DIMACS Series in Discrete Mathematics
and Theoretical Computer Science

Towards a Discipline of Experimental Algorithmics

Bernard M.E. Moret

ABSTRACT. The last 30 years have seen enormous progress in the design of algorithms, but comparatively little of it has been put into practice, even within academic laboratories. Indeed, the gap between theory and practice has continuously widened over these years. Moreover, many of the recently developed algorithms are very hard to characterize theoretically and, as initially described, suffer from large running-time coefficients. Thus the algorithms and data structures community needs to return to implementation as one of its principal standards of value; we call such an approach *Experimental Algorithmics*.

Experimental Algorithmics studies algorithms and data structures by joining experimental studies with the traditional theoretical analyses. Experimentation with algorithms and data structures is proving indispensable in the assessment of heuristics for hard problems, in the characterization of asymptotic behavior of complex algorithms, in the comparison of competing designs for tractable problems, in the formulation of new conjectures, and in the evaluation of optimization criteria in a multitude of applications. Experimentation is also the key to the transfer of research results from paper to production code, providing as it does a base of well-tested implementations.

We present our views on what is a suitable problem to investigate with this approach, what is a suitable experimental setup, what lessons can be learned from the empirical sciences, and what pitfalls await the experimentalist who fails to heed these lessons. We illustrate our points with examples drawn from our research on solutions for NP-hard problems and on comparisons of algorithms for tractable problems, as well as from our experience as reviewer and editor.

1. Introduction

Implementation, although perhaps not rigorous experimentation, was characteristic of early work in algorithms and data structures. Donald Knuth insisted on implementing every algorithm he designed and on conducting a rigorous analysis of the resulting code (in the famous MIX assembly language) [**25**], while other pioneers such as Floyd are remembered as much for practical "tricks" (e.g., the four-point method to eliminate most points in an initial pass in the computation of a convex hull, see, e.g. [**36**]) as for more theoretical contributions. Throughout the last 20

The author's work on various related projects has been supported by the Office of Naval Research under grants N00014-92-C-2144, N00014-95-1-1010, and N00014-97-1-0244, and by the National Science Foundation under grant ITR 00-81404.

years, Jon Bentley has demonstrated the value of implementation and testing of algorithms, beginning with his text on writing efficient programs [**3**] and continuing with his invaluable *Programming Pearls* columns in *Communications of the ACM*, now collected in a new volume [**6**], and his *Software Explorations* columns in the *UNIX Review*. Over 10 years ago, David Johnson, in whose work on optimization for NP-hard problems experimentation took pride of place, started the annual ACM/SIAM Symposium on Discrete Algorithms (SODA), which has sought, and every year featured a few, experimental studies. It is only in the last few years, however, that the algorithms community has shown signs of returning to implementation and testing as an integral part of algorithm development. Other than SODA, publication outlets remained rare until the late nineties: the *ORSA J. Computing* and *Math. Programming* have published several strong papers in the area, but the standard journals in the algorithm community, such as the *J. Algorithms*, *J. ACM*, *SIAM J. Computing*, and *Algorithmica*, as well as the more specialized journals in computational geometry and other areas, have been slow to publish experimental studies. (It should be noted that many strong experimental studies dedicated to a particular application have appeared in publication outlets associated with the application area; however, many of these studies ran tests to understand the data or the model rather than to understand the algorithm.) The online *ACM Journal Experimental Algorithmics* is dedicated to this area and is starting to publish a respectable number of studies. The two workshops targeted at experimental work in algorithms, the *Workshop on Algorithm Engineering* (WAE), held every late summer in Europe, and the *Workshop on Algorithm Engineering and Experiments* (ALENEX), held every January in the United States, are also attracting growing numbers of submissions. Support for an experimental component in algorithms research is growing among funding agencies as well. We may thus be poised for a revival of experimentation as a research methodology in the development of algorithms and data structures, a most welcome prospect, but also one that should prompt some reflection.

As we contemplate approaches based on (or at least making extensive use of) experimentation, we may want to reflect on the meanings of the two adjectives used to denote such approaches. According to the *Collegiate Webster*, these adjectives are defined as follows.

- **experimental**: 1. relating to or based on experience; 2. founded upon experiments; 3. serving the ends of experimentation; 4. tentative.
- **empirical**: 1. relying on experience or observation alone; 2. based on experience or observation; 3. capable of being verified or disproved through experience or observation.

Certainly, part (2) of the definition of "experimental" and parts (2) and (3) of the definition of 'empirical" capture much of what most of us would agree is essential in the use of experiments. Unfortunately, both words have problematic connotations: the "tentative" meaning of "experimental" and the exclusion of theory in the first definition of "empirical." A completely empirical approach may be perfectly suitable for a natural science, where the final arbiter is nature as revealed to us through experiments and measurements and where the "laws of nature" can at best be approximated through models, but it is incomplete in the artificial and mathematically precise world of computing, where the behavior of most algorithms or data structures can, at least in principle, be characterized analytically. Natural scientists run experiments because they have no other way of learning from nature. In contrast, algorithm designers run experiments mostly because an analytical characterization is too hard to achieve in practice. (Much the same is done by computational scientists in physics, chemistry, and biology, but typically their aim is to analyze new data or to compare the predictions given by a model with the measurements made from nature, not to characterize the behavior of an algorithm.) Algorithm designers are measuring the actual algorithm, not a model, and the results are not assessed against some gold standard (nature), but simply reported as such or compared with other experiments of the same type. (Of course, we do also build models and gauge them against the real system; typically, our models are mathematical functions that characterize some aspect of the algorithm, such as its asymptotic running time.)

Why this epistemological digression? Because it points to the necessity of both learning from the natural sciences, where experimentation has been used for centuries and where the methodology known as "the scientific method" has been developed to optimize the use of experiments, and of staying aware of the fundamental difference between the natural sciences and computer science, since the goal of experimentation in algorithmic work differs in important ways from that in the natural sciences.

2. Background and Motivation

For over thirty years, the standard mode of theoretical analysis, and thus also the main tool used to guide new designs, has been the asymptotic analysis ("big Oh" and "big Theta") of worst-case behavior (running time or quality of solution). The asymptotic mode eliminates potentially confusing behavior on small instances due to start-up costs and clearly shows the growth rate of the running time. The worst-case (per operation or amortized) mode gives us clear bounds and also simplifies the analysis by removing the need for any assumptions about the data. The resulting presentation is easy to communicate and reasonably well understood, as well as machine-independent. However, we pay a heavy price for these gains:

- The range of values in which the asymptotic behavior is clearly exhibited ("asymptopia," as it has been named by many authors) may include only instance sizes that are well beyond any application. A typical example is the algorithm of Fredman and Tarjan for minimum spanning trees. Its asymptotic worst-case running time is $O(|E|\beta(|E|, |V|))$—where $\beta(m, n)$ is given by $\min\{i \mid \log^{(i)} n \leq m/n\}$, so that, in particular, $\beta(n, n)$ is just $\log^* n$. This bound is much better for dense graphs than that of Prim's algorithm, which is $O(|E| \log |V|)$, but experimentation [**38**] verifies that the crossover point occurs for dense graphs with well over a billion edges—beyond the size of any reasonable data set.
- In another facet of the same problem, the constants hidden in the asymptotic analysis may prevent any practical implementation from running to completion, even if the growth rate is quite reasonable. An extreme example of this problem is provided by the theory of graph minors: Robertson and Seymour (see [**42**]) gave a cubic-time algorithm to determine whether a given graph is a minor of another, but the proportionality constants are

gigantic—on the order of 10^{150}—and have not been substantially lowered yet, making the algorithm entirely impractical.

- The worst-case behavior may be restricted to a very small subset of instances and thus not be at all characteristic of instances encountered in practice. A classic example here is the running time of the simplex method for linear programming; for over thirty years, it has been known that the worst-case behavior of this method is exponential, but also that its practical running time is typically bounded by a low-degree polynomial [**1**]. Approximation algorithms with performance guarantees often present the same pattern: the approximations they return are often much better than the bounds indicate, even when these bounds are known to be tight.
- Even in the absence of any of these problems, deriving tight asymptotic bounds may be very difficult. All optimization metaheuristics for NP-hard problems (such as simulated annealing or genetic algorithms) suffer from this drawback: by considering a large number of parameters and a substantial slice of recent execution history, they create a complex state space which is very hard to analyze with existing methods, whether to bound the running time or to estimate the quality of the returned solution. Another classic example is the deceptively simple Union-Find data structure (see, e.g., [**36**], Section 3.2): the combination of Union by rank (or size) and path compression was known to yield very efficient behavior, yet its exact characterization eluded researchers for many years, until Tarjan proved tight bounds [**47**]. (Of course, in this particular case, experimentation would have proved futile, since no amount of experimentation, on an conceivable dataset, could show supralinear growth in the running time.)

These are the most obvious drawbacks. A more insidious drawback, yet one that could prove much more damaging in the long term, is that worst-case asymptotic analysis tends to promote the development of "paper-and-pencil" algorithms, that is, algorithms that never get implemented. This problem compounds itself quickly, as further developments rely on earlier ones, with the result that many of the most interesting algorithms published over the last five years rely on several layers of complex, unimplemented algorithms and data structures. In order to implement one of these recent algorithms, a computer scientist would face the daunting prospect of developing implementations for all successive layers. Moreover, the "paper-and-pencil" algorithms often ignore issues critical in making implementations efficient: low-level algorithmic issues (from elementary ideas such as the use of sentinels to more elaborate ones such as the use of "sacks" in sophisticated priority queues [**38**]) and architecture-dependent issues (particularly caching issues); the implementer will have to resolve these issues "on the fly," possibly with very poor results. Transforming paper-and-pencil algorithms into efficient and useful implementations is today referred to as *algorithm engineering*; case studies show that the use of algorithm engineering techniques, all of which are based on experimentation, can improve the running time of code by up to three orders of magnitude [**35**] as well as yielding robust libraries of data structures with minimal overhead, as done in the LEDA library [**33, 34**]. Algorithm engineering is reviving many of the approaches used by computing pioneers such as Floyd and Knuth, who combined theoretical insights with practical ideas and even machine-dependent "tricks" to derive efficient algorithms that performed well in both theory and practice.

There is no reason to abandon asymptotic worst-case analysis: it has served the community very well for over thirty years and led to major algorithmic advances. But there is a definite need to supplement it with experimentation, which implies that most algorithms should be implemented, not just designed. Many algorithms are in fact quite difficult to implement—because of their intricate nature and also because the designers described them at a very high level. Many examples of such can be found in computational geometry: Chazelle's linear-time simplicity testing [**9**], Chazelle's convex decomposition algorithm [**8**], and Chang and Yap's "potato-peeling" algorithm [**7**] all are very intricate and remain—to my knowledge—unimplemented. But the practitioner is not the only one who stands to benefit from implementation: often an implementation forces the theoretician to face issues glossed over in the high-level design phase. Resolving these issues may bring about a deeper understanding of the algorithm and a resulting simplification or more modestly may lead the theoretician to new conjectures. Major theoretical breakthroughs, such as Chazelle's linear-time simplicity test or Robertson and Seymour's polynomial-time minor test, are their own justification, but incremental results should be judged on more practical grounds: do they lead to better, faster, more robust implementations? Finally, experimentation should also test the usefulness of a problem formulation: not only is it too easy to devise cute puzzles of dubious interest, but even in problems motivated by real applications we need to find out what criteria to optimize and what parameters to take into account. Validating a model or a particular criterion requires large-scale experimentation; in a recent study, we used over 20 years of CPU time (on modern workstations) to obtain data on the quality of phylogeny reconstructions produced by quartet-based algorithms [**45**].

3. Modes of Empirical Assessment

We can classify modes of empirical assessment into a number of non-exclusive categories:

- Checking for accuracy or correctness in extreme cases (e.g., standardized test suites for numerical computing).
- Assessing the quality of heuristics for the approximate solution of NP-hard problems (and, incidentally, generating hard instances).
- Measuring the running time of exact algorithms on real-world instances of NP-hard problems.
- Comparing the actual performance of competing algorithms for tractable problems and characterizing the effects of algorithm engineering.
- Discovering the speed-up achieved by parallel algorithms on real machines.
- Investigating and refining models and optimization criteria—what should be optimized? and what parameters matter?
- Testing the quality and robustness of simulations, of optimization strategies for complex systems, etc.

The first category has reached a high level of maturity in numerical computing, where standard test suites are used to assess the quality of new numerical codes. Similarly, the operations research community has developed a number of test cases for linear program solvers. We have no comparable emphasis to date in combinatorial and geometric computing. The last category is the target of large efforts within the Department of Defense, whose increasing reliance on modeling and simulation has placed it at the forefront of a movement to develop validation and verification tools; the algorithm community can help by providing certification levels for the various data structures and optimization algorithms embedded within large simulation systems. Studying speed-ups in parallel algorithms remains for now a rather specialized endeavor, in part because of the dedicated nature of software (which typically cannot be run on another machine without major performance losses) and because of our attending lack of a good model of parallel computation. Investigation and refinement of models and optimization criteria is of major concern today, particularly in areas such as computational biology and computational chemistry. While many studies are published, most demonstrate a certain lack of sophistication in the conduct of the computational studies—suffering as they do from various sources of errors. We eschew a lengthy discussion of this important area and instead present sound principles and illustrate pitfalls in the context of the two categories that have seen the bulk of research to date in the algorithms community. Most of these principles and pitfalls can be related directly to the testing and validation of discrete optimization models in the natural sciences.

3.1. Assessment of Competing Algorithms and Data Structures for Tractable Problems. The goal here is to measure the actual performance of competing algorithms for well-solved problems. This is fairly new work in combinatorial algorithms and data structures, but common in Operations Research; early (1960s) work in data structures typically included code and examples, but no systematic study. The 1970s were a dry time in the area, until Sedgewick's work on quicksort [**44**]. More recent and comprehensive work began with Bentley's many contributions in his *Programming Pearls* (starting in 1983 [**4**]), then with Jones' comparison of data structures for priority queues [**24**], Dandamudi and Sorenson's empirical comparison of k-d tree implementations [**14**], and Stasko and Vitter's combination of analytical and experimental work in the study of pairing heaps [**46**]. The first experimental study on a large scale was that of Moret and Shapiro on sorting algorithms [**36**] (Chapter 8), followed by that of the same authors on algorithms for constructing minimum spanning trees [**36, 38**]. In 1991, Johnson and others initiated the very successful DIMACS Computational Challenges, the first of which [**22**] focused on network flow and shortest path algorithms, indirectly giving rise to several modern, thorough studies, by Cherkassky *et al.* on shortest paths [**11**], by Cherkassky *et al.* on the implementation of the push-relabel method for matching and network flows [**13, 12**], and by Goldberg and Tsioutsiouliklis on cut trees [**18**]. The DIMACS Computational Challenges (the fifth, in 1996, focused on another tractable problem, priority queues and point location data structures) have served to highlight work in the area, to establish common data formats (particularly formats for graphs and networks), and to set up the first tailored test suites for a host of problems.

Recent conferences (such as the *Workshop on Algorithm Engineering* and the *Workshop on Algorithm Enginering and Experiments*[1]) have emphasized the need to develop libraries of robust, well-tested implementations of the basic discrete

[1]See the front page of the *ACM J. Experimental Algorithmics* at `www.jea.acm.org` for links to these conferences.

and combinatorial algorithms, a task that only the LEDA project [**33, 34**] has successfully undertaken to date.

Much interest has focused over the last 3-4 years on the question of tailoring algorithms and implementations to the cache structure and policies of the architecture. Pioneering studies by Ladner and his coworkers [**27, 28, 29**] established that such optimization was feasible, algorithmically interesting, and worthwhile, even for such old friends as sorting algorithms [**2, 29, 40, 49**] and priority queues [**28, 43**]; indeed, even matrix multiplication, which has been optimized in numerical libraries for over 30 years, is amenable to such techniques [**16**]. Ad hoc reduction in memory usage and improvement in patterns of memory addressing have been reported to gain speedups of as much as a factor of 10 [**35**]. The related, and much better studied, model of out-of-core computing, as pioneered by Vitter and his coworkers [**48**], has inspired new work in cache-aware and cache-independent algorithm design.

3.2. Assessment of Heuristics. Here the goal is to measure the performance of heuristics on real and artificial instances and to improve the theoretical understanding of the problem, presumably with the aim of producing yet better heuristics or proving that current heuristics are optimal. By performance is implied both the running time and the quality of the solution produced.

Since the behavior of heuristics is very difficult to characterize analytically, experimental studies have been the rule. The Operations Research community, which has a long tradition of application studies, has slowly developed some guidelines for experimentation with integer programming problems (see [**1**], Chapter 18). The first large-scale combinatorial study to include both real-world and generated instances was probably our work on the minimum test set problem [**37**], but other large-scale studies were published in the same time frame, most notably the classic and exemplary study of simulated annealing by David Johnson's group [**20, 21**]. The Second DIMACS Computational Challenge [**23**] was devoted to satisfiability, graph coloring, and clique problems and thus saw a large collection of results in this area. The ACM/SIAM Symposium on Discrete Algorithms (SODA) has included a few such studies in each of its dozen events to date, such as the study of cut algorithms by Chekuri *et al.* [**10**]. The Traveling Salesperson problem has seen large numbers of experimental studies (including the well publicized study of Jon Bentley [**5**]), made possible in part by the development of a library of test cases [**41**]. Graph coloring, whether in its NP-hard version of chromatic number determination or in its much easier (yet still challenging) version of planar graph coloring, has seen much work as well; the second study of simulated annealing conducted by Johnson's group [**20**] discussed many facets of the problem, while Morgenstern and Shapiro [**39**] provided a detailed study of algorithms to color planar graphs.

Challenges in this area include the generation of meaningful test instances and the derivation of strong lower bounds that enable us to assess the quality of the heuristic solutions.

4. Worthwhile Problems

In view of the preceding, what should researchers in the area be working on? We propose below a partial list and briefly discuss the reasons for our choices.

4.1. Testing and improving algorithms for hard problems. Understanding how a heuristic works to cut down on computational time is generally too difficult to achieve through formal derivations; much the same often goes for bounding the quality of approximations obtained with many heuristics.[2] Yet both aspects are crucial in evaluating performance and in helping us design better heuristics.

In the same vein, understanding when an exact algorithm runs quickly is often too difficult for formal methods. It is much easier to characterize the worst-case running time of an algorithm than to develop a classification of input data in terms of a few parameters that suffice to predict the actual running time in most cases. Experimentation can help us assess the performance of an algorithm on real-world instances (a crucial point) and develop at least *ad hoc* boundaries between instances where it runs fast and instances that exhibit the exponential worst-case behavior.

4.2. Comparing existing algorithms and data structures for tractable problems. Our task is somewhat easier with algorithms for tractable problems than with heuristics for intractable problems, yet characterizing the behavior of either on real-world instances is generally very hard simply because we often lack the crucial instance parameters with which to correlate running times. Experimentation can quickly pinpoint good and bad implementations and whether theoretical advantages are retained in practice. In the process, newer insights may be gleaned that might enable a refinement or simplification of the algorithm. Experimentation can also enable us to determine the actual constants in the running time analysis; determining such constants beforehand is quite difficult (see [**17**] for a possible methodology), but a simple regression analysis from the data can gives us quite accurate values. Experimental studies naturally include caching effects, whereas adding those into the analysis in a formal manner is very challenging.

4.3. Algorithm engineering. Many of the goals described under the previous two rubrics also fall under this heading. More specifically, however, algorithm engineering seeks to produce the most efficient, as well as most usable, implementation possible. A cycle of testing and refinement, supported by profilers and other measurement tools, is the key to removing bottlenecks in running time and to obtaining a good balance among the various routines. Reducing the use of memory, changing the addressing patterns to make the implementation more cache-friendly (which may imply some data structure changes as well), unrolling loops to keep variables in registers or L1 cache, and taking a good look at what information is really needed and how it is used at each stage in the algorithm, are all measures that can lead to significant improvements in running time: in a recent effort, we used all of these techniques to obtain a speed-up by three orders of magnitude [**35**].

4.4. Supporting and refining conjectures. Any theoretician knows the pangs of committing to a research question without being too sure of the outcome and the frustration of attempting to prove a statement that might not even be true. Having a means of testing a conjecture over a range of instances might, in the best case, set one's mind at rest and, in the worst case, avoid a lot of wasted work. More importantly, good experiments are a rich source of new conjectures and theorems.

[2]Of course, we have many elegant results bounding the worst-case performance of approximation algorithms, but many of these bounds, even when attainable, are overly pessimistic for real data.

4.5. Developing tools. Anyone contemplating the coding of a library module for some data structure or basic algorithm must take reasonable precautions to ensure that her implementation will be as efficient as possible and to document conditions under which it will perform well or poorly. Testing the implementation on a range of machines with various compilers and environments will increase confidence in the characterizations. For instance, extensive testing of the LEDA modules by their developers and many others led to a 2- to 3-fold reduction in their running-time overhead [**32**].

At a more basic level, we need flexible tools to collect data (how do we measure cache behavior, for instance?) and analyze it. We should not underestimate the value of experimentation with algorithms as a discovery tool; in order to make such experimentation even more valuable, statistical tools with good graphical and animation capabilities are urgently needed.

4.6. Conducting experiments to assess the relevance of optimization criteria. The pure theoretician often has only one answer when asked why (s)he worked on a problem: because it was intriguing. But it is fatally easy to generate volumes of intriguing, unsolved optimization problems; before committing scarce resources to their solution, it behooves us to evaluate their importance and relevance as well as we can. Applications provide sufficient justification to study problems; but the model used by the algorithm designer, which is necessarily a simplification of the real problem, must be validated through large-scale experimentation, preferably using a mix of real data and simulations.

5. Experimental Setup

How should an experimental study be conducted, once a topic has been identified? Surely the most important criterion to keep in mind is that an experiment is run either as a discovery tool or as a means to answer specific questions. Experiments as explorations are common to all endeavors, in computing, in the sciences, and indeed in any human activity; the setup is essentially arbitrary—in particular it should not be allowed to limit one's creativity. So we shall focus instead on experiments as means to answer specific questions—the essence of the scientific method used in all physical sciences. In this methodology, we begin by formulating a hypothesis or a question, then set about gathering data to test or answer it, while ensuring reproducibility and significance. In terms of experiments with algorithms, these characteristics give rise to the following procedural rules—but the reader should keep in mind that most researchers would mix the two activities for quite a while before running their "final" set of experiments:

- Begin the work with a clear set of objectives: which questions will you be asking, which statements will you be testing?
- Once the experimental design is complete, simply gather data. (No alterations are to be made until all data have been gathered, so as to avoid bias or drift.)
- Analyze the data to answer only the original objectives. (Later, consider how a new cycle of experiments can improve your understanding.)

At all stages, we should beware of a number of potential pitfalls, including various biases due to:

- The choice of machine (caching, addressing, data movement), of language (register manipulation, built-in types), or of compiler (quality of optimization and code generation).
- The quality of the coding (consistency and sophistication of programmers).
- The selection or generation of instances (we must use sufficient size and variety to ensure significance).
- The method of analysis (many steps can be taken to improve the significance of the results as well as to bring out trends).

Caching, in particular, may have very strong effects when comparing efficient algorithms. For instance, in our study of MST algorithms, we observed 3:1 ratios of running time depending on the order in which the adjacency lists were stored. Pioneering studies by Ladner and his students [**27, 28, 29**] have quantified many aspects of caching and offer suggestions on how to work around (or take advantage of) caching effects.

Other typical pitfalls that arise in experimental work with algorithms include

- Uninteresting work: comparing programming languages or specific platforms, in particular unusual ones; comparing algorithms with widely different behavior (linear and quadratic, say); etc.
- Bad setup: testing up to some fixed running time or space without verifying whether the asymptotic behavior has manifested; testing too few instances; using rough code without any attempt at optimization and measuring running times; using "found code" without any documentation (a temptation these days on the net); ignoring existing test suites; ignoring existing and widely used libraries; etc.
- Bad analysis or presentation: discarding data that do not fit without any explanation or even warning; presenting all of the data without analysis; using comparisons to undefined "standards" (e.g., to the system sort routine).

Johnson [**19**] offers a much more detailed list of the various problems he has observed in experimental studies, particularly those dealing with heuristics for hard optimization problems.

Most of these pitfalls can be avoided with the type of routine care used by experimentalists in any of the natural sciences. However, we should point out that confounding factors can assume rather subtle forms. Knuth long ago pointed out curious effects of apparently robust pseudorandom number generators (see [**25**], Vol. II); the creation of unexpected patterns as an artifact of a hidden routine (or, in the case of timing studies, as an artifact of interactions between the memory hierarchy and the code) could easily lead the experimenter to hypothesize nonexistent relationships in the data. The problem is compounded in complex model spaces, since obtaining a fair sampling of such a space is always problematic. Thus it pays to go over the design of an experimental study a few times just to assess its sensitivity to potential confounding factors—and then to examine the results with the same jaundiced eye.

6. What to Measure?

One of the key elements of an experiment is the metrology. What do we measure, how do we measure it, and how do we ensure that measurements do not interfere with the experiments? If there is one universal piece of advice in this area, it is *always look beyond the obvious measures!* Obvious measures may include the value of the solution (for heuristics and approximation algorithms), the running time (for almost every study), the running space, etc. These measures are indeed useful, but a good understanding of the algorithm is unlikely to emerge from such global quantities. We also need structural measures of various types (number of iterations; number of calls to a crucial subroutine; etc.), if only to serve as a scale for determining such things as convergence rates. Knuth [**26**] has advocated the use of *mems*, or memory references, as a structural substitute for running time. Other authors have used the number of comparisons, the number of data moves (both classical measures for sorting algorithms), the number of assignments, etc. Most programming environments offer some type of profiler, a support system that samples code execution at fixed intervals and sets up a profile of where the execution time was spent (which routines used what percentage of the CPU time) as well as of how much memory was used; with suitable hardware support, profilers can also report caching statistics. Profiling is invaluable in algorithm engineering—multiple cycles of profiling and revising the most time-consuming routines can easily yield gains of one to two orders of magnitude in running time.

In our own experience, we have found that there is no substitute, when evaluating competing algorithms for tractable problems, for measuring the actual running time; indeed, time and mems measurements, to take one example, may lead one to entirely different conclusions. However, the obvious measures are often the hardest to interpret as well as the hardest to measure accurately and reproducibly. Running time, for instance, is influenced by caching, which in turn is affected by any other running processes and thus effectively not reproducible exactly. In the case of competing algorithms for tractable problems, the running time is often extremely low (we can obtain a minimum spanning tree for a sparse graph of a million vertices in much less than a second on a typical desktop machine), so that the granularity of the system clock may create problems—this is a case where it pays to repeat the entire algorithm many times over on the same data, in order to obtain running times with at least two digits of precision. In a similar vein, measuring the quality of a solution can be quite difficult, due to the fact that the optimal solution can be very closely approached on instances of small to medium size or due to the fact that the solution is essentially a zero-one decision (as in determining the chromatic index of a graph or the primality of a number), where the appropriate measure is statistical in nature (how often is the correct answer returned?) and thus requires a very large number of test instances.

7. How to Present and Analyze the Data

Perhaps the first requirement in data presentation is to ensure reproducibility by other researchers: we need to describe in detail what instances were used (how they were generated or collected), what measurements were collected and how, and, preferably, where the reader can find all of this material on-line. The second requirement is rather obvious, but often ignored for all that: we cannot just discard what appear to be anomalies, at least not unless we can explain their presence; an anomaly without an explanation is not an error, but an indicator that something unusual (and possibly interesting) is going on. We have already mentioned several times that every effort should be made to minimize the influence of the environment:

platform, coding, compiling, paging, caching, etc., through cross-checking across multiple platforms and environments, through the use of normalization routines, and through environmental precautions (such as running on otherwise quiescent machines).

The data should then be analyzed with suitable statistical methods. Since attaining levels of statistical significance may be quite difficult in the large state spaces we commonly use, various techniques to make the best use of available experiments should be applied (see McGeoch's excellent survey [**30**] for a discussion of several such methods). Cross-checking the measurements with any available theoretical results, especially those that attempt to predict the actual running time (such as the "equivalent code fragments" approach of [**17**]), is crucial; any serious discrepancy needs to be investigated.

Finally, the data need to be presented to the readers in a form that humans can easily process—not in tabular form, not as raw plots with multiple crossing curves, but with suitable scaling and normalization and with the use of good graphics, colors, etc. Normalization and scaling are a particularly important part of both analysis and presentation: not only can they bring out trends not otherwise evident, but they can help in filtering out noise and thus increasing the significance of the results. Animations can convey enormous amounts of information very succinctly, so consider providing such if the work needed to produce them is not excessive.

8. Illustration: Algorithms for Constructing a Minimum Spanning Tree

We shall not repeat here the results given in [**38**], but rather highlight the problems encountered during the study and some of the solutions we found to be effective. We studied MST algorithms because of their practical importance, because instances encountered in practice can be very large, and because the implementer faces a very large number of algorithmic choices, each with its own choice of supporting data structures. In 1989, when we started the study, we had at least the following choice of algorithms: Kruskal's (with a priority queue, with prior sorting, or with sorting on demand), Prim's (with any of a large number of priority queues, from binary heaps to rank- and run-relaxed heaps), Cheriton and Tarjan's (with and without the lazy variation) Fredman and Tarjan's, Gabow *et al.*'s, and the entirely different algorithm of Fredman and Willard; to this list we could now add newer and asymptotically faster algorithms by Klein and Tarjan, by Karger, by Chazelle, and by Pettie and Ramachandran. Prim's algorithm, the most commonly used (for good reason, as our study demonstrated), could in turn be implemented with any of a dozen or more priority queue designs, each with its own implementation choices. Very few of these choices had been implemented at that time.

We ran an experimental study using three different platforms (two CISC machines and one RISC machine) and multiple languages and compilers, but with one programmer writing all of the code, so as to keep the level of coding consistent throughout. We explored low-level decisions (pointers vs. array indices, data moves vs. indirection, etc.) before committing to specific implementations. We used five different graph families in the tests and also constructed specific worst-case families with adversaries; all of our families included very large graphs (up to a million vertices and over a million edges). We ran at least 20 instances at each size, checking independent series of experiments for consistency in the results. Finally, we took precautions from the start to minimize the effects of paging (easy) and of caching (hard).

Our data collection and analysis had four goals: (i) to minimize any residual effects of caching and any other machine dependencies; (ii) to normalize running times across machines; (iii) to gauge the influence of lower-order terms and to verify the asymptotic behavior; and (iv) to visualize quickly the relative efficiency of each algorithm for each type and size of graph. We realized all four goals at once by the simple strategy of normalizing, independently on each platform, the running times measured for the various MST algorithms by the running times of a simple, linear-time procedure with roughly similar memory reference patterns—in our case a procedure that counted the number of edges of the graph by traversing the adjacency lists. The similar memory addressing patterns canceled out most of the caching effects; the similar work in dereferencing pointers canceled out most of the CISC machines peculiarities; and the direct comparison to the (then unattainable) lower bound of a linear-time procedure immediately showed the asymptotic behavior and highlighted the relative efficiency of each algorithm.

Early in the implementation phase, we realized that Fibonacci heaps and relaxed heaps were not competitive. We followed a suggestion made by Driscoll *et al.* [**15**] about relaxed heaps: to group nodes into larger units so that changes in key value would most often be resolved within a unit and not require restructuring the heap; we implemented this idea, which we called *sacks*, for other types of heaps. This was a crucial decision for Fibonacci heaps, which became much more competitive with the addition of sacks—a new result that could only have come about through implementation.

At the conclusion of our work, we had comforting findings for the practitioner: the fastest algorithm by far was also the simplest, Prim's, implemented with pairing heaps or simple binary heaps. The more sophisticated implementations could not pay off for reasonable graph sizes, nor could the more sophisticated algorithms. But we also had a sobering report: our last implementations of Prim's algorithm with Fibonacci heaps were nearly ten times faster than our first! Thus even experienced programmers who understand the details of their data structures and algorithms can refine implementations to the point of evolving entirely new conclusions—a key aspect of algorithm engineering.

This study, along with an earlier study on sorting algorithms, enables us to draw some conclusions regarding experimental studies of algorithms for well-solved problems:

- Multi-machine, multi-compiler trials are needed. The preference of one architecture for data moves over indirection, for instance, could easily mask other effects, as could the highly variable details of the caching system. The first DIMACS challenge used the simple strategy of running a collection of benchmarks on each platform to enable comparison of results, but more sophisticated approaches are required, particularly with respect to cache-awareness. One attractive possibility is normalization by the running time of a known routine with well understood characteristics, but even this strategy is limited to relatively simple programs.
- A very large range of sizes is indispensable. Since the algorithms compared are all efficient and since sophisticated algorithms tend to demonstrate their asymptotic behavior for larger sizes than simpler algorithms, we

should run our tests up to the largest sizes that can be accommodated on our platforms, even if these sizes may exceed any likely to be encountered in practice. A large range of sizes will also help visualizing the asymptotic behavior and may uncover unexpected problems attributable to caching.
- Extreme care must be used when generating instances. This problem is particularly acute when instances are defined by multiple parameters, as in graphs and networks: large numbers of different families can be defined, with potentially very different behaviors. We should ensure that realistic instances are being generated, that large instances generated with pseudo-random number generators do not present artificial patterns caused by problems with the generator, and also that, whenever possible, some worst-case families are included in the study.[3]
- Real datasets should always be used. Few data generators can accurately reproduce the distribution of real data. Researchers have long known that real instances of hard optimization problems can be surprisingly easy to solve (as well as, more rarely, surprisingly hard), but much the same can be said of instances of tractable problems.
- Normalization by a suitable baseline routine is very successful in smoothing out variations in architecture and caching, as well as in highlighting the asymptotic behavior and relative efficiency of the competing algorithms. Whenever our competing algorithms are closely tied, data presentation is of crucial importance.

9. Conclusions

Experimentation should become once again the "gold standard" in algorithm design, for several compelling reasons:

- Experimentation can lead to the establishment of well tested and well documented libraries of routines and instances.
- Experimentation can bridge the gap between practitioner and theoretician.
- Experimentation can help theoreticians develop new conjectures and new algorithms, as well as a deeper understanding (and thus perhaps a cleaner version) of existing algorithms.
- Experimentation can point out areas where additional research is most needed.

However, experimentation in algorithm design needs some methodological development. While it can and, to a large extent, should seek inspiration from the natural sciences, its different setting (a purely artificial one in which the experimental procedure and the subject under test are unavoidably mixed) requires at least extra precautions. Fortunately, a number of authors have blazed what appear to be a good trail to follow; hallmarks of good experiments include:

- clearly defined goals;

[3]We observed one curious and totally unexpected behavior with generated data for the MST problem. One of our families of datasets consists of worst-case instances for Prim's algorithm run with binary heaps—the data are created by running the algorithm and picking values that maximize the number of elementary heap manipulations. These data, when fed to the version of Kruskal's algorithm that uses quicksort (implemented with a median-of-three strategy for choosing the partitioning element) uniformly caused quicksort to exhibit quadratic behavior!

- large-scale testing, both in terms of a range of instance sizes and in terms of the number of instances used at each size;
- a mix of real-world instances and generated instances, including any significant test suites in existence;
- clearly articulated parameters, including those defining artificial instances, those governing the collection of data, and those establishing the test environment (machines, compilers, etc.);
- statistical analyses of the results and attempts at relating them to the nature of the algorithms and test instances; and
- public availability of instances and instance generators to allow other researchers to run their algorithms on the same instances and, preferably, public availability of the code for the algorithms themselves.

References

[1] Ahuja, R.K., Magnanti, T.L., and Orlin, J.B. *Network Flows.* Prentice Hall, NJ, 1993.

[2] Arge, L., Chase, J., Vitter, J.S., and Wickremesinghe, R., "Efficient sorting using registers and caches," *Proc. 4th Workshop on Algorithm Eng. WAE 2000*, to appear in LNCS series, Springer Verlag (2000).

[3] Bentley, J.L. *Writing Efficient Programs.* Prentice-Hall, Englewood Cliffs, NJ, 1982.

[4] Bentley, J.L., "Programming pearls: cracking the oyster," *Commun. ACM* **26**, 8 (1983), 549–552.

[5] Bentley, J.L. Experiments on geometric traveling salesman heuristics. AT&T Bell Laboratories, CS TR 151, 1990.

[6] Bentley, J.L. *Programming Pearls.* 2nd ed., ACM Press, 1999.

[7] Chang, S., and Yap, C.K., "A polynomial solution for potato-peeling and other polygon inclusion and enclosure problems," *Discrete & Comput. Geom.* **1** (1986), 155–182.

[8] Chazelle, B., "Convex decompositions of polyhedra, a lower bound and worst-case optimal algorithm," *SIAM J. Comput.* **13** (1984), 488–507.

[9] Chazelle, B., "Triangulating a simple polygon in linear time," *Discrete & Comput. Geom.* **6** (1991), 485–524.

[10] Chekuri, C.S., Goldberg, A.V., Karger, D.R., Levine, M.S., and Stein, C., "Experimental study of minimum cut algorithms," *Proc. 8th ACM/SIAM Symp. on Discrete Algs. SODA97*, SIAM Press (1997), 324–333.

[11] Cherkassky, B.V., Goldberg, A.V., and Radzik, T., "Shortest paths algorithms: theory and experimental evaluation," *Math. Progr.* **73** (1996), 129–174.

[12] Cherkassky, B.V., and Goldberg, A.V., "On implementing the push-relabel method for the maximum flow problem," *Algorithmica* **19** (1997), 390–410.

[13] Cherkassky, B.V., Goldberg, A.V., Martin, P, Setubal, J.C., and Stolfi, J., "Augment or push: a computational study of bipartite matching and unit-capacity flow algorithms," *ACM J. Exp. Algorithmics* **3**, 8 (1998), `www.jea.acm.org/1998/CherkasskyAugment/`.

[14] Dandamudi, S.P., and Sorenson, P.G., "An empirical performance comparison of some variations of the k-d tree and bd tree," *Int'l J. Computer and Inf. Sciences* **14**, 3 (1985), 134–158.

[15] Driscoll, J.R., Gabow, H.N., Shrairman, R., and Tarjan, R.E., "Relaxed heaps: an alternative to Fibonacci heaps with applications to parallel computation," *Commun. ACM* **11** (1988), 1343–1354.

[16] Eiron, N., Rodeh, M., and Stewarts, I., "Matrix multiplication: a case study of enhanced data cache utilization," *ACM J. Exp. Algorithmics* **4**, 3 (1999), `www.jea.acm.org/1999/EironMatrix/`.

[17] Finkler, U., and Mehlhorn, K., "Runtime prediction of real programs on real machines," *Proc. 8th ACM/SIAM Symp. on Discrete Algs. SODA97*, SIAM Press (1997), 380–389.

[18] Goldberg, A.V., and Tsioutsiouliklis, K., "Cut tree algorthms: an experimental study," *J. Algs.* **38**, 1 (2001), 51–83.

[19] Johnson, D.S., "A theoretician's guide to the experimental analysis of algorithms," this volume.

[20] Johnson, D.S., Aragon, C.R., McGeoch, L.A., and Schevon, C., "Optimization by simulated annealing: an experimental evaluation. 1. Graph partitioning," *Operations Research* **37** (1989), 865–892.

[21] Johnson, D.S., Aragon, C.R., McGeoch, L.A., and Schevon, C., "Optimization by simulated annealing: an experimental evaluation. 2. Graph coloring and number partitioning," *Operations Research* **39** (1991), 378–406.

[22] Johnson, D.S., and McGeoch, C.C., eds. *Network Flows and Matching: First DIMACS Implementation Challenge*, *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* **12**, 1993.

[23] Johnson, D.S., and Trick, M., eds. *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*, *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* **26**, to appear.

[24] Jones, D.W., "An empirical comparison of priority queues and event-set implementations," *Commun. ACM* **29** (1986), 300–311.

[25] Knuth, D.E. *The Art of Computer Programming*, Volumes I (3rd ed.), II (3rd ed.), and III (2nd ed.). Addison-Wesley, Mass., 1997, 1997, and 1998.

[26] Knuth, D.E. *The Stanford GraphBase: A Platform for Combinatorial Computing.* Addison-Wesley, Reading Mass., 1993 (p. 460).

[27] Ladner, R., Fix, J.D., and LaMarca, A., "The cache performance of traversals and random accesses," *Proc. 10th ACM/SIAM Symp. on Discrete Algs. SODA99*, SIAM Press (1999), 613–622.

[28] LaMarca, A., and Ladner, R., "The influence of caches on the performance of heaps," *ACM J. Exp. Algorithmics* **1**, 4 (1996), `www.jea.acm.org/1996/LaMarcaInfluence`.

[29] LaMarca, A., and Ladner, R., "The influence of caches on the performance of sorting," *Proc. 8th ACM/SIAM Symp. on Discrete Algs. SODA97*, SIAM Press (1997), 370–379.

[30] McGeoch, C.C., "Analysis of algorithms by simulation: variance reduction techniques and simulation speedups," *ACM Comput. Surveys* **24** (1992), 195–212.

[31] McGeoch, C.C., "A bibliography of algorithm experimentation," this volume.

[32] Mehlhorn, K., private communication (1998).

[33] Mehlhorn, K., and Näher, S., "LEDA, a platform for combinatorial and geometric computing," *Commun. ACM* **38** (1995), 96–102.

[34] Melhorn, K., and Näher, S. *The LEDA Platform of Combinatorial and Geometric Computing.* Cambridge U. Press, 1999.

[35] Moret, B.M.E., Bader, D.A., Warnow, T., Wyman, S.K., and Yan, M., "A detailed study of breakpoint analysis," *Proc. 6th Pacific Symp. Biocomputing PSB 2001*, Hawaii, World Scientific Pub. (2001), 583–594.

[36] Moret, B.M.E., and H.D. Shapiro. *Algorithms from P to NP, Volume I: Design and Efficiency.* Benjamin-Cummings Publishing Co., Menlo Park, CA, 1991.

[37] Moret, B.M.E., and Shapiro, H.D., "On minimizing a set of tests," *SIAM J. Scientific & Statistical Comput.* **6** (1985), 983–1003.

[38] Moret, B.M.E., and Shapiro, H.D., "An empirical assessment of algorithms for constructing a minimal spanning tree," in *Computational Support for Discrete Mathematics*, N. Dean and G. Shannon, eds., *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* **15** (1994), 99–117.

[39] Morgenstern, C., and Shapiro, H.D., "Heuristics for rapidly four-coloring large planar graphs," *Algorithmica* **6** (1991), 869–891.

[40] Rahman, N., and Raman, R., "Analysing cache effects in distribution sorting," *Proc. 3rd Workshop on Algorithm Eng. WAE99*, in LNCS **1668**, Springer Verlag (1999), 183–197.

[41] Reinelt, G. *The Traveling Salesman: Computational Solutions for TSP Applications. Lecture Notes in Computer Science* **840** (1994), Springer Verlag, Berlin.

[42] Robertson, N., and Seymour, P., "Graph minors—a survey," in *Surveys in Combinatorics*, J. Anderson, ed., Cambridge U. Press, Cambridge, UK (1985), 153–171.

[43] Sanders, P., "Fast priority queues for cached memory," *ACM J. Exp. Algorithmics* **5**, 7 (2000), `www.jea.acm.org/2000/SandersPriority/`.

[44] Sedgewick, R., "Implementing Quicksort programs," *Comm. ACM* **21**, 10 (1978), 847-857.

[45] St. John, K., Warnow, T., Moret, B.M.E., and Vawter, L., "Performance study of phylogenetic methods: (unweighted) quartet methods and neighbor-joining," *Proc. 12th Ann. Symp. Discrete Algs. SODA 01*, SIAM Press (2001), 196–205.

[46] Stasko, J.T., and Vitter, J.S., "Pairing heaps: experiments and analysis," *Commun. ACM* **30** (1987), 234–249.
[47] Tarjan, R.E., "Efficiency of a good but not linear set union algorithm," *J. ACM* **22**, 2 (1975), 215–225.
[48] Vitter, J.S., "External memory algorithms and data structures: dealing with massive data," to appear in *ACM Comput. Surveys*, available at `www.cs.duke.edu/~jsv/Papers/Vit.IO_survey.ps.gz`.
[49] Xiao, L., Zhang, X., and Kubricht, S., "Improving memory performance of sorting algorithms," *ACM J. Exp. Algorithmics* **5**, 3 (2000), `www.jea.acm.org/2000/XiaoMemory/`.

Department of Computer Science, University of New Mexico, Albuquerque, NM 87131-1836
E-mail address: `moret@cs.unm.edu`