few words, and preferably one that is deterministic. If a more complicated benchmark algorithm is desired, then the only viable option is to provide the source code for an implementation, or use one whose source code is already available on the Web.

PET PEEVE 8. *Using running time as a stopping criterion.*

Many types of approximation algorithm, for example multiple-start local search and truncated branch-and-bound, can provide better and better solutions the longer one lets them run. In a practical situation where limited computation time is available, one may well want to simply "run the code for an hour" and take the best solution found so far. Note, however, that "run the code for an hour" is not a well-defined algorithm and will not yield reproducible results. Should one use a machine with a different processor or operating system, or just a more/less efficient implementation of the algorithm on the same machine, one might get solutions having a distinctly different level of quality. Thus defining an algorithm in this way is not acceptable for a scientific paper.

The temptation to use this stopping criterion is especially strong when one is comparing a variety of different algorithms. Many authors, in hopes of providing a "fair" comparison, have performed experiments in which each algorithm was allowed the same amount of running time, say 60 minutes. However, these comparisons are not reproducible. If one reruns the same test on a machine 10 times faster, the algorithms will presumably all do better but their relative rankings may change substantially.

So what is one to do if one wants to make reproducible "fair" comparisons? One solution is to design your codes so that some readily measurable combinatorial count (number of neighborhoods searched, number of branching steps, etc.) is used as the stopping criterion. You then have a well-defined algorithm, for which running time and quality of solution can be measured as a function of the combinatorial count, and the latter is (or should be) reproducible. Given data on the average values of these two metrics as a function of count, one can then *derive* a table that reports results for a specific running time, but also includes the combinatorial counts associated with the running time for each algorithm. Subsequent researchers can then in principle reproduce the table for the given counts, and observe how the differences caused by their machines and implementations affected the various running times (which may or may not remain roughly equal).

PET PEEVE 9. *Using the optimal solution value as a stopping criterion.*

My complaint here is not about optimization algorithms, which stop when they have found an optimal solution and *proved* that it is optimal. Rather, it is about a practice that I have encountered in more than a few papers in the metaheuristic literature, where the algorithms concerned have no mechanism for verifying optimality but simply look for a very good solution until some stopping criterion is reached. In the papers in question, the algorithms are given the instance and the optimal solution value (if known), and are halted early should a solution with the optimal value be found. (The algorithms typically have a back-up stopping criterion for instances with unknown optima and for runs that fail to find a known optima.) The temptation to truncate searches in this way is I guess natural: if you already know what the optimal solution value is, why let your heuristic waste time after it has reached that goal?

The drawback to this approach is that in practice one would not typically run an approximation algorithm on instances for which an optimal solution is already known. Hence in practice one would not know when the optimal solution value had been reached, and so this stopping criteria cannot be used. Thus the tests for instances with known optima need not reflect performance in practice (where the benefit of early stopping will never be felt), and would not be "reproducible" in the sense that dramatically different running times might be reported for similar (or even the same) instances, depending on whether one knew the optimal value or not.

PET PEEVE 10. *Hand-tuned algorithm parameters.*

Many heuristics have parameters that need to be set before the algorithm can be run. For instance, in a multiple-start local optimization algorithm, one must specify the number of starts. More elaborate metaheuristics such as simulated annealing, tabu search, genetic algorithms, etc. can come with whole ensembles of adjustable parameters. For such an algorithm to be well-defined, one must either fix those parameters or define them in a way that depends on measurable aspects of the instance being handled. In the former case (fixed parameters) there is no problem of reproducibility as long as the paper reports the settings chosen (although the paper might be more informative and useful if some insight is provided on how performance would be affected by changes in the settings). In the latter case, however, many papers use different settings for different instances without explaining how those settings were derived beyond perhaps the words "after experimentation, we found the following settings to work well for this instance/class of instances."

There are at least two drawbacks to this approach. First, it means that the algorithm is ill-specified, since the parameter-setting process is not explained in detail. (Indeed, since it involved human experimentation and judgment, it may not have involved any sort of codified procedure.) Second, since the reported running times typically do not include the time spent in determining the parameters, it leads to a serious underestimate of the computing time needed to apply the algorithm to a new instance/class of instances.

The rule I would apply is the following: If different parameter settings are to be used for different instances, the adjustment process must be well-defined and algorithmic, the adjustment algorithm must be described in the paper, and the time for the adjustment must be included in all reported running times.

PET PEEVE 11. *The one-run study.*

Unless a study covers a wide enough range of instances (and a large enough number of runs in the case of randomized algorithms), the conclusions drawn may well be *wrong*, and hence irreproducible for that reason. The amount of data needed will of course vary with the detail of the conclusions. A first paper on a given algorithm may well settle for trying to indicate general trends in its performance without quantifying them, and this may require fewer test instances and/or fewer runs per test instance than a subsequent paper that wishes to characterize specific aspects of performance more accurately. However, it should be remembered that randomly-generated instances from the same distribution can have widely varying properties, especially when the instances are of small size, and that many randomized algorithms can also yield widely-varying results for the same instance. Thus it can be dangerous to infer too much from a single run on a single instance.

PET PEEVE 12. *Using the best result found as an evaluation criterion.*

With randomized algorithms, one can avoid the dangers of the one-run study by performing multiple runs on each instance. These, however, can lead to other temptations. In providing tables of results of randomized approximation algorithms, many papers include a column for the best result found as well as (or instead of) one for the average result found. The objections to this are twofold. First, the best solution found is a sample from the tail of a distribution, and hence less likely to be reproducible than the average. Second, in such tables, the running times reported are usually for a single run of the algorithm, rather than for the entire ensemble of runs that yielded the reported "best." Thus the time for obtaining this answer is obscured. Indeed if, as is often the case, the number of runs actually performed is not clearly stated, there is no way to determine the running time. And even if the number of runs is stated, note that the simple process of multiplying the running time by the number of runs may *overestimate* the time needed, given that certain actions, such as reading the instance and setting up data structures, may need to be done only once when multiple runs are performed.

If you think that in practice users may want to perform many runs and take the best, a more appropriate experimental approach would be to evaluate the algorithm "Perform k runs and take the best" directly, reporting both running time and solutions for this iterative approach. As we already saw in the discussion of Principle 4, there are efficient techniques for performing such evaluations using bootstrapping.

Principle 7. Ensure Comparability

This Principle is essentially the reverse side of the Principle about tying your paper to the literature. That earlier principle referred to the past literature. This one refers to the future. You should write your papers (and to the extent possible make relevant data, code, and instances publicly available in a long-term manner) so that future researchers (yourself included) can accurately compare their results for new algorithms/instances to your results. Part of this is simply following the above recommendations for ensuring reproducibility, but there are additional steps necessary. In trying to compare my own results to the literature, I have encountered several difficulties due to omissions of various sorts in past papers.

PET PEEVE 13. *The uncalibrated machine.*

Although most papers at least mention the machine on which their experiments were run, they often omit other key factors such as processor speed (not always evident from the machine model name), operating system and language/compiler. And even when they present all this information, it can still be difficult to estimate the relative difference in speeds between the earlier system and your own, a difficulty that only gets worse as time passes and machines get outdated and forgotten. When talking about machines several generations apart, even if one knows the quoted processor speeds for the two machines, e.g. 20 Mhz versus 500 Mhz, the raw speed of the processor in Megahertz is a notoriously unreliable predictor of performance. (Even for processors currently on the market, differences in machine architecture render comparisons based solely on quoted processor speed highly suspect.)

SUGGESTION 6. *Use benchmark codes to calibrate machine speeds.*

The benchmark is distributed as portable source code that preferably involves data structures and algorithmic operations somewhat similar to those used in the algorithms studied. You compile and run the benchmark code on the same machine and with the same compiler that you use for the implementations you are studying reporting its running times for a specified set of publicly available test instances of varying size. Future researchers can then calibrate their own machines in the same way and, based on the benchmark data reported in the original paper, attempt to normalize the old results to their current machines.

In the world of scientific computing, where floating point computation and `Fortran` are the dominant computational paradigms, this approach is widely practiced. The benchmark codes are from the `LINPACK` suite described in [Don00], a report that lists, for various combinations of machine, operating system, and compiler, the number of megaflops (millions of floating point operations per second) performed on each of three benchmark computations. These benchmark codes have certain drawbacks for our purposes, however. Being written in `FORTRAN` and dominated by highly structured loops of floating point operations, they may not adequately reflect the codes that many of us study, which are combinatorial in nature and written in `C` or `C++`. Furthermore, because the `LINPACK` benchmarks are limited to just three computations, the derived numbers may not adequately quantify memory hierarchy effects.

Thus in the world of combinatorial algorithms it may be worthwhile to promulgate new and more relevant benchmark codes. This is the approach taken in several DIMACS Implementation Challenges. For example, see the Web site for the DIAMCS TSP Challenge at `http://www.research.att.com/~dsj/chtsp/`). Here the benchmark algorithm is itself a nontrivial TSP heuristic, and the instances cover a large range of sizes, from 1,000 to 10,000,000 cities. Given the benchmark results for two machines, one can plot the relative speeds of the two machines as a function of instance size, and use interpolation to normalize the time of an algorithm for a specific instance on one machine to what it would likely be on the other. There are limits on the accuracy of this approach, of course, and preliminary data from the DIMACS TSP Challenge suggests that one cannot hope that the normalized results will be closer than a factor of two to what one would have gotten had the same system been used for both studies. This, however, is much better than nothing, since many other factors can introduce discrepancies this large.

PET PEEVE 14. *The lost testbed.*

As I mentioned earlier when talking about variance reduction, there is much to be gained when comparing algorithms if one can run them on the same instances. (This is less important for reproducibility, where what really wants to verify are the conclusions of the study, which typically are independent of the precise answers given for various instances.) Thus for the sake of future researchers, you should do your best to perform many of your runs on instances to which subsequent researchers will have access. These days the standard solution is to make sure that the instances (or their generators) are available on the Web. This is of course less crucial if your testbed includes many samples from each distribution studied, as this enables you to average out much of the variability encountered in individual instances. However for individual instances, the future researcher may have no insight into whether that instance was particularly easy or hard or atypical in other ways without having access to the instance itself.

Admittedly, this requirement does open up possibilities for abuse. Future researchers interested in winning horse races might be tempted to optimize their codes for precisely those instances you studied, and thus beat you based on instance-specific code-tuning alone. This, however, is less likely if your testbed is sufficiently large and varied.

Another alternative to resolving the above two pet peeves is simply to make the source code for your algorithm available. What better way of ensuring comparability? (It would however still be worthwhile to also provide machine calibrations and your testbed instances, if only to enable future researchers to verify that the code you provide does indeed correspond to the one whose results you reported.) There can of course be obstacles to making the code available. Depending on the policies of your employer, it may or may not be possible to release your code to the public. Even if it is allowed, you may not want to release it until you are finished performing your own experiments with it (or have corrected all the spelling errors and put in all those comments that good programming style recommends). However, even if you are willing to make the source code available, there is still one more obstacle that may be encountered.

PITFALL 6. *Lost code/data.*

If you lose your code, you lose your best approach to obtaining comparability between it and future algorithms designed by yourself and others. If you lose the data on which the summaries reported in your papers are based, you lose the chance of going back and answering more detailed questions raised by readers or future papers. So why would one allow this to happen? I can mention several ways from bitter personal experience.

- Modifying the code without saving a copy of the original version.
- Unthinking responses to pleas of systems administrators to "clean up."
- Failure to keep back-up copies.
- Poorly organized directories containing multiple variants on code/data without clearly identifying the crucial ones.

Fortunately, for me the last is the most common, and with effort one can recover from it. One can also recover from completely lost data by re-running the experiments, and so one might be tempted to discard the original data after the paper is complete, especially if it takes up large quantities of disk space. However, re-running the experiments is unlikely to yield exactly the same running times (especially if you are no longer using the same machine), so you may not be able to justify the precise figures reported in your paper. For randomized approximation algorithms, you also may not be able to exactly reproduce solution quality results (unless you saved the seeds and use the same random number generator). Of course, if your results are truly "reproducible," your regenerated results will most likely support the same conclusions as your original data did. However, just as scientists are required to preserve their lab notebooks, it is still good policy to keep the original data if you are going to publish results and conclusions based on it.

Principle 8. Report the Full Story

Although it is typically overkill to present all your raw data, if you are overly selective you may fail to reach accurate conclusions or adequately support them. You may also impair your paper's reproducibility and comparability. If you report only averages, you should say how many instances (and how many runs in the case of randomized algorithms) you based the averages on. If the precise values of these averages are important for your conclusions, you should also provide information about the distribution of the results. The latter might simply be standard deviations if the data appears roughly normally distributed, but might need to be more pictorial (histograms, boxplots) if the distribution is more idiosyncratic. And any scaling or normalizing of measurements should be carefully explained so that the raw averages can be regenerated if desired. In particular, although pictorial summaries of data can often be the best way to support your overall conclusions, this should not typically be the only presentation of the data since pictures typically lack precision. Tables of data values/averages may be relegated to an appendix, but should be included.

Of course, one can go overboard.

PET PEEVE 15. *False precision.*

One of the most common statistical abuses is the presentation of averages to far more digits of accuracy than is justified by the data, with the subsequent temptation to draw conclusions about differences that are basically just in the noise. In the case of running times, even reporting individual measurements can be misleading because of the imprecision of the timing mechanisms in most operating systems and the unpredictable effect that other users and system effects may have on reported running times.

Admittedly, one can be lead to including all that extra precision simply for data layout reasons. For example, consider a table that includes average running times for a large range of instance sizes. In this case, one may need to go down to units of 10^{-2} to even get two digits of accuracy on the smallest instances, and once the column has some entries with two digits to the right of the decimal point, the temptation is to keep them even when the entries for large instances may be bigger than 10^6. My own solution (when I'm not violating this Pet Peeve myself) is simply to replace the insignificant digits with 0's and add a comment in the text or caption to explain what I'm doing.

A second major part of telling the "full story" concerns anomalous results. You should not hide them, either by omitting them or by failing to point them out when they are present. For reasons of intellectual honesty one should of course include results that are only "anomalous" in that they are inconsistent with the conclusions you wish to draw (e.g, results for instances where the code that you wish to champion underperforms). However, one should also report any strange (but reproducible) phenomena one encounters in one's experiments. Ideally, one should also provide explanations for such anomalies, but even if you have none, the reader should know of their existence. Currently unexplainable anomalies may eventually prove to be the key to major insights into the algorithms, implementations, or test data.

PET PEEVE 16. *Unremarked anomalies.*

The worst anomaly is the one you don't notice. Unfortunately, many papers contain them, leaving the reader to wonder whether it is a true anomaly or simply a typographical error (or evidence of an implementation bug). Anomalies are important. Look for them.

PET PEEVE 17. *The* ex post facto *stopping criterion.*

This is similar to my earlier pet peeve about using the optimal solution value as a stopping criterion, but concerns telling the full story rather than reproducibility. Many search heuristics do not stop when they find a local optimum, but continue looking for good solutions even if this means examining solutions that are worse than ones you have already seen. Typically one runs such a heuristic for some fixed number of steps and then returns the best solution encountered. Similarly, when performing multiple-runs of randomized algorithms, one typically fixes the number of runs in advance and on completion takes the best result found.

Unfortunately, when studying such algorithms, some authors don't report the total running time, but merely the time until that best solution was first encountered. In effect they are investigating a clairvoyant algorithm that knows in advance the best solution value it will ever see and so stops as soon as a solution with that value is encountered. This is not something we can do in the real world. Thus, although the statistic reported is presumably reproducible, it obscures the full story by underestimating the total time that the algorithm would take.

A better approach taken by many authors is to report, in addition to full running times, the number of steps/iterations taken before the best solution was found. Such information may help in determining general rules for how to set the parameter governing the total number of steps/iterations to be performed.

The above is a special case of the following more general (and perhaps more controversial) complaint:

PET PEEVE 18. *Failure to report overall running times.*

Even if your main subject is not running times, you should report yours. As noted above, running time is an important component of newsworthiness, and so a reader may legitimately want to know this information before committing time to a detailed study of your results. Here are some of the common reasons stated for *not* reporting overall running times. For each stated reason, I give a counter-argument as to why overall running time is still of interest.

1. The main subject of your study is only one component of the running time (say the local optimization phase). However: readers may legitimately want to know how important the contribution of the component being studied is, and for this they need information on overall running time.

2. The main subject of your study is a combinatorial count related to the algorithm's operation. However: readers may legitimately expect you to establish the meaningfulness

of the count in question, which presumably means studying its correlations with running time. For example, I have seen studies of different pruning strategies for an exhaustive search algorithm that measure only the number of subproblems explored under each. This can be misleading if in fact the "better" pruning scheme uses far more time in processing each node and hence leads to a larger overall running time.

3. Your paper concerns approximation algorithms and your main concern is simply in measuring the quality of the solutions they produce. However: a key reason for using approximation algorithms is to trade quality of solution for reduced running time, and readers may legitimately want to know how that tradeoff works out for the algorithms in question.

4. Your paper's goal is to estimate average-case solution quality for a well-studied approximation algorithm whose overall running time has previously been studied. This is actually a pretty good excuse (I've used it myself). However (and this argument applies to all the cases above as well): a reader interested in reproducing your results may well want to know how much computation will be involved in doing so. The same goes if what you are really doing is simply using algorithms as as tool in experimental mathematics (estimating asymptotic constants, determining properties of distributions, etc.).

Note that although the above may be the stated reasons for not reporting overall running time, the real reason is typically either that it takes a bit more effort to record running times or that the current code is sufficiently slow that reporting its running times may cast doubts on the significance of the results being reported. However: the need for a minor bit of extra work is not an excuse, nor, according to Principle 5, is failure to use efficient implementations.

Principle 9. Draw Well-Justified Conclusions and Look for Explanations

The purpose of doing experiments with algorithms is presumably to learn something about their performance, and so a good experimental paper will have conclusions to state and support. This is worth emphasizing, since, based on conference submissions I have seen, many beginners apparently do not realize it.

PET PEEVE 19. *Data without interpretation.*

It is not enough simply to perform tests, tabulate your results, and leave the reader the job of finding the implications of the data. (If there aren't any implications that you can find or express, then you've done the wrong experiments and shouldn't be writing a paper yet, as you've failed the "newsworthiness" test.) At a minimum, one should be able to summarize patterns in the data. Ideally, one might also be able to pose more general conjectures that the data supports and which subsequent research (experimental or theoretical) may try to confirm or deny.

Note that the conclusion might well have to be "the data is inconclusive," but if you've set up your study in order to answer interesting questions (as I've recommended), you should be able to derive an answer of some sort. You also need to make a convincing case that your data does indeed support the answers you state.

PET PEEVE 20. *Conclusions without support.*

It is surprising how often papers state conclusions that are not well-supported by the data they present, either because of glaring exceptions they don't comment on, trends they don't notice, or some other failure. As an example of the second failing, consider a paper I recently read that claimed two algorithms to be equally good because each was better half the time, but failed to notice that one was better only on the smaller instances in the testbed, and the other was better (by increasing margins) as instance size increased. A particularly common example of poorly-supported conclusion is the following.

PET PEEVE 21. *Myopic approaches to asymptopia.*

Given that theorists are interested in asymptotics (asymptotic running times, asymptotic worst-case ratios, etc.) it is natural for experimentalists from that community to try to address such questions experimentally. However, if there is anything one learns in this field, it is that behavior on small (or even relatively large) instances does not always predict behavior for *very* large instances. Thus studies that try to extrapolate asymptotic running time by studying times for instances of size 1000 or less can often be led astray. My own early bin packing experiments in [Joh73], based on experiment with 200-item lists were also woefully misleading. I concluded that for item sizes uniformly distributed between 0 and 1, Best Fit was on average 5.6% worse than optimal. Subsequent experiments with lists of 100,000 items or more [BJLM83] implied that Best Fit was in fact asymptotically optimal, as has now been proved [Sho86]. Similarly, for many TSP heuristics, the relative quality of the tours found by two heuristics on 100-city instances can be far different from the relative quality found for 10,000-city instances (in either direction) [JBMR03]. If one is interested in asymptotics, one should study as large instances as possible. In addition, if one is interested in running time, it pays to study it in as much detail as possible.

SUGGESTION 7. *Use profiling to understand running times.*

An algorithm's running time is typically made up of many components, each with its own growth rate. A $\Theta(n)$ component with a high constant of proportionality may dominate a $\Theta(n^2)$ component when n is small, although the latter will eventually determine the algorithm's behavior. Profiling your code, which can provide you with both the number of calls to the various subroutines as well as the time spent in them, can help you spot those higher-exponent components of running time well before they make a significant contribution to the overall time.

Note that in helping us get a better estimate of the true asymptotic running time, profiling also provides an explanation of how that running time comes about. Such explanations are a key part of an experimental analysis paper. In the real-world, the key performance characteristics of an algorithm are it's resource requirements (running time, memory usage, etc.) and (with approximation algorithms) the quality of the solutions it generates. It is thus natural to concentrate on measurements of these quantities when performing a first experimental analysis. However, if our goal is understanding, we must soon turn to the question of explaining why the observed measurements occur. (Plausible explanations also add to the credibility of the experimental results themselves.)

Explanations are at their most believable if backed up by compelling evidence. When a worst-case $\Theta(N^2)$ algorithm appears to be taking $\Theta(N \log N)$ in practice, what part of the worst-case analysis proved to be overly pessimistic? Why does a seemingly more complicated algorithm take less time in practice than a supposedly simple and efficient one? Often these questions can be answered by profiling or by instrumenting the code to perform additional measurements. (Measurements of this sort have also on more than one occasion led to the "explanation" that an implementation was defective, something one would definitely like to know before publishing results about it.)

Principle 10. Present Your Data in Informative Ways

The best way to support your conclusions (and sometimes also the easiest way to derive them) is to display your data in such a way as to highlight the trends it exhibits, the distinctions it makes, and so forth. There are many good display techniques depending on the types of points one wants to make. Illustrating them all would unfortunately require a much longer paper than this. For now, readers are best advised to take a critical look at the examples of good experimental papers listed in this and other papers, and try to adapt those techniques they themselves find most informative. For additional ideas, one can also consult the general literature about data display (e.g., see [Cle85, Cle93, Tuf83]). My approach here, as with the other Principles, will be to organize the discussion around warnings about potential mistakes and suggestions about how to avoid them.

PET PEEVE 22. *Tables without pictures.*

Tables by themselves are usually a very inefficient way of telling your story. Most readers (and talk attendees) *hate* presentations where the punch line is a large, multi-columned table in fine print. If there is any graphical way to summarize the data and reveal its message, it is almost always to be preferred to a table alone.

PET PEEVE 23. *Pictures without tables.*

On the other hand, although pictures can often tell your story more quickly, they are usually a poor way of presenting the details of your results. I don't want to have to get out a ruler to estimate the value of the solution your algorithm found so that I can compare it with the results of my own algorithm. So a good paper should contain *both* pictures and tables, although feel free to stick the tables in an appendix so that readers who just want to see the big picture don't have to deal with them.

PET PEEVE 24. *Pictures yielding too little insight.*

Although pictures can add much to our understanding of the data, not all pictures are equally useful. Care must be taken in figuring what to display and how to display it, and without such care one can get figures that tell us very little. Suppose for example we wish to construct a meaningful display about running time growth. (I choose this example because so many papers do a poor job of it.) Suppose we have tested five algorithms on TSP instances ranging in size from 100 to 1,000,000 cities, going up by factors of roughly $\sqrt{10}$, and the table below gives the average running time data we obtained (no doubt stated with more precision than is justified for the larger instances and less than we would like for the smallest ones).

	100	316	1,000	3,162	10,000	31,623	100,000	316,227	1,000,000
Algorithm A	0.00	0.02	0.08	0.29	1.05	5.46	23.0	89.6	377
Algorithm B	0.00	0.03	0.11	0.35	1.38	6.50	30.6	173.3	669
Algorithm C	0.01	0.06	0.21	0.71	2.79	10.98	42.7	329.5	1253
Algorithm D	0.02	0.09	0.43	1.64	6.98	37.51	192.4	789.7	5465
Algorithm E	0.03	0.14	0.57	2.14	10.42	55.36	369.4	5775.0	33414

Simply staring at this table is not going to give us much insight other than the fact that the algorithms have been roughly ordered by speed, with the same ranking holding for each instance size. Can a picture tell us more? Figure 1 illustrates four different ways we might display the data, all of which I have seen in experimental papers, and only one of which (I would argue) gives much useful additional insight.

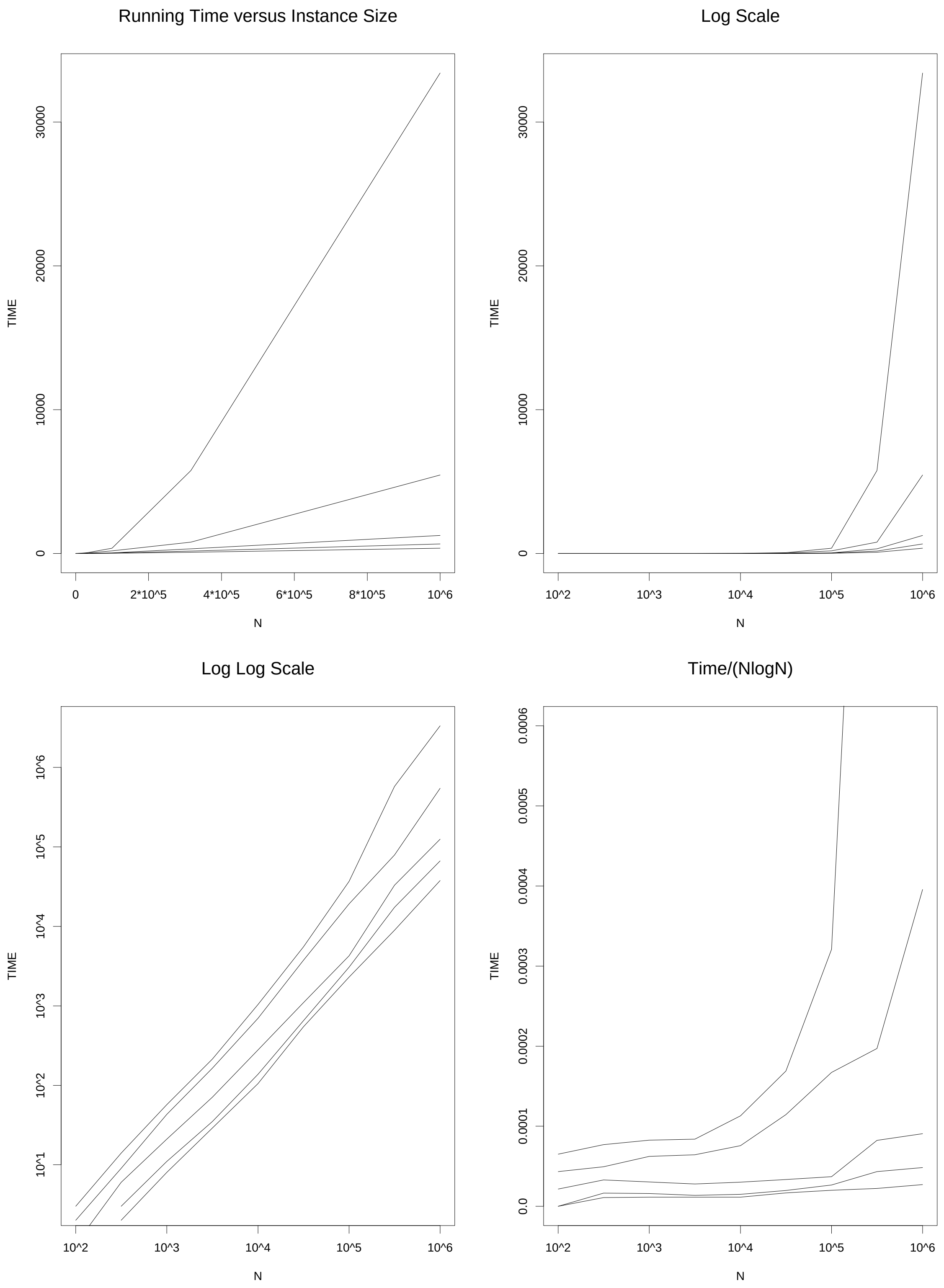

Figure 1: Displaying running time as a function of instance size.

The display in the upper left hand corner of Figure 1 simply charts running time versus number of cities. Note that a large majority of the data points are concentrated in the leftmost tenth of the figure and have y-coordinates that are hard to distinguish from 0. So from this picture all we can conclude (or would be able to conclude if I had labeled the curves) is that for large instances Algorithms D and E are a lot slower than the others, with the gap increasing. We have lost the insight that the relative rankings are consistent for all instance sizes.

The display in the upper right hand corner of Figure 1 uses a logarithmic scale for the x-coordinate, and so now the data points are evenly spread in the x direction, but because the running times for instances with 100,000 or fewer cities are still so small in comparison to those for larger instances, we get no more insight than we did from the previous display.

The display in the bottom left uses logarithmic scales for both axes, and now the consistent ranking of the algorithms for the different instance sizes is visible. Moreover, in theory one could determine something about the running time growth rates by measuring the slopes of the five curves. However, visually the slopes all look more or less the same and it is essentially impossible to estimate their values without resorting to a ruler and scratch paper.

The final display (bottom right hand corner) exploits the following suggestion.

SUGGESTION 8. *Display normalized running times.*

Having determined from our theoretical understanding of the algorithms and detailed analysis of the data that the fastest three algorithms have running times that grow as roughly $n \log n$, we divide all the running times by this quantity, and chart the resulting normalized times using a logarithmic x-axis. The resulting picture then reveals not only that the running time rankings are consistent and that algorithms D and E are much slower than the first three, but also that those first three have close to $\Theta(n \log n)$ running times, and that the latter are asymptotically much worse. We also get a clearer idea of the relative speeds of A, B, and C (or would if one added horizontal grid lines at appropriate intervals.) The one thing we lose is a clear picture of what the actual running times are, but that's what tables are for. We also get an illustration of another of my pet peeves.

PET PEEVE 25. *Inadequately or confusingly labeled pictures.*

Clearly if the data were real the curves in Figure 1 should have been labeled with the algorithms to which they correspond. A common practice is to use different symbols for the data points on the curve, but often the symbols are not sufficiently distinct to distinguish without a magnifying glass. Moreover, they can become almost totally indecipherable when data points are close together or coincide, as can happen in situations like the following.

PET PEEVE 26. *Pictures with too much information.*

Pictures do their job best when they are clear and uncluttered. If the bottom right display of the previous discussion had covered 20 algorithms instead of 5, it might well have lost its impact by making it difficult to distinguish (or even keep track of) the trend lines for the different algorithms. In general, the more algorithms you try to cover in a single figure, the harder it is to make your points clearly. In addition, to the extent that you include algorithms with widely different run times, you may be forced to compress data and obscure distinctions in order to fit everything in. A better solution might be to have multiple figures, each devoted to a subset of the algorithms in a given performance range. You can still provide an overall view that links all the results together if you use overlapping sets.

PET PEEVE 27. *Confusing pictorial metaphors.*

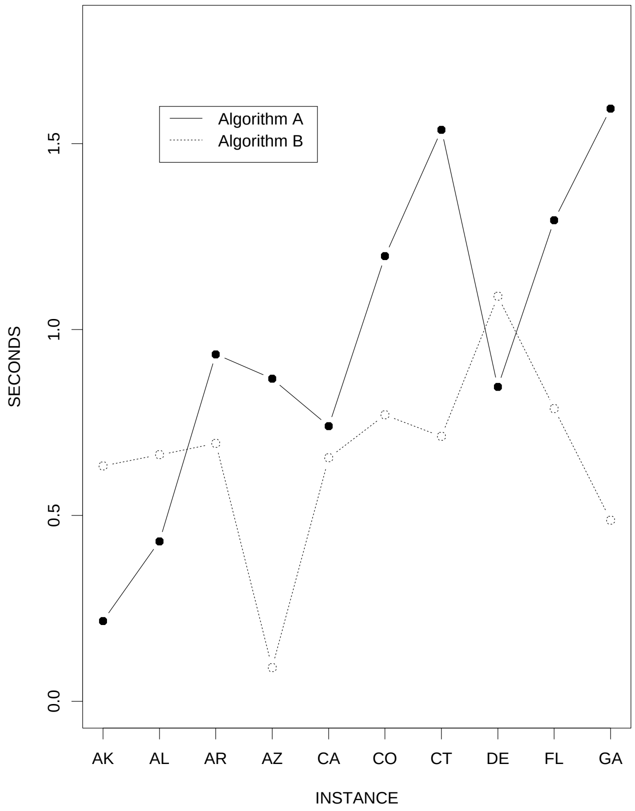

Figure 2: Spurious trend lines.

Too much creativity can be a dangerous thing. If you can't explain a figure in a short caption (or perhaps a short caption and an additional short paragraph in the text), then you may not be able to get your message across to your readers.

PET PEEVE 28. *The spurious trend line.*

This common but aggravating practice is illustrated in Figure 2. Here we have performed experiments on a collection of 10 instances. They might be 10 samples from the same instance distribution or simply 10 members of some standard testbed. Each instance has a name, and after sorting the names into alphabetical (or some other arbitrary) order, we plot our running times (or other metrics) for the 10 instances, and then for each algorithm *connect the data points by lines.*

The reasons this is done are understandable. Some plotting software packages do it automatically. Or perhaps one is concerned that the data points for the algorithms might be confused and so links them up to clarify which ones go with which algorithm. However, linking data points together in this way normally has certain implications. It suggests that there are underlying trends in behavior as you move from left to right. It also sometimes suggests the conjecture that one can interpolate between the results presented based on the curves shown. Neither of these inferences is typically warranted in this situation however, unless one seriously means to suggest that algorithmic performance is influenced by the alphabetic ranking of an instance's name.

Pet Peeve 29. *Poorly structured tables.*

Even if you are going to relegate your table to an appendix, but especially if you haven't figured out how to make your point graphically, it is important to order the rows and columns so as to highlight important information. For example, even though it is fairly common, it is usually not a good idea simply to sort your columns or rows by instance or algorithm name. The danger is not (as in the previous discussion) that you will suggest false trends, but rather that you will obscure true ones. Ordering instances by size makes a lot more sense than ordering them by name, since one can then more readily spot trends in time or solution quality that correlate with size. Ordering algorithms by solution quality or running time also makes these relationships more visible. Finally, if a single table reports values for several metrics, thought should be given to how best to order the columns/rows for the metrics.

Pet Peeve 30. *Making your readers do the arithmetic.*

This is another common problem with tables, and refers to what might be called the "missing column." As an example, suppose you are evaluating an approximation algorithm on instances for which good lower bounds (or the optimal solution values) are known. In this case, do not simply present a table of the solution values found by the algorithms, thus leaving readers with the task of performing the division that will tell them how close those solutions are to the corresponding lower bounds (which is the information a reader would typically want to know). This derived statistic should have been in the table to begin with.

Pet Peeve 31. *The undefined metric.*

This typically occurs in tables, although the same thing can show up on the axis labels for figures: A cryptic or ambiguous label is given to a column/axis and remains unexplained, either in caption or the text of the paper. The most common example is the running time column where it is not clear if the times reported include instance reading or preprocessing times (the latter of which can for some algorithms be substantial), or whether it refers to the time for a single run or the total time for all the runs performed. Another example is the column marked "number of iterations" when nowhere in the paper is it clearly stated what constitutes an iteration. Another is the column marked "time" with no indication of whether it is measured in seconds, microseconds, or what. It really isn't difficult to spell such things out clearly, and there is no excuse for not doing so.

Pet Peeve 32. *Comparing apples with oranges.*

My prime example of this problem is the table that presents running times for various algorithms on a given set of instances, but for each algorithm the entries have been taken from an earlier paper whose experiments were performed on a different (and often much slower) machine. Even when the caption or text does contain the caveat that different machines were involved, a reader skimming the paper is likely to be seriously misled about the relative running times of the algorithms. Readers who want a more accurate comparison are left with the task of normalizing the running times for themselves, making this another illustration of the problem of making the reader do the arithmetic.

Pet Peeve 33. *Detailed statistics on unimportant questions.*

If one is to get the full picture of an algorithm's performance, one must perform tests on many instances. The precise determination of results for one instance (or one random distribution of instances) may be less important than the overall picture. Doing a thousand runs to get narrow confidence limits on one such data point is usually a mis-allocation of resources, since it may prevent you from examining other data points.

Another example of misguided use of statistics is the use of detailed tests to determine with what confidence we can say that algorithm A is better than algorithm B on a particular instance or class of instances. If the class is narrow, the results are too narrow to be of interest (for example, applying this approach to a set of small instances and ignoring the fact that relative performances may change drastically as instance size increases). If the class is broad, such tests may obscure the much more interesting fact that the identity of the better algorithm may depend on the type of instance. Moreover, if one really needs sophisticated statistical tests to distinguish the performance of two algorithms, a more appropriate conclusion might be that for practical purposes their performance with respect to the given metric is "about the same" and hence the decision on which algorithm to use should be based on other grounds.

Statistics are useful, but should be used to support general conclusions (or particularly interesting specific ones). In particular, they do have a necessary role to play in experimental average-case papers, as for instance in [JMR96, PM96].

PET PEEVE 34. *Comparing approximation algorithms as to how often they find optima.*

One situation in which one *should* be interested in details of the distribution of results is the case of randomized approximation algorithms. Here, as has already been mentioned, one has the opportunity to run the algorithm multiple times and thus exploit the variability of the distribution of solutions.

One aspect of such distributions that is definitely of interest in certain situations is the probability that the algorithm will find an optimal solution for a given instance. For example, it is my observation that for most instances of the TSP with 50 cities or less, the Lin-Kernighan algorithm (randomized by its choice of starting solution) typically finds the optimal solution with high probability.

However, using the frequency with which optimal solutions are found as the primary metric in comparing algorithms has serious limitations. First, it restricts attention to test instances for which optimal solutions are known (see Pet Peeve 4). Second, it ignores the question of how near to optimal the algorithm gets when it does not find the optimum. And finally, it is a metric that will fail to make any distinctions at all between algorithms once instances get reasonably large and, as expected, the probability of finding an optimal solution goes to zero for all the competitors.

PET PEEVE 35. *Too much data.*

In doing an experimental study, getting additional data is almost always valuable. However, in writing up your studies once you have fully analyzed your data, presenting too much data can overwhelm your readers and obscure your main points. Raw data should be replaced by averages and other summary statistics when appropriate. If several instance classes yield basically the same results, choose a representative one and present the data for that one only, while pointing out in the text which other classes behave similarly. Algorithms that are dominated and not interesting for other reasons should be dropped from tables and charts. Data thus omitted may be included in appendices, or perhaps better yet be relegated to a Web archive where interested readers can access it. (With the growing popularity of online journals, the distinction between appendix and Web archive of course is beginning to disappear).

Section 2. Extensions and Other Points to Ponder

This paper has attempted to give advice about how best to analyze algorithms experimentally and report your results. Much of the discussion has been about what *not* to do, but I hope that the reader has come away with ideas about good practices as well, and that those who do not agree with all the points I have made have at least been stimulated to think more deeply about the issues involved.

The discussions of the preceding sections were primarily addressed to the experimentalist interested in studying sequential algorithms for the "classic problem" in which one is given a single input and asked to produce a single output (whose quality can be measured by a single-valued objective function). Not all computational tasks fit this paradigm, but many of the issues raised and suggestions made apply more broadly. I leave it as an exercise to the reader to determine which ones apply and which need to be modified in other situations, such as the following:

- Parallel/distributed algorithms for classic problems.
- Algorithms for problems with multi-valued or imprecise objective functions.
- Algorithms for program testing/checking/correcting.
- Dynamic/reactive algorithms that are required to respond to a sequence of requests with a conceptually infinite time horizon.

Given that the operating principle of this paper has been basically to present lists of aphorisms, let me conclude with one additional list, this one addressing the extent to which we should rely on various common resources without constant vigilance.

Whom can you trust?

1. Never trust a random number generator.
2. Never trust your code to be correct.
3. Never trust a previous author to have known all the literature.
4. Never trust your memory as to where you put that data (and how it was generated).
5. Never trust your computer to remain unchanged.
6. Never trust backup media or Web sites to remain readable indefinitely.
7. Never trust a so-called expert on experimental analysis.

References

[ABCC98] D. Applegate, R. Bixby, V. Chvatal, and W. Cook. On the solution of traveling salesman problems. *Documenta Mathematica Journal der Deutschen Mathematiker-Vereinigung*, ICM III:646–656, 1998. Additional information and source code for the `Concorde` package is available at `http://www.keck.caam.rice.edu/concorde.html`.

[Ben91] J. L. Bentley. Tools for experiments on algorithms. In R. F. Rashid, editor, *CMU Computer Science: A 25th Anniversary Commemorative*, pages 99–23. ACM Press, New York, 1991.

[Ben92] J. L. Bentley. Fast algorithms for geometric traveling salesman problems. *ORSA Journal on Computing*, 4(4):387–411, 1992.

[BGK+95] R. S. Barr, B. L. Golden, J. P. Kelly, M. G. C. Resende, and W. R. Stewart. Designing and reporting on computational experiments with heuristic methods. *Journal of Heuristics*, 1(1):9—32, 1995.

[BJLM83] J. L. Bentley, D. S. Johnson, F. T. Leighton, and C. C. McGeoch. An experimental study of bin packing. In *Proceedings of the 21st Annual Allerton Conference on Communication, Control, and Computing*, pages 51–60, Urbana, 1983. University of Illinois.

[CGR94] B. V. Cherkassky, A. V. Goldberg, and T. Radzik. Shortest paths algorithms: Theory and experimental evaluation. In *Proceedings of the Fifth ACM-SIAM Symposium on Discrete Algorithms*, pages 516–525. ACM-SIAM, New York, Philadelphia, January 1994.

[Cle85] W. S. Cleveland. *Elements of Graphing Data*. Wadsworth, Monteray, CA, 1985.

[Cle93] W. S. Cleveland. *Visualizing Data*. Hobart Press, Summit, NJ, 1993.

[CS00] M. Coffin and M. J. Saltzman. Statistical analysis of computational tests of algorithms and heuristics. *INFORMS J. Computing*, 12(1):24–44, 2000.

[Don00] J. J. Dongarra. Performance of various computers using standard linear equations software. Technical report, No. CS-89-85, Computer Science Department, University of Tennesssee, Knoxville, TN, August 24, 2000. The currently most up-to-date version should be available from `http://www.netlib.org/benchmark/performance.ps`.

[FJMO95] M. L. Fredman, D. S. Johnson, L. A. McGeoch, and G. Ostheimer. Data structures for traveling salesmen. *J. Algorithms*, 18(3):432–479, 1995. Preliminary version: *Proceedings of the Fourth ACM-SIAM Symposium on Discrete Algorithms*, pages 145–154, SIAM, Philadelphia, 1993.

[HK70] M. Held and R. M. Karp. The traveling-salesman problem and minimum spanning trees. *Operations Research*, 18:1138–1162, 1970.

[HK71] M. Held and R. M. Karp. The traveling-salesman problem and minimum spanning trees: Part II. *Math. Prog.*, 1:6–25, 1971.

[Hoo93] J. Hooker. Needed: An empirical science of algorithms. *Operations Research*, 42(2):201–212, March-April 1993.

[JBMR03] D. S. Johnson, J. L. Bentley, L. A. McGeoch, and E. E. Rothberg. Near-optimal solutions to very large traveling salesman problems. Monograph, in preparation, 2003.

[JM97] D. S. Johnson and L. A. McGeoch. The traveling salesman problem: A case study in local optimization. In E. Aarts and J. K. Lenstra, editors, *Local Search in Combinatorial Optimization*, pages 215–310. John Wiley & Sons, 1997. Preliminary draft available at `http://www.research.att.com/~dsj/`.

[JMR96] D. S. Johnson, L. A. McGeoch, and E. E. Rothberg. Asymptotic experimental analysis for the held-karp traveling salesman bound. In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 341–350, Atlanta, January 1996.

[Joh73] D. S. Johnson. *Near-Optimal Bin Packing Algorithms.* PhD thesis, Project MAC TR-100, MIT, Cambridge, MA, June 1973.

[Joh90] D. S. Johnson. Local optimization and the traveling salesman problem. In *Proc. 17th Colloq. on Automata, Languages, and Programming*, pages 446–461. Lecture notes in Computer Science 443, Springer-Verlag, Berlin, 1990.

[Joh96] D. S. Johnson. A theoretician's guide to the experimental analysis of algorithms, 1996. Seven page postscript draft available at `http://www.research.att.com/~dsj/papers/exper.ps`.

[McG92] C. C. McGeoch. Analyzing algorithms by simulation: Variance reduction techniques and simulation speedups. *ACM Computing Surveys*, 245(2):195–212, June 1992.

[McG96] C. C. McGeoch. Toward an experimental method for algorithm simulation. *INFORMS Journal on Computing*, 1(1):1—15, Winter 1996.

[McG01] C. C. McGeoch. Experimental analysis of algorithms. In P. Pardalos and E. Romeijn, editors, *Handbook of Global Optimization, Volume 2: Heurstic Approaches.* Kluwer Academic, 2001. To appear.

[MM99] C. C. McGeoch and B. M. E. Moret. How to present a paper on experimental work with algorithms. *SIGACT News*, (113):85–90, December 1999. Also available at `www.cs.amherst.edu/~ccm/howto.ps`.

[PM96] A. G. Percus and O. C. Martin. Finite size and dimensional dependence in the euclidean traveling salesman problem. *Phys. Rev. Lett.*, 76(8):1188–1191, 1996.

[Rei81] G. Reinelt. TSPLIB–a traveling salesman problem library. *ORSA Journal on Computing*, 3:376–384, 1981. Website: `http://www.iwr.uni-heidelberg.de/iwr/comopt/software/TSPLIB95/`.

[RU01] R. L. Rardin and R. Uzsoy. Experimental evaluation of heuristic optimization algorithms: A tutorial. *J. Heuristics*, 7:262–304, 2001.

[Sho86] P. W. Shor. The average case analysis of some on-line algorithms for bin packing. *Combinatorica*, 6:179–200, 1986.

[Tuf83] E. R. Tufte. *The Visual Display of Quantitative Information.* Graphics Press, 1983.

Appendix

This appendix collects together into separate lists the "Suggestions," "Pitfalls," and "Pet Peeves," that were highlighted during the course of the discussions in Section 2.

Pitfalls

1. Dealing with dominated algorithms.
2. Devoting too much computation to the wrong questions.
3. Getting into an endless loop in your experimentation.
4. Start by using randomly generated instances to evaluate the behavior of algorithms. End up using algorithms to investigate the properties of randomly-generated instances.
5. Too much code tuning.
6. Lost code/data.

Suggestions

1. Think before you compute.
2. Use exploratory experimentation to find good questions.
3. Use variance reduction techniques.
4. Use bootstrapping to evaluate multiple-run heuristics.
5. Use self-documenting programs.
6. Use benchmark algorithms to calibrate machine speeds.
7. Use profiling to understand running times.
8. Display normalized running times.

Pet Peeves

1. Authors and referees who don't do their homework
2. Concentration on unstructured random instances.
3. The millisecond testbed.
4. The already-solved testbed.
5. Claiming inadequate programming time/ability as an excuse.
6. Supplied code that doesn't match a paper's description of it.
7. Irreproducible standards of comparison.
8. Using running time as a stopping criterion.
9. Using the optimal solution value as a stopping criterion.
10. Hand-tuned algorithm parameters.
11. The one-run study.
12. Using the best result found as an evaluation criterion.
13. The uncalibrated machine.

14. The lost testbed.
15. False precision.
16. Unremarked anomalies.
17. The *ex post facto* stopping criterion.
18. Failure to report overall running times.
19. Data without interpretation.
20. Conclusions without support.
21. Myopic approaches to asymptopia.
22. Tables without pictures.
23. Pictures without tables.
24. Pictures yielding too little insight.
25. Inadequately or confusingly labeled pictures.
26. Pictures with too much information.
27. Confusing pictorial metaphors.
28. The spurious trend line.
29. Poorly structured tables.
30. Making your readers do the arithmetic.
31. The undefined metric.
32. Comparing apples with oranges.
33. Detailed statistics on unimportant questions.
34. Comparing approximation algorithms as to how often they find optima.
35. Too much data.