Algorithms and Experiments:
The New (and Old) Methodology

Bernard M.E. Moret
Department of Computer Science
University of New Mexico
Albuquerque, NM 87131, USA
moret@cs.unm.edu

Henry D. Shapiro
Department of Computer Science
University of New Mexico
Albuquerque, NM 87131, USA
shapiro@cs.unm.edu

Abstract: The last twenty years have seen enormous progress in the design of algorithms, but little of it has been put into practice. Because many recently developed algorithms are hard to characterize theoretically and have large running-time coefficients, the gap between theory and practice has widened over these years. Experimentation is indispensable in the assessment of heuristics for hard problems, in the characterization of asymptotic behavior of complex algorithms, and in the comparison of competing designs for tractable problems.

Implementation, although perhaps not rigorous experimentation, was characteristic of early work in algorithms and data structures. Donald Knuth has throughout insisted on testing every algorithm and conducting analyses that can predict behavior on actual data; more recently, Jon Bentley has vividly illustrated the difficulty of implementation and the value of testing. Numerical analysts have long understood the need for standardized test suites to ensure robustness, precision and efficiency of numerical libraries. It is only recently, however, that the algorithms community has shown signs of returning to implementation and testing as an integral part of algorithm development. The emerging disciplines of experimental algorithmics and algorithm engineering have revived and are extending many of the approaches used by computing pioneers such as Floyd and Knuth and are placing on a formal basis many of Bentley's observations.

We reflect on these issues, looking back at the last thirty years of algorithm development and forward to new challenges: designing cache-aware algorithms, algorithms for mixed models of computation, algorithms for external memory, and algorithms for scientific research.
Key Words: Algorithm engineering, cache-aware algorithms, efficiency, experimental algorithmics, external memory algorithms, implementation, methodology,

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) [Knu98], 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 and the "bounce" techniques for binary heaps, see, e.g. [MS91a]) as for more theoretical contributions.

Throughout the last 25 years, Jon Bentley has demonstrated the value of implementation and testing of algorithms, beginning with his text on writing efficient programs [Ben82] and continuing with his invaluable *Programming Pearls* columns in *Communications of the ACM*, now collected in a new volume [Ben99], and his *Software Explorations* columns in the *UNIX Review*. David Johnson, whose own work on optimization for NP-hard problems involves extensive experimentation, 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.

2 Empiricism in Algorithm Design

Natural scientists have perfected since at least the Middle Ages a particular form of enquiry, which has come to be called the scientific method. It is founded upon experiments; in its most basic form, it consists of using accumulated data to formulate a conjecture within a particular model, then conducting additional experiments to affirm or refute the conjecture. Such a completely empirical approach is well suited for a natural science, where the final arbiter is nature as revealed to us through experiments and measurements, but it is incomplete in the artificial and mathematically precise world of computing, where the behavior of an algorithm or data structure can often, 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. Thus computer scientists must both learn from the natural sciences, where experimentation has been used for centuries and where the scientific method has been developed to optimize the use of experiments, but must also remain 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.

3 Asymptotic Analysis vs. Implementation

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 conceivable application. A good 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 $O(|E|\log|V|)$ when implemented with binary heaps, but experimentation [MS94] verifies that the crossover point occurs for dense graphs with well over a million vertices and thus hundreds of millions of 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 [RS85]) gave a cubic-time algorithm to determine whether a given graph is a minor of another, but the proportionality constants are gigantic—a recent estimate was on the order of 10^{150} [Fel99]—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 [AMO93]. (Indeed, in some of its newer versions, its running time is competitive with that of the modern interior-point methods [BFG$^+$00].)
- 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.

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 ten years rely on several layers of complex, unimplemented algorithms and data structures. In order to implement one of these recent algorithms, a programmer would face the daunting prospect of developing implementations for all preceding layers. Moreover, the "paper-and-pencil" algorithms often ignore issues critical in making implementations efficient, such as low-level algorithmic issues and architecture-dependent issues (particularly caching), as well as issues of robustness (such as the potential effects of numerical errors or unexpected symmetries in geometric computations, although recent recent conferences in computational geometry have featured a number of papers addressing these issues). 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 [MWB$^+$01] as well as yielding robust libraries of data structures with minimal overhead, as done in the LEDA library [MN95, MN99].

There is no reason to abandon asymptotic worst-case analysis; 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 designer described them at a very high level. The practitioner is not the only one who stands to benefit from implementation: the detailed, step-by-step understanding required for implementation may enable the designer to notice features that had remained invisible in the high-level design and so to bring about a simplified or improved design.

4 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 algorithms (heuristics, approximations, or exact solvers) for the solution of NP-hard problems.
- Comparing the actual performance of competing algorithms for tractable problems and characterizing the effects of algorithm engineering.
- Investigating and refining models and optimization criteria—what should be optimized? and what parameters matter?

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. 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.

4.1 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. Scattered articles during the 70s (see, e.g., [DS85]) kept a low level of experimentation active, but did not attempt to provide methodological pointers. More recent and comprehensive work began with Bentley's many contributions in his *Programming Pearls* (starting in 1983 [Ben83]), then with Jones' comparison of data structures for priority queues [Jon86] and Stasko and Vitter's combination of analytical and experimental work in the study of pairing heaps [SV87]. An early experimental study on a large scale was that of Moret and Shapiro on sorting algorithms [MS91a] (Chapter 8), itself inspired by the work of

Knuth in his Volume III [Knu98], followed by that of the same authors on algorithms for constructing minimum spanning trees [MS94]. In 1991, David Johnson and others initiated the very successful DIMACS Computational Challenges, the first of which [JM93] focused on network flow and shortest path algorithms, indirectly giving rise to several modern, thorough studies, by Cherkassky *et al.* on shortest paths [CGR96], by Cherkassky *et al.* on the implementation of the push-relabel method for matching and network flows [CGM$^+$98, CG97], and by Goldberg and Tsioutsiouliklis on cut trees [GT01]. 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.

Much interest has focused over the last three to four years on the question of tailoring algorithms and implementations to the cache structure and policies of the architecture. Caching effects can significantly alter the predictions of asymptotic analysis; a classic example is hashing: most textbook on data structures still advocate using double hashing in preference to linear probing, whereas experimental data clearly indicates that linear probing is the faster method, thanks to its good locality (see [BMQ98]). Pioneering studies by Ladner and his coworkers [LL96, LL97] established that cache optimization was feasible, algorithmically interesting, and worthwhile, even for such old friends as sorting algorithms [ACVW01, LL97, RR99, XZK00] and priority queues [LL96, San00]; indeed, even matrix multiplication, which has been optimized in numerical libraries for over 40 years (including optimizations for paging behavior), is amenable to such techniques [ERS90]. 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 [MWB$^+$01]. The related, and much better studied, model of out-of-core computing, as pioneered by Vitter and his coworkers [Vit01], has inspired new work in cache-aware and cache-independent algorithm design. Once again, though, this trend was pioneered over 40 years ago, when programmers had to work with very limited memory and studied detailed optimization strategies for accessing early secondary-storage devices such as magnetic drums, and followed in the seventies by much work on out-of-core computing—Knuth has detailed analyses of external sorting algorithms in his Volume III [Knu98]. Today, we are confronted with much deeper memory hierarchies and enormous volumes of data, so we need to return to these optimization techniques and extend them to apply throughout the hierarchy.

Characterizing the behavior of algoritms 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 [FM97] 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.

4.2 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 have guaranteed performance bounds. 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 [AMO93], Chapter 18). Inspired in part by experimental studies of integer-programming algorithms for combinatorial optimization, such as algorithms for the set-covering problem—see, e.g., [BH80], we conducted a large-scale combinatorial study on the minimum test set problem [MS85], one of the first such studies in Computer Science to include both real-world and generated instances. 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 [JAMS89, JAMS91], which, among other things, demonstrated the value of a varied collection of test instances. The Second DIMACS Computational Challenge [JT96] 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.* [CaDRKLS97]. The Traveling Salesperson problem has seen large numbers of experimental studies (including the well publicized study of Jon Bentley [Ben90]), made possible in part by the development of a library of test cases [Rei94]. 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 [JAMS89] discussed many facets of the problem, while Morgenstern and Shapiro [MS91b] provided a detailed study of algorithms to color planar graphs.

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. 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-world data. 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.

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; the setup is essentially arbitrary—it should not be allowed to limit one's creativity. We 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.
- 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; in our study of sorting algorithms, we observed a nonlinear running time for radix sort (contradicting the theoretical analysis), which is a simple consequence of caching effects. Recent studies by LaMarca and Ladner [LL96, LL97] have quantified many aspects of caching and offer suggestions on how to work around (or take advantage of) caching effects.

Johnson [Joh01] offers a 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 [Knu98], 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? 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 alone. 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 [Knu93] 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 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 [McG92] 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 [FM97]), is crucial; any serious discrepancy needs to be investigated. 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.

8 Conclusions

Implementation and 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 a deeper understanding of existing algorithms and thus lead to new conjectures and new 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;
– 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.

Acknowledgments

Bernard Moret's work was supported in part by the National Science Foundation under grant ITR 00-81404.

References

[ACVW01] L. Arge, J. Chase, J. S. Vitter, and R. Wickremesinghe, *Efficient sorting using registers and caches*, Proc. 4th Workshop on Algorithm Eng. WAE 2000, Springer Verlag, 2001, to appear in LNCS.

[AMO93] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network flows*, Prentice Hall, Englewood Cliffs, NJ, 1993.

[Ben82] J. L. Bentley, *Writing efficient programs*, Prentice-Hall, Englewood Cliffs, NJ, 1982.

[Ben83] ———, *Programming pearls: cracking the oyster*, Commun. ACM **26** (1983), no. 8, 549–552.

[Ben90] ———, *Experiments on geometric traveling salesman heuristics*, Report CS TR 151, AT&T Bell Laboratories, 1990.

[Ben99] ———, *Programming pearls*, ACM Press, New York, 1999.

[BFG+00] R. E. Bixby, M. Fenelon, Z. Gu, E. Rothberg, and R. Wunderling, *MIP: Theory and practice—closing the gap*, System Modelling and Optimization: Methods, Theory and Applications (M. J. D. Poweel and S. Scholtes, eds.), Kluwer Acad. Pub., 2000, pp. 19–49.

[BH80] E. Balas and A. Ho, *Set covering algorithms using cutting planes, heuristics, and subgradient optimization: A computational study*, Math. Progr. **12** (1980), 37–60.

[BMQ98] J. R. Black, C. U. Martel, and H. Qi, *Graph and hashing algorithms for modern architectures: design and performance*, Proc. 2nd Workshop on Algorithm Eng. WAE 98, Max-Planck Inst. für Informatik, 1998, in TR MPI-I-98-1-019, pp. 37–48.

[CaDRKLS97] C. S. Chekuri, A. V. Goldberg adn D. R. Karger, M. S. Levine, and C. Stein, *Experimental study of minimum cut algorithms*, Proc. 8th ACM/SIAM Symp. on Discrete Algs. SODA 97, SIAM Press, 1997, pp. 324–333.

[CG97] B. V. Cherkassky and A. V. Goldberg, *On implementing the push-relabel method for the maximum flow problem*, Algorithmica **19** (1997), 390–410.

[CGM+98] B. V. Cherkassky, A. V. Goldberg, P. Martin, J. C. Setubal, and J. Stolfi, *Augment or push: a computational study of bipartite matching and unit-capacity flow algorithms*, ACM J. Exp. Algorithmics **3** (1998), no. 8, `www.jea.acm.org/1998/CherkasskyAugment/`.

[CGR96] B. V. Cherkassky, A. V. Goldberg, and T. Radzik, *Shortest paths algorithms: theory and experimental evaluation*, Math. Progr. **73** (1996), 129–174.

[DS85] S. P. Dandamudi and P. G. Sorenson, *An empirical performance comparison of some variations of the k-d tree and bd tree*, Int'l J. Computer and Inf. Sciences **14** (1985), no. 3, 134–158.

[ERS90] N. Eiron, M. Rodeh, and I. Stewarts, *Matrix multiplication: a case study of enhanced data cache utilization*, ACM J. Exp. Algorithmics **4** (1990), no. 3, `www.jea.acm.org/1999/EironMatrix/`.

[Fel99] M. Fellows, 1999, private communication.

[FM97] U. Finkler and K. Mehlhorn, *Runtime prediction of real programs on real machines*, Proc. 8th ACM/SIAM Symp. on Discrete Algs. SODA 97, SIAM Press, 1997, pp. 380–389.

[GT01] A. V. Goldberg and K. Tsioutsiouliklis, *Cut tree algorithms: an experimental study*, J. Algs. **38** (2001), no. 1, 51–83.

[JAMS89] D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon, *Optimization by simulated annealing: an experimental evaluation. 1. graph partitioning*, Operations Research **37** (1989), 865–892.

[JAMS91] ———, *Optimization by simulated annealing: an experimental evaluation. 2. graph coloring and number partitioning*, Operations Research **39** (1991), 378–406.

[JM93] D. S. Johnson and C. C. McGeoch (eds.), *Network flows and matching: First DIMACS implementation challenge*, vol. 12, Amer. Math. Soc., 1993.

[Joh01] D. S. Johnson, *A theoretician's guide to the experimental analysis of algorithms*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Amer. Math. Soc., 2001, to appear.

[Jon86] D. W. Jones, *An empirical comparison of priority queues and event-set implementations*, Commun. ACM **29** (1986), 300–311.

[JT96] D. S. Johnson and M. Trick, *Cliques, coloring, and satisfiability: Second DIMACS implementation challenge*, vol. 26, Amer. Math. Soc., 1996.

[Knu93] D. E. Knuth, *The Stanford GraphBase: A platform for combinatorial computing*, Addison-Wesley, Reading, Mass., 1993.

[Knu98] ———, *The art of computer programming, vols I (3rd ed.), II (3rd ed.), and III (2nd ed.)*, Addison-Wesley, Reading, Mass., 1997, 1997, and 1998.

[LL96] A. LaMarca and R. Ladner, *The influence of caches on the performance of heaps*, ACM J. Exp. Algorithmics **1** (1996), www.jea.acm.org/1996/LaMarcaInfluence/.

[LL97] ———, *The influence of caches on the performance of sorting*, Proc. 8th ACM/SIAM Symp. on Discrete Algs. SODA 97, SIAM Press, 1997, pp. 370–379.

[McG92] C. C. McGeoch, *Analysis of algorithms by simulation: variance reduction techniques and simulation speedups*, ACM Comput. Surveys **24** (1992), 195–212.

[MN95] K. Mehlhorn and S. Näher, *LEDA, a platform for combinatorial and geometric computing*, Commun. ACM **38** (1995), 96–102.

[MN99] K. Melhorn and S. Näher, *The LEDA platform of combinatorial and geometric computing*, Cambridge U. Press, Cambridge, UK, 1999.

[MS85] B. M. E. Moret and H. D. Shapiro, *On minimizing a set of tests*, SIAM J. Scientific & Statistical Comput. **6** (1985), 983–1003.

[MS91a] ———, *Algorithms from P to NP, volume I: Design and efficiency*, Benjamin-Cummings Publishing Co., Menlo Park, CA, 1991.

[MS91b] C. Morgenstern and H. D. Shapiro, *Heuristics for rapidly four-coloring large planar graphs*, Algorithmica **6** (1991), 869–891.

[MS94] B. M. E. Moret and H. D. Shapiro, *An empirical assessment of algorithms for constructing a minimal spanning tree*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science (N. Dean and G. Shannon, eds.), vol. 15, Amer. Math. Soc., 1994, pp. 99–117.

[MWB$^+$01] B. M. E. Moret, S. K. Wyman, D. A. Bader, T. Warnow, and M. Yan, *A new implementation and detailed study of breakpoint analysis*, Proc. 6th Pacific Symp. Biocomputing PSB 2001, World Scientific Pub., 2001, pp. 583–594.

[Rei94] G. Reinelt, *The traveling salesman: Computational solutions for tsp applications*, Springer Verlag, Berlin, 1994, in LNCS 840.

[RR99] N. Rahman and R. Raman, *Analysing cache effects in distribution sorting*, Proc. 3rd Workshop on Algorithm Eng. WAE 99 (Berlin), Springer Verlag, 1999, in LNCS 1668, pp. 183–197.

[RS85] N. Robertson and P. Seymour, *Graph minors—a surveys*, Surveys in Combinatorics (J. Anderson, ed.), Cambridge U. Press, Cambridge, UK, 1985, pp. 153–171.

[San00] P. Sanders, *Fast priority queues for cached memory*, ACM J. Exp. Algorithmics **5** (2000), no. 7, www.jea.acm.org/2000/SandersPriority/.

[SV87] J. T. Stasko and J. S. Vitter, *Pairing heaps: experiments and analysis*, Commun. ACM **30** (1987), 234–249.

[Vit01] J. S. Vitter, *External memory algorithms and data structures: dealing with massive data*, ACM Comput. Surveys (2001), to appear, available at www.cs.duke.edu/ jsv/Papers/Vit.IO_survey.ps.gz.

[XZK00] L. Xiao, X. Zhang, and S. Kubricht, *Improving memory performance of sorting algorithms*, ACM J. Exp. Algorithmics **5** (2000), no. 3, www.jea.acm.org/2000/XiaoMemory/.