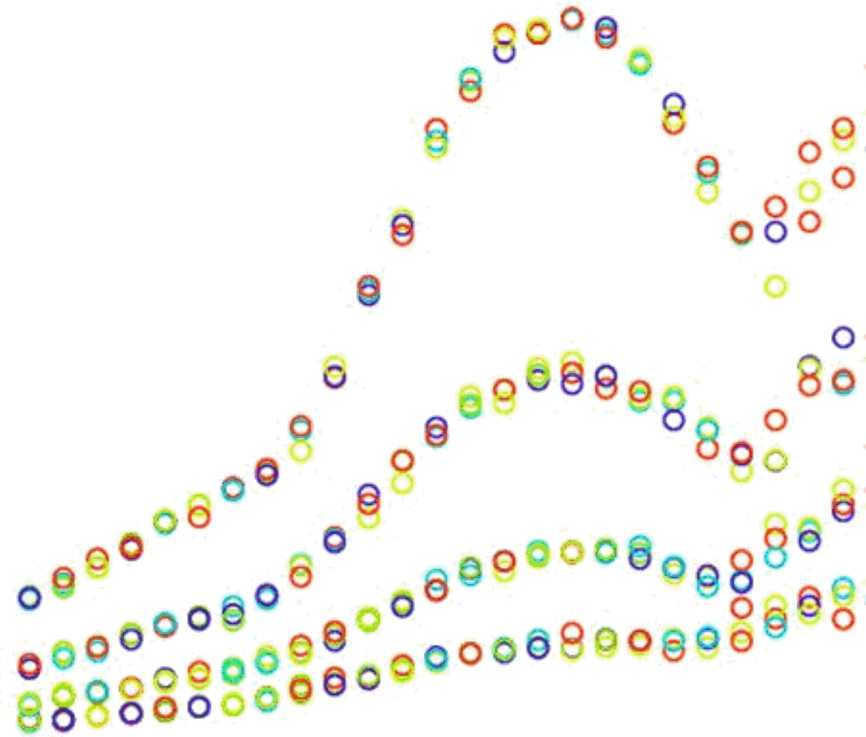
catherine c. mcgeoch



a guide to experimental algorithmics

CAMBRIDGE

A Guide to Experimental Algorithmics

CATHERINE C. MCGEOCH
Amherst College

CAMBRIDGE UNIVERSITY PRESS
Cambridge, New York, Melbourne, Madrid, Cape Town,
Singapore, São Paulo, Delhi, Mexico City

Cambridge University Press
32 Avenue of the Americas, New York, NY 10013-2473, USA

www.cambridge.org
Information on this title: www.cambridge.org/9780521173018

First published 2012

Printed in the United States of America

A catalog record for this publication is available from the British Library.

ISBN 978-1-107-00173-2 Hardback
ISBN 978-0-521-17301-8 Paperback

Additional resources for this publication at www.cs.amherst.edu/alglab

Contents

Preface

This guidebook is written for anyone – student, researcher, or practitioner – who wants to carry out computational experiments on algorithms (and programs) that yield correct, general, informative, and useful results. (We take the wide view and use the term "algorithm" to mean "algorithm or program" from here on.)

Whether the goal is to predict algorithm performance or to build faster and better algorithms, the experiment-driven methodology outlined in these chapters provides insights into performance that cannot be obtained by purely abstract means or by simple runtime measurements. The past few decades have seen considerable developments in this approach to algorithm design and analysis, both in terms of number of participants and in methodological sophistication.

In this book I have tried to present a snapshot of the state-of-the-art in this field (which is known as *experimental algorithmics* and *empirical algorithmics*), at a level suitable for the newcomer to computational experiments. The book is aimed at a reader with some undergraduate computer science experience: you should know how to program, and ideally you have had at least one course in data structures and algorithm analysis. Otherwise, no previous experience is assumed regarding the other topics addressed here, which range widely from architectures and operating systems, to probability theory, to techniques of statistics and data analysis

A note to academics: The book takes a nuts-and-bolts approach that would be suitable as a main or supplementary text in a seminar-style course on advanced algorithms, experimental algorithmics, algorithm engineering, or experimental methods in computer science. Several case studies are presented throughout; a companion website called *AlgLab – Open Laboratory for Experiments on Algorithms* makes the files, programs, and tools described in the case studies available for downloading. Suggestions for experimental problems and projects appear at the end of each chapter.

This book wouldn't exist without the "number of participants" alluded to earlier, members of the research community who have worked to develop this new methodology while contributing a huge body of experiment-based research on design and analysis of algorithms, data structures, heuristics, and models of computation. I am grateful for all those collegial conversations during break-out sessions, carried out over countless cups of coffee: thanks to David Bader, Giuseppe Italiano, David S. Johnson, Richard Ladner, Peter Sanders, Matt Stallmann, and Cliff Stein. A huge thank you, especially, to Jon Bentley, whose comments, story ideas, and criticisms of draft versions of this book were immensely valuable. My editor Lauren Cowles also did a magnificent job of helping me to untangle knots in the draft manuscript.

Possibly more important to the final product than colleagues and readers are the family and friends who remind me that life is more than an endless bookwriting process: to Alex and Ian, Ruth and Stephen, Susan Landau, and Maia Ginsburg, thank you for keeping me sane.

And finally, very special thanks to the guy who fits all of the above categories and more: colleague, technical adviser, reader, supporter, husband, and friend. Thank you Lyle.

Catherine C. McGeoch
Amherst, Massachusetts
July 2011

1

Introduction

The purpose of computing is insight, not numbers.
Richard Hamming, *Numerical Methods for Scientists and Engineers*

Some questions:

- You are a working programmer given a week to reimplement a data structure that supports client transactions, so that it runs efficiently when scaled up to a much larger client base. Where do you start?
- You are an algorithm engineer, building a code repository to hold fast implementations of dynamic multigraphs. You read papers describing asymptotic bounds for several approaches. Which ones do you implement?
- You are an operations research consultant, hired to solve a highly constrained facility location problem. You could build the solver from scratch or buy optimization software and tune it for the application. How do you decide?
- You are a Ph.D. student who just discovered a new approximation algorithm for graph coloring that will make your career. But you're stuck on the average-case analysis. Is the theorem true? If so, how can you prove it?
- You are the adviser to that Ph.D. student, and you are skeptical that the new algorithm can compete with state-of-the-art graph coloring algorithms. How do you find out?

One good way to answer all these questions is: *run experiments to gain insight.*

This book is about *experimental algorithmics*, which is the study of algorithms and their performance by experimental means. We interpret the word *algorithm* very broadly, to include algorithms and data structures, as well as their implementations in source code and machine code. The two main challenges in algorithm studies addressed here are:

- *Analysis*, which aims to predict performance under given assumptions about inputs and machines. Performance may be a measure of time, solution quality, space usage, or some other metric.
- *Design*, which is concerned with building faster and better algorithms (and programs) to solve computational problems.

Very often these two activities alternate in an algorithmic research project – a new design strategy requires analysis, which in turn suggests new design improvements, and so forth.

A third important area of algorithm studies is *models of computation*, which considers how changes in the underlying machine (or machine model) affect design and analysis. Problems in this area are also considered in a few sections of the text.

The discussion is aimed at the newcomer to experiments who has some familiarity with algorithm design and analysis, at about the level of an undergraduate course. The presentation draws on knowledge from diverse areas, including theoretical algorithmics, code tuning, computer architectures, memory hierarchies, and topics in statistics and data analysis. Since "everybody is ignorant, only on different subjects" (Will Rogers), basic concepts and definitions in these areas are introduced as needed.

1.1 Why Do Experiments?

The foundational work in algorithm design and analysis has been carried out using a *theoretical* approach, which is based on abstraction, theorem, and proof. In this framework, algorithm design means creating an algorithm in pseudocode, and algorithm analysis means finding an asymptotic bound on the dominant operation under a worst-case or average-case model.

The main benefit of this abstract approach is universality of results – no matter how skilled the programmer, or how fast the platform, the asymptotic bound on performance is guaranteed to hold. Furthermore, the asymptotic bound is the most important property determining performance at large n, which is exactly when performance matters most. Here are two stories to illustrate this point.

- Jon Bentley [7] ran a race between two algorithms to solve the maximum-sum subarray problem. The $\Theta(n^3)$ algorithm was implemented in the fastest environment he could find (tuned C code on a 533MHz Alpha 21164), and the $\Theta(n)$ algorithm ran in the slowest environment available (interpreted Basic on a 2.03MHz Radio Shack TRS-80 Model II). Despite these extreme platform differences, the crossover point where the fast asymptotic algorithm started beating the

Here are a few examples showing how experiments have played a central role in both algorithm design and algorithm engineering.

- The 2006 *9th DIMACS Implementation Challenge–Shortest Paths* workshop contained presentations of several projects to speed up single-pair shortest-path (SPSP) algorithms. In one paper from the workshop, Sanders and Shultes [24] describe experiments to engineer an algorithm to run on roadmap graphs used in global positioning system (GPS) Routing applications: the Western Europe and the United States maps contain ($n = 18$ million, $m = 42.5$ million) and ($n = 23.9$ million, $m = 58.3$ million) nodes and edges, respectively. They estimate that their tuned implementation of Dijkstra's algorithm runs more than a million times faster on an average query than the best known implementation for general graphs.
- Bader et al. [2] describe efforts to speed up algorithms for computing optimal phylogenies, a problem in computational biology. The breakpoint phylogeny heuristic uses an exhaustive search approach to generate and evaluate candidate solutions. Exact evaluation of *each* candidate requires a solution to the traveling salesman problem, so that the worst-case cost is $O(2n!!)$ [*sic*–double factorial] to solve a problem with n genomes. Their engineering efforts, which exploited parallel processing as well as algorithm and code tuning, led to speedups by factors as large as 1 million on problems containing 10 to 12 genomes.
- Speedups by much smaller factors than a million can of course be critically important on frequently used code. Yaroslavskiy et al. [27] describe a project to implement the Arrays.sort() method for JDK 7, to achieve fast performance when many duplicate array elements are present. (Duplicate array elements represent a worst-case scenario for many implementations of quicksort.) Their tests of variations on quicksort yielded performance differences ranging from 20 percent faster than a standard implementation (on arrays with no duplicates), to more than 15 times faster (on arrays containing identical elements).
- Sometimes the engineering challenge is simply to demonstrate a working implementation of a complex algorithm. Navarro [21] describes an effort to implement the LZ-Index, a data structure that supports indexing and fast lookup in compressed data. Navarro shows how experiments were used to guide choices made in the implementation process and to compare the finished product to competing strategies. This project is continued in [11], which describes several tuned implementations assembled in a repository that is available for public use.

These examples illustrate the ways in which experiments have played key roles in developing new insights about algorithm design and analysis. Many more examples can be found throughout this text and in references cited in the Chapter Notes.

1.2 Key Concepts

This section introduces some basic concepts that provide a framework for the larger discussion throughout the book.

A Scale of Instantiation

We make no qualitative distinction here between "algorithms" and "programs." Rather, we consider algorithms and programs to represent two points on a *scale of instantiation*, according to how much specificity is in their descriptions. Here are some more recognizable points on this scale.

- At the most abstract end are *metaheuristics* and *algorithm paradigms*, which describe generic algorithmic structures that are not tied to particular problem domains. For example, Dijkstra's algorithm is a member of the greedy paradigm, and tabu search is a metaheuristic that can be applied to many problems.
- The *algorithm* is an abstract description of a process for solving an abstract problem. At this level we might see Dijkstra's algorithm written in pseudocode. The pseudocode description may be more or less instantiated according to how much detail is given about data structure implementation.
- The *source program* is a version of the algorithm implemented in a particular high-level language. Specificity is introduced by language and coding style, but the source code remains platform-independent. Here we might see Dijkstra's algorithm implemented in C++ using an STL priority queue.
- The *object code* is the result of compiling a source program. This version of the algorithm is written in machine code and specific to a family of architectures.
- The *process* is a program actively running on a particular machine at a particular moment in time. Performance at this level may be affected by properties such as system load, the size and shape of the memory hierarchy, and process scheduler policy.

Interesting algorithmic experiments can take place at any point on the instantiation scale. We make a conceptual distinction between the *experimental subject*, which is instantiated somewhere on the scale, and the *test program*, which is implemented to study the performance of the subject.

For example, what does it mean to measure an algorithm's time performance? Time performance could be defined as a count of the dominant cost, as identified by theory: this is an abstract property that is universal across programming languages, programmers, and platforms. It could be a count of instruction executions, which is a property of object code. Or it could be a measurement of elapsed time, which depends on the code as well as on the platform. There is one test program, but the experimenter can choose to measure any of these properties, according to the level of instantiation adopted in the experiment.

In many cases the test program may be exactly the subject of interest – but it need not be. By separating the two roles that a program may play, both as test subject and as testing apparatus, we gain clarity about experimental goals and procedures. Sometimes this conceptual separation leads to better experiments, in the sense that a test program can generate better-quality data more efficiently than a conventional implementation could produce (see Chapter 6 for details).

This observation prompts the first of many guidelines presented throughout the book. Guidelines are meant to serve as short reminders about best practice in experimental methodology. A list of guidelines appears in the Chapter Notes at the end of each chapter.

Guideline 1.1 *The "algorithm" and the "program" are just two points on a scale between abstract and instantiated representations of a given computational process.*

The Algorithm Design Hierarchy

Figure 1.1 shows the *algorithm design hierarchy*, which comprises six levels that represent broad strategies for improving algorithm performance. This hierarchical approach to algorithm design was first articulated by Reddy and Newell [23] and further developed by Bentley [6], [7]. The list in Figure 1.1 generally follows Bentley's development, except two layers–algorithm design and code tuning – are now split into three – algorithm design, algorithm tuning, and code tuning. The distinction is explained further in Chapter 4.

The levels in this hierarchy are organized roughly in the order in which decisions must be made in an algorithm engineering project. You have to design the algorithm before you implement it, and you cannot tune code before the implementation exists. On the other hand, algorithm engineering is not really a linear process – a new insight, or a roadblock, may be discovered at any level that makes it necessary to start over at a higher level.

Chapter 4 surveys tuning strategies that lie at the middle two levels of this hierarchy – algorithm tuning and code tuning. Although concerns at the other levels are outside the scope of this book, do not make the mistake of assuming that they are not important to performance. The stories in Section 1.1 about Bentley's race and Skiena's pyramid numbers show how important it is to get the asymptotics right in the first place.

In fact, the greatest feats of algorithm engineering result from combining design strategies from different levels: a 10-fold speedup from rearranging file structures at the system level, a 100-fold speedup from algorithm tuning, a 5-fold speedup from code tuning, and a 2-fold improvement from using an optimizing compiler, will combine *multiplicatively* to produce a 10,000-fold improvement in overall running time. Here are two stories that illustrate this effect.

- **System structure.** Decompose the software into modules that interact efficiently. Check whether the target runtime environment provides sufficient support for the modules. Decide whether the final product will run on a concurrent or sequential platform.
- **Algorithm and data structure design.** Specify the exact problem that is to be solved in each module. Choose appropriate problem representations. Select or design algorithms and data structures that are asymptotically efficient.
- **Implementation and algorithm tuning.** Implement the algorithm, or perhaps build a family of implementations. Tune the algorithm by considering high-level structures relating to the algorithm paradigm, input classes, and cost models.
- **Code tuning.** Consider low-level code-specific properties such as loops and procedure calls. Apply a systematic process to transform the program into a functionally equivalent program that runs faster.
- **System software.** Tune the runtime environment for best performance, for example by turning on compiler optimizers and adjusting memory allocations.
- **Platform and hardware.** Shift to a faster CPU and/or add coprocessors.

Figure 1.1. The algorithm design hierarchy. The levels in this hierarchy represent broad strategies for speeding up algorithms and programs.

Cracking RSA-129. Perhaps the most impressive algorithm engineering achievement on record is Atkins et al.'s [1] implementation of a program to factor a 129-digit number and solve an early RSA Encryption Challenge. Reasonable estimates at the time of the challenge were that the computation would take 4 quadrillion years. Instead, 17 years after the challenge was announced, the code was cracked in an eight-month computation: this represents a 6 quadrillion–fold speedup over the estimated computation time.

The authors' description of their algorithm design process gives the following insights about contributions at various levels of the algorithm design hierarchy.

- The task was carried out in three phases: an eight-month distributed computation (1600 platforms); then a 45-hour parallel computation (16,000 CPUs); then a few hours of computation on a sequential machine. Assuming optimal speedups due to concurrency, the first two phases would have required a total of 1149.2

years on a sequential machine. Thus *concurrency* contributed at most a 1150-fold speedup.

- Significant *system* design problems had to be solved before the computation could take place. For example, the distributed computation required task modules that could fit into main memory of all platforms offered by volunteers. Also, data compression was needed to overcome a critical memory shortage late in the computation.
- According to Moore's Law (which states that computer speeds typically double every 18 months), faster hardware alone could have contributed a 2000-fold speedup during the 17 years between challenge and solution. But in fact the original estimate took this effect into account. Thus no speedup over the estimate can be attributed to *hardware*.
- The authors describe *code tuning* improvements that contributed a factor of 2 speedup (there may be more that they did not report).
- Divide 6 quadrillion by $2300 = 1150 \times 2$: the remaining 2.6–trillion-fold speedup is due to improvements at the *algorithm design* level.

Finding Phylogenies Faster. In a similar vein, Bader et al. [2], [19] describe their engineering efforts to speed up the breakpoint phylogeny algorithm described briefly in Section 1.1.

- Since the generation of independent candidate solutions can easily be parallelized, the authors implemented their code for a 512-processor Alliance Cluster platform. This decision at the *systems* level to adopt a parallel solution produced an optimal 512-fold speedup over a comparable single-processor version.
- *Algorithm design* and *algorithm tuning* led to speedups by factors around 100; redesign of the data structures yielded another factor of 10. The cumulative speedup is 1000. The authors applied cache-aware tuning techniques to obtain a smaller memory footprint (from 60MB down to 1.8MB) and to improve cache locality. They remark that the new implementation runs almost entirely in cache for their test data sets.
- Using profiling to identify timing bottlenecks in a critical subroutine, they applied *code tuning* to obtain 6- to 10-fold speedups. The cumulative speedup from algorithm and code tuning was between 300 and 50,000, depending on inputs.

This combination of design improvements resulted in cumulative speedups by factors up to 1 million on some inputs.

Guideline 1.2 *When the code needs to be faster, consider all levels of the algorithm design hierarchy.*

and simulation speedups. These ideas are illustrated using two algorithms for the self-organizing sequential search problem.

- Finally, Chapter 7 surveys data analysis and statistical techniques that are relevant to common scenarios arising in algorithmic experiments. Most of the data sets used as illustrations in this section come from the case study experiments described in previous chapters.

The case studies mentioned here play a central role in the presentation of concepts strategies, and techniques. All of the solver programs and input generators described in these case studies are available for downloading from the *Algorithmics Laboratory* (AlgLab), which is a companion Web site to this text. Visit `www.cs.amherst.edu/alglab` to learn more.

The reader is invited to download these materials and try out the ideas in this guidebook, or to extend these examples by developing new experiments. Suggestions for additional experiments appear in the Problems and Projects section at the end of each chapter.

1.4 Chapter Notes

The Chapter Notes section at the end of each chapter collects guidelines and gives references to further reading on selected topics. Here are the guidelines from this chapter.

1.1 The "algorithm" and the "program" are just two points on a scale between abstract and instantiated representations of a given computational process.

1.2 When the code needs to be faster, consider all levels of the algorithm design hierarchy.

1.3 The experimental path is not straight, but cyclical: planning alternates with execution; experimental design alternates with tool building; analysis alternates with data collection.

Readings in Methodology

Here is a reading list of papers and books that address topics in experimental methodology for problems in algorithm design and analysis.

Articles

"Designing and reporting on computational experiments with heuristic methods," by R. S. Barr et al. Guidelines on experimental design and reporting standards, emphasizing heuristics and optimization problems [3].

"Ten Commandments for Experiments on Algorithms," by J. L. Bentley. The title speaks for itself. [5]

"Algorithm Engineering," by C. Demetrescu, I. Finocchi, and G. F. Italiano. A survey of issues and problems in algorithm engineering. [10]

How Not to Do It, by I. P. Gent et al., Pitfalls of algorithmic experiments and how to avoid them. [12].

"Testing heuristics: We have it all wrong," by J. Hooker. A critique of experimental methodology in operations research. [13]

"A theoretician's guide to the experimental analysis of algorithms," by D. S. Johnson. Pet peeves and pitfalls of conducting and reporting experimental research on algorithms, aimed at the theoretician. [14]

"Toward an experimental method in algorithm analysis," by C. C. McGeoch. Early discussion of some of the ideas developed in this book. [17]

"How to present a paper on experimental work with algorithms," by C. McGeoch and B. M. E. Moret. Guidelines for presenting a research talk, aimed at the academic. [18]

"Algorithm Engineering – an attempt at a definition using sorting as an example," by Peter Sanders. A description of the field, including issues and open questions. [25]

Books

1. *Experimental Methods for the Analysis of Optimization Algorithms*, T. Bartz-Beielstein, et al., eds. Broad coverage of topics in experimental methodology, especially statistics and data analysis, emphasizing problems in optimization. [4]
2. *Programming Pearls,* by J. L. Bentley. Written for the practicing programmer, the book addresses topics at the interface between theory and practice and contains many tips on how to perform experiments. [7]
3. *Empirical Methods for Artificial Intelligence,* by P. Cohen. A textbook on statistics and data analysis, with many illustrations from experiments on heuristic algorithms. [9]
4. *Algorithm Engineering: Bridging the Gap between Algorithm Theory and Practice,* M. Müller-Hanneman and S. Schirra, eds. A collection of articles addressing topics in engineering and experimentation, aimed at graduate students and research scientists. [20]

A timeline

The discipline of experimental algorithmics has come of age in recent years, due to the efforts of a growing community of researchers. Members of this group have worked to organize workshops and publication venues, launch repositories

and libraries for engineered products, and develop methodologies for this new approach to algorithm research.

Here is list of meetings and journals that provide publication venues for research in experimental algorithmics, in chronological order by date of launch. Consult these resources to find many examples of research contributions to algorithm design and analysis, as well as discussions of methodological issues.

1989 The *ORSA Journal on Computing* is launched to publish articles in the intersection of operations research and computer science. In 1996 the name of the sponsoring organization changed; the journal is now called the *INFORMS Journal on Computing*.

1990 The first ACM-SIAM Symposium on Data Structures and Algorithms (SODA) is organized by David Johnson. The call for papers explicitly invites "analytical or experimental" analyses, which may be "theoretical or based on real datasets."

1990 The first DIMACS Implementation Challenge is coorganized by David Johnson and Catherine McGeoch. The DIMACS Challenges are year-long, multiteam, cooperative research projects in experimental algorithmics.

1995 Inaugural issue of the ACM *Journal of Experimental Algorithmics*, Bernard Moret, editor in chief.

1997 The first Workshop on Algorithm Engineering (WAE) is organized by Giuseppe Italiano. In 2002 this workshop joins the European Symposium on Algorithms (ESA), as the "Engineering and Applications" track.

1999 The annual workshop on Algorithm Engineering and Experiments (ALENEX) is coorganized in 1999 by Mike Goodrich and Catherine McGeoch. It was inspired by the Workshop on Algorithms and Experiments (ALEX), organized in 1998 by Roberto Battiti.

2000 The first of several Dagstuhl Seminars on Experimental Algorithmics and Algorithm Engineering is organized by Rudolf Fleischer, Bernard Moret, and Erik Schmidt.

2001 The First International Workshop on Efficient Algorithms (WEA) is organized by Klaus Jansen and Evripidis Bampis. In 2003 it becomes the International Workshop on Experimental and Efficient Algorithms (WEA) coordinated by José Rolim. In 2009 it becomes the Symposium on Experimental Algorithms (SEA).

1.5 Problems and Projects

1. Find three experimental analysis papers from the publication venues described in the Chapter Notes. Where do the experiments fall on the scale of instantiation

described in Section 1.2. Why do you think the authors choose to focus on those instantiation points?

2. Read Hooker's [13] critique of current practice in experimental algorithmics and compare it to the three papers in the previous question. Is he right? How would you improve the experimental designs and/or reporting of results? Read Johnson's [14] advice on pitfalls of algorithmic experimentation. Did the authors manage to avoid most of them? What should they have done differently?
3. Find an algorithm engineering paper from one of the publication venues described in the Chapter Notes. Make a list of the design strategies described in the paper and assign them to levels of the algorithm design hierarchy described in Figure 1.1. How much did each level contribute to the speedup?

Bibliography

[1] Atkins, Derek, Michael Graff, Arjen K. Lenstra, and Paul C. Leyland, "The magic words are squeamish ossifrage" in Proceedings of *ASIACRYPT'94*, pp. 263–277, 1994.

[2] Bader, David A, Bernard M. E. Moret, Tandy Warnow, Stacia K. Wyman, and Mi Yan, "High-performance algorithm engineering for gene-order phylogenetics," Power Point talk. *DIMACS Workshop on Whole Genome Comparison*, DIMACS Center, Rutgers University, March 1 2001.

[3] Barr, R. S., B. L. Golden, J. P. Kelley, M. G. C. Resende, and W. R. Steward, "Designing and reporting on computational experiments with heuristic methods," *Journal of Heuristics* Vol 1, Issue 1, pp. 9–32, 1995.

[4] Bartz-Beielstein, Thomas, M. Chiarandini, Luis Paquette, and Mike Preuss, eds., *Experimental Methods for the Analysis of Optimization Algorithms*, Springer, 2010.

[5] Bentley, Jon Louis, "Ten Commandments for Experiments on Algorithms," in "Tools for Experiments on Algorithms," in R. F. Rashid, ed., *CMU Computer Science: A 25th Anniversary Commemorative*, ACM Press, 1991.

[6] Bentley, Jon Louis, *Writing Efficient Programs*, Prentice Hall, 1982.

[7] Bentley, Jon Louis, *Programming Pearls*, 2nd ed., ACM Press and Addison Wesley, 2000.

[8] Cherkassky, B. V., A. V. Goldberg, and T. Radzik, "Shortest paths algorithms: Theory and experimental evaluation," *Mathematical Programming* Vol 73, Issue 2, pp. 129–171, 1996.

[9] Cohen, Paul, *Empirical Methods for Artificial Intelligence*, MIT Press, 1995.

[10] Demetrescu, Camil, Irene Finocchi, and Giuseppe F. Italiano, "Algorithm Engineering," in the *Algorithms Column*, *EATCS Bulletin,* pp. 48–63, 2003.

[11] Ferragina, Paolo, Rodrigo González, Gonzalo Navarro, and Rosasno Venturini, "Compressed Text Indexes: From Theory to Practice," *ACM Journal of Experimental Algorithmics*, Vol 13, Article 1.12, December 2008.

[12] Gent, Ian P., S. A. Grant, E. MacIntyre, P. Prosser, P. Shaw, M. Smith, and T. Walsh, *How Not to Do It*, University of Leeds School of Computer Studies, Research Report Series, Report No. 97.27, May 1997.

[13] Hooker, John, "Testing heuristics: We have it all wrong," *Journal of Heuristics*, Vol 1, pp. 33–42, 1995.

[14] Johnson, David S., "A theoretician's guide to the experimental analysis of algorithms," in M. H. Goldwasser, D. S. Johnson, and C. C. McGeoch, eds., *Data Structures, Near Neighbor Searches, and Methodology: Fifth and Sixth Implementation Challenges*, AMS, pp. 215–50, 2002.

[15] LaMarca, Anthony, and Richard E. Ladner, "The influence of caches on the performance of heaps," *ACM Journal of Experimental Algorithmics*, Vol 1, 1996.

[16] LaMarca, Anthony, and Richard E. Ladner, "The influence of caches on the performance of sorting," *Journal of Algorithms*, Vol 31, Issue 1, pp. 66–104, April 1999.

[17] McGeoch, Catherine C., "Toward an experimental method in algorithm analysis," *INFORMS Journal on Computing*, Vol 8, No 1, pp. 1–15, Winter 1996.

[18] McGeoch, Catherine C., and Bernard M. E. Moret, "How to present a paper on experimental work with algorithms," *SIGACT News*, Vol 30, No 4, pp. 85–90, 1999.

[19] Moret, Bernard M. E., David A. Bader, and Tandy Warnow, "High-performance algorithm engineering for computational phylogenetics," *Journal of Supercomputing*, vol 22, pp. 99–111, 2002.

[20] Müller-Hanneman, Matthias, and Stefan Schirra, eds., *Algorithm Engineering: Bridging the Gap between Algorithm Theory and Practice*, *Lecture Notes in Computer Science*. Vol 5971, Springer, 2010.

[21] Navarro, Gonzalo, "Implementing the LZ-index: Theory versus Practice," *ACM Journal of Experimental Algorithmics*, Vol 13, Article 1.2, November 2008.

[22] Panny, Wolfgang, "Deletions in random binary search trees: A story of errors," *Journal of Statistical Planning and Inference*, Vol 140, Issue 9, pp. 2335–45, August 2010.

[23] Reddy, Raj, and Allen Newell, "Multiplicative speedup of systems," in A. K. Jones, ed., *Perspectives on Computer Science*, Academic Press, pp. 183–98, 1977.

[24] Sanders, Peter, and Dominik Schultes, "Robust, almost constant time shortest-path queries in road networks, " in C. Demetrescu, A. V. Goldberg, and D. S. Johnson, eds., *The Shortest Paths Problem: Ninth DIMACS Implementation Challenge*, AMS, 2009.

[25] Sanders, Peter, "Algorithm Engineering – an attempt at a definition," in *Efficient Algorithms*, Vol 5760 of *Lecture Notes in Computer Science*, pp. 321–430, Springer, 2009.

[26] Skiena, Steve, *The Algorithm Design Manual*, Springer, 1997. The companion Web site is *The Stony Brook Algorithm Repository*, Available From: `www.cs.sunuysb.edu/~algorith/`, 2008-07-10.

[27] Yaroslavskiy, Vladimir, Joshua Bloch, and Jon Bentley, "Quicksort 2010: Implementing and timing a family of functions." Power Point Slides. 2010.

```
Random (G, I)
   bestCount = Infinity
   bestColoring = null
   for (i=1; i<=I; i++){
       G.unColor()         //remove colors
       G.randomVertexOrder()
       count = Greedy(G)
       if (count < bestCount) {
          bestCount = count
          bestColoring = G.saveColoring()
       }
   }
   report (bestColor, bestCount)
```

Figure 2.3. The Random algorithm. Random applies Greedy repeatedly, using a random vertex order each time, and reports the best coloring found.

The color count achieved by Greedy depends on the order in which vertices are considered. For example, if vertices are colored in reverse order $8 \ldots 1$, the color count would be 3:

8	7	6	5	4	3	2	1
Red	Yellow	Yellow	Green	Red	Red	Yellow	Green

There must exist a vertex order for which Greedy finds an optimal coloring, but since there are $n!$ vertex orders, trying them all takes too much time. The Random algorithm in Figure 2.3 applies Greedy I times, using a random vertex permutation each time, and remembers the best coloring it finds. The `G.randomVertexOrder()` function creates a random permutation of the vertices, and `G.saveColoring()` makes a copy of the current coloring.

Here are some questions we could ask about the performance of Greedy and Random.

1. How much time do they take on average, as a function of n and m (and I)?
2. Are they competitive with state-of-the-art GC algorithms?
3. On what types of inputs are they most and least effective?
4. How does I affect the trade-off between time and color count in Random?
5. What is the best way to implement `G.checkColor(c,v)` and `G.assignColor(c,v)`?

Each of these questions can be attacked via experiments – but each is best answered with a different experiment. For example, question 1 should be studied

by measuring time performance on random graphs, with a wide range of n, m values to evaluate function growth best. Question 2 should be attacked by measuring both time and solution quality, using a variety of graph classes and some state-of-the art algorithms for comparison, and problem sizes that are typical in practice.

An *experimental design* is a plan for an experiment that targets a specific question. The design specifies what properties to measure, what input classes to incorporate, what input sizes to use, and so forth. Like battle plans, experimental designs may be small and tactical, suitable for reconnaissance missions, or large and strategic, for full-scale invasions.

Experimental designs can be developed according to formal procedures from a subfield of statistics known as design of experiments (DOE). But the pure DOE framework is not always suitable for algorithmic questions – sometimes designs must be based upon problem-specific knowledge and common sense. The next section describes some basic goals of algorithmic experiments. Section 2.2 introduces concepts of DOE and shows how to apply them, formally and informally, to meet these goals.

2.1 Experimental Goals

The immediate goal of the experiment is to answer the particular question being posed. But no matter what the question, some goals are common to all experimental work:

1. Experiments must be *reproducible* – that is, anyone who performs the same experiment should get similar results. For an experiment to be reproducible, the results must be *correct*, in the sense that the data generated accurately reflect the property being studied, and *valid*, which means that the conclusions drawn are based on correct interpretations of the data.
2. An *efficient* experiment produces correct results without wasting time and resources. One aspect of efficiency is *generality*, which means that the conclusions drawn from one experiment apply broadly rather than narrowly, saving the cost of more experiments.

In academic research, a third goal is *newsworthiness*. A newsworthy experiment produces outcomes that are interesting and useful to the research community, and therefore publishable. Two prerequisites for newsworthy experiments are wise choice of experimental subject (so that interesting results can reasonably be expected) and familiarity with the current literature (so that new results can be recognized). David Johnson [16] also points out that newsworthiness depends on

the "generality, relevance, and credibility of the results obtained and the conclusions drawn from them." Tips for increasing generality, relevance, and credibility of experimental results are presented throughout this section.

The rest of the section considers how to create experiments that meet these goals.

The Pilot and the Workhorse

Experiments have two flavors: the less formal *pilot* or *exploratory* study, and the more carefully designed *workhorse* study. A pilot study is a scouting mission or skirmish in the war against the unknown. It typically occurs in the information-gathering stage of a research project, before much is known about the problem at hand. It may consist of several experiments aimed at various objectives:

1. To check whether basic assumptions are valid and whether the main ideas under consideration have merit.
2. To provide focus by identifying the most important relationships and properties and eliminating unpromising avenues of research.
3. To learn what to expect from the test environment. How long does a single trial take? How many samples are needed to obtain good views of the data? What is the largest input size that can feasibly be measured?

The pilot study may be motivated by fuzzy questions, like Which data structure is better? Which input parameters appear to be relevant to performance? The workhorse study comprises experiments built upon precisely stated problems: Estimate, to within 10 percent, the mean comparison costs for data structures A and B, on instances drawn randomly from input class C; bound the leading term of the (unknown) cost function $F(n)$.

Designs for workhorse experiments require some prior understanding of algorithm mechanisms and of the test environment. This understanding may be gleaned from pilot experiments; furthermore, a great deal of useful intelligence – which ideas work and do not work, which input classes are hard and easy, and what to expect from certain algorithms – may be found by consulting the experimental literature. See the resources listed in Section 1.4.

Guideline 2.1 *Leverage the pilot study – and the literature – to create better workhorse experiments.*

How much reconnaissance is needed before the battle can begin? As a good rule of thumb, David Johnson [16] suggests planning to spend half your experimentation time in the pilot study and half running workhorse experiments. Of course it is not always easy to predict how things will turn out. Sometimes the pilot study is sufficient to answer the questions at hand; sometimes the formal experiments

raise more questions than they answer. It is not unusual for these two modes of experimentation to alternate as new areas of inquiry emerge.

The pilot and workhorse studies play complementary roles in achieving the general goals of reproducibility and efficiency, as shown in the next two sections.

Correct and Valid Results

A *spurious result* occurs when the experimenter mistakenly attributes some outcome to the wrong cause. Spurious results might seem unlikely in computational experiments, since the connection between cause and effect – between input and output – is about as clear as it gets. But the road to error is wide and well traveled. Here are some examples.

Ceiling and floor effects occur when a performance measurement is so close to its maximum (or minimum) value that the experiment cannot distinguish between effects and noneffects. For example, the following table shows solutions reported by three research groups (denoted GPR [15], CL [12], and LC [21]), on 6 of the 32 benchmark graphs presented to participants in the DIMACS Graph Coloring Challenge [17]. The left column names the file containing the input graph, the next two columns show input sizes, and the three remaining columns show the color counts reported by each group on each input.

File Name	n	m	GPR	CL	LC
`R125.1.col`	125	209	5	5	5
`R125.5.col`	125	7501	46	46	46
`mulsol.i.1.col`	197	3925	49	49	49
`DSJ125.5.col`	125	7782	20	18	17
`DSJ250.5.col`	250	31336	35	32	29
`DSJ500.5.col`	500	125248	65	57	52

Looking at just the top three lines we might conclude that the algorithms perform equally well. But in fact these color counts are optimal and can be produced by just about any algorithm. It is spurious to conclude that the three algorithms are equivalent: instead we should conclude that the experiment leaves no room for one algorithm to be better than another. This is an example of a floor effect because color counts are the lowest possible for these instances. The bottom three lines do not exhibit floor or ceiling effects and are better suited for making comparisons – for example, that performance is ordered $LC < CL < GPR$ on these inputs.

In general, floor and ceiling effects should be suspected when all the measurements from a set of trials are the same, especially if they are all at the top or bottom of their range. This is a sign that the experiment is too easy (or too hard)

to distinguish the algorithmic ideas being compared. A good time to identify and discard uninteresting inputs and poor designs is during the pilot study.

A second type of spurious reasoning results from *experimental artifacts*, which are properties of the test code or platform that affect measurements in some unexpected way – the danger is that the outcome will be mistakenly interpreted as a general property of the algorithm. Artifacts are ubiquitous: any experienced researcher can reel out cautionary tales of experiments gone awry. Here are some examples:

- Time measurements can depend on many factors unrelated to algorithm or program performance. For example, Van Wyk et al. [24] describe a set of tests to measure times of individual C instructions. They were surprised to find that the statement `j -= 1;` ran 20 percent faster than `j--;`, especially when further investigation showed that both instructions generated identical machine code! It turned out that the 20 percent difference was an artifact of instruction caching: the timing loop for the first instruction crossed a cache boundary while the second fit entirely within the cache.
- Bugs in test programs produce wrong answers. This phenomenon is pervasive but rarely mentioned in print, with the exception of Gent et al.'s [13] entertaining account of experimental mishaps:

  > We noticed this bug when we observed very different performance running the same code on two different continents (from this we learnt, DO USE DIFFERENT HARDWARE). All our experiments were flawed and had to be redone.
  >
  > All three implementations gave different behaviours .... Naturally our confidence went out the window. Enter the "paranoid flag." We now have two modes of running our experiments, one with the paranoid flag on. In this mode, we put efficiency aside and make sure that the algorithms and their heuristics do exactly the same thing, as far as we can tell.

- Pseudorandom number generators can produce patterns of nonrandomness that skew results. I once spent a week pursuing the theoretical explanation for an interesting property of the move-to-front algorithm described in Chapter 6: the interesting property disappeared when the random number generator was swapped out in a validation test. Gent et al. [13] also describe experiments that produced flawed results because "the combination of using a power of 2 and short streams of random numbers from `random()` had led to a significant bias in the way problems were generated."
- Floating point precision errors can creep into any calculation. My early experiments on bin packing algorithms (described in Section 3.2) were run on a VAX/750 and checked against a backup implementation on a Radio Shack TRS-80 Model III. At one point the two programs reported different answers when

strong connotations for programmers. A translation to standard DOE terminology appears in the Chapter Notes.

Performance metric: A dimension of algorithm performance that can be measured, such as time, solution quality (e.g. color count), or space usage.

Performance indicator: A quantity associated with a performance metric that can be measured in an experiment. For example, the time performance of Random might be measured as CPU time or as a count of the dominant operation. Performance indicators are discussed in Chapter 3 and not considered further here.

Parameter: Any property that affects the value of a performance indicator. Some parameters are *categorical*, which means not expressed on a numerical scale. We can recognize three kinds of parameters in algorithmic experiments:

- **Algorithm parameters** are associated with the algorithm or the test program. For example, Random takes parameter I, which specifies a number of iterations. Also, the `G.checkColor(c,v)` and `G.assignColor(c,v)` functions could be implemented in different ways: the source code found in each function is a categorical parameter.
- **Instance parameters** refer to properties of input instances. For example, *input size* is nearly always of interest – in graph coloring, input size is described by two parameters n and m. Other graph parameters, such as maximum vertex degree, might also be identified. The (categorical) parameter *input class* refers to the source and general properties of a set of instances. For example, one collection of instances might come from a random generator, and another from a cell tower frequency assignment application.
- **Environment parameters** are associated with the compiler, operating system, and platform on which experiments are run.

Factor: A parameter that is explicitly manipulated in the experiment.

Level: A value assigned to a factor in an experiment. For example, an experiment to study Random might involve four factors set to the following levels: $class = (random, cell\ tower)$; $n = (100, 200, 300)$, $m = (sparse, complete)$, and $I = (10^2, 10^4, 10^6)$.

Design point: A particular combination of levels to be tested. If all combinations of levels in the preceding example are tested, the experiment contains $36 = 2 \times 3 \times 2 \times 3$ design points: one of them is $(class = random, n = 100, m = 4950, I = 100)$.

Trial or **test:** One run of the test program at a specific design point, which produces a measurement of the performance indicator. The design may specify some number of (possibly random) trials at each design point.

Fixed parameter: A parameter held constant through all trials.

Noise parameter: A parameter with levels that change from trial to trial in some uncontrolled or semicontrolled way. For example, the input class may contain random graphs $G(n, p)$, where n is the number of vertices and p is the probability that any given edge is present; the number of edges m in a particular instance is a semicontrolled noise parameter that depends on factors n and p. In a different experiment using instances from a real-world application, both n and m might be considered noise parameters (varying but not controlled) if, say, the design specifies that input graphs are to be sampled from an application within a given time frame.

Computational experiments are unusual from a DOE perspective because, unlike textbook examples involving crop rotations and medical trials, the experimenter has near-total control over the test environment. Also, very often the types of questions asked about algorithm performance do not exactly match the DOE framework. This creates new opportunities for developing high-yield designs, and for making mistakes. The next two sections survey two aspects of experimental design in this context. Section 2.2.1 surveys input classes and their properties, and Section 2.2.2 presents tips on choosing factors, levels, and parameters to address common categories of questions.

2.2.1 Selecting Input Classes

Input instances may be collected from real-world application domains or constructed by generation programs. They can be incorporated in algorithmic experiments to meet a variety of objectives, listed in the following.

- *Stress-test inputs* are meant to invoke bugs and reveal artifacts by invoking boundary conditions and presenting easy-to check cases. An input generator for Greedy, for example, might build an empty graph (no edges), a complete graph (full edge sets), a variety of graphs with easy-to-check colorings (trees, rings, grids, etc.), and graphs that exercise the vertex permutation dependencies. Some generic stress-test resources are also available – for example, Paranoia [18] is a multilanguage package for testing correctness of floating point arithmetic.
- *Worst-case* and *bad-case instances* are hard (or expensive) for particular algorithms to solve. These instances are used to assess algorithm performance boundaries. For example, Greedy exhibits especially poor performance on

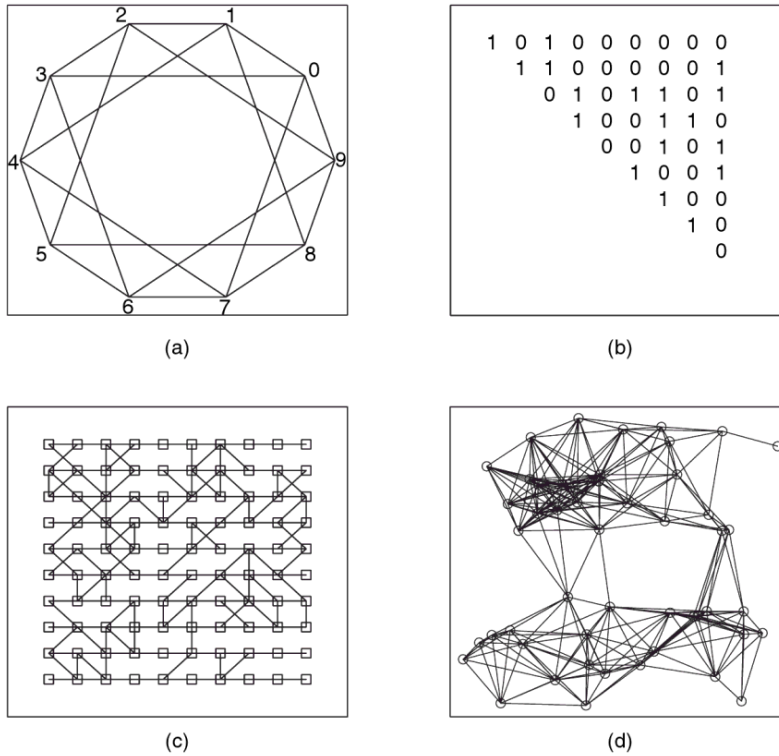


Figure 2.4. Input classes for graph coloring algorithms. Panel (a) shows a crown graph, which is known to be hard for Greedy to solve. Panel (b) shows the adjacency matrix for a random graph of $n = 10$ vertices, where each edge (labeled 1) is selected with probability $p = 0.5$. Panel (c) shows a semi-random grid graph. Panel (d) shows a proximity graph, which mimics a cell-phone tower application.

crown graphs like the one shown in Figure 2.4 (a). In this graph each even-numbered vertex is connected to every odd-numbered vertex except the one directly across from it. Crown graphs can be colored by using just two colors, but Greedy may use up to $n/2$ colors.

- *Random inputs* are typically controlled by a small number of parameters and use random number generators to fill in the details. For example, Figure 2.4 (b) shows the upper diagonal of an adjacency matrix for a random graph $G(n, p)$: here $n = 10$ and each edge (denoted 1) is present with probability $p = 0.5$. Random inputs are useful for measuring average-case performance under some theoretically tractable model. Also, if every instance is generated with nonzero probability, experiments using random inputs can reveal the range of all possible outcomes.

- *Structured random inputs* come from generators built for two purposes:
  - *Algorithm-centered generators* are built with parameters that exercise algorithm mechanisms. For example, the performance of Greedy depends partly on the *regularity* of the input graph. In a perfectly regular graph all vertices have the same number of incident edges: an example of a regular graph of degree 8 is a grid-graph where each vertex has four neighbors at the N, S, E, W compass points and four neighbors on the NE, NW, SE, SW diagonals. Figure 2.4 (c) shows a semi-random grid graph – in this instance, each non boundary vertex is connected to its E, W neighbors with probability 1, and to each of its SW, S, SE neighbors with probability 1/3. A generator of these graphs can be used to focus on how Greedy responds to changes in graph regularity.
  - *Reality-centered generators* capture properties of real-world inputs. For example, the cell tower application described previously can be modeled by placing random points in the unit square, with edges connecting points that are within radius r of one another. This type of graph is called a *proximity graph*. A proximity graph with $n = 50$ and $r = 0.25$ is shown in Figure 2.4 (d). These types of generators can often be improved with a little (Web) research into quantitative properties of reality: how many cell towers are typically found in different types of regions (urban, rural, mountainous, etc.)? What it the typical broadcast distance?
- *Real instances* are collected from real-world applications. A common obstacle to using these types of instances in algorithmic experiments is that they can be difficult to find in sufficient quantities for thorough testing.
- *Hybrid instances* combine real-world structures with generated components. This approach can be used to expand a small collection of real instances to create a larger testbed. Three strategies for generating hybrid graphs for graph coloring are as follows: (1) start with a real-world instance and then perturb it by randomly adding or subtracting edges and/or vertices; (2) create a suite of small instances from random sections of a large instance; or (3) build a large instance by combining (randomly perturbed) copies of small instances.
- Sometimes a *public testbed* is available for the algorithm being studied. In the case of graph coloring, testbed inputs are available at the DIMACS Challenge Web site [17], and Joseph Culberson's Graph Coloring Resources Page [11], among other sources. In academic circles, running experiments using testbed instances maximizes relevance and newsworthiness by producing results that are directly comparable to results of experiments carried out by others. But it is not necessary to restrict experiments to testbed instances; any of the categories in this list may be used to expand and improve on earlier work.

Each category has its merits and drawbacks. Input generators can produce large numbers of instances; they are more compact to store and share; and they can be

tuned to provide broad coverage of an input space or to focus on properties that drive algorithm performance or mimic realistic situations. But they may fail to answer what may be the main question: how well does the algorithm perform in practice?

Inputs from real-world applications are ideal for answering that question, but, on the other hand, they can be hard to find in sufficient quantities for testing. Also they may contain highly problem-specific hidden structures that produce hard-to-explain and therefore hard-to-generalize results.

The choice of instance classes to test should reflect general experimental goals as well as the specific question at hand:

- To meet goals of correctness and validity, use stress-test inputs and check that random generators really do generate instances with the intended properties. Use pilot experiments to identify, and remove from consideration, instances that are too easy or too hard to be useful for distinguishing competing algorithmic ideas.
- For general results, incorporate good variety in the set of input classes tested. But avoid variety for variety's sake: consider how each class contributes new insights about performance. Worst-case instances provide general upper bounds; random generators that span the input space can reveal the range of possible outcomes. Real-world instances from application hot spots can highlight properties of particular interest to certain communities; algorithm-centered inputs reveal how the algorithm responds to specific input properties; and so forth.
- More ambitious analyses tend to require more general input classes and tight control of parameters. When the goal is to build a model of algorithm performance in terms of input parameters, success is more likely if the inputs obey simple random models or are produced by algorithm-centered generators that allow explicit control of relevant properties, so that experimental designs can focus on the question that prompts the experiment.

Guideline 2.4 *Choose input classes to support goals of correctness and generality, and to target the question at hand.*

In addition to the preceding considerations, Dorothea Wagner [23] has proposed guidelines for developing and maintaining public instance testbeds to support algorithm research. One common complaint is that testbeds are often assembled without much of a screening process and may contain several uninteresting and/or unjustified instances. There is a need for more testbed instances that meet at least one of the following requirements.

- They have features that are relevant to algorithm performance.
- They have provable properties.
- They permit controlled experiments using parameterization.

Guideline 2.6 *When comparing algorithm (or program) design options, choose performance indicators and factors to highlight the differences among the options being compared.*

Another early experimental goal is to get a rough idea of the functional relationship between key parameters (especially input size) and algorithm performance.

A good design strategy in this situation is to try a *doubling* experiment. Sedgewick [22] points that the growth rates of many common functions in algorithm analysis are easy to deduce if cost is measured as n doubles. For example, suppose we measure cost $C(n)$ at problem sizes $n = 100, 200, 400, 800 \ldots$. The results can be interpreted as follows:

1. If measurements do not change with n, $C(n)$ is constant.
2. If costs increment by a constant as n doubles, for example, if $C(n) = 33, 37, 41, 45$, then $C(n) \in \Theta(\log n)$.
3. If costs double as n doubles, $C(n)$ is linear.
4. To determine whether $C(n) \in \Theta(n \log n)$, divide each measurement by n and check whether the result $C(n)/n$ increments by a constant.
5. If cost quadruples each time n doubles, $C(n) \in \Theta(n^2)$.

Similar rules can be worked out for other common function classes; see Sedgewick [22] for details.

Doubling experiments are valuable for checking whether basic assumptions about performance are correct. For example, Bentley [5] describes a study of the `qsort` function implemented in the S statistical package. Although the function implements Quicksort, which is well known to be $O(n \log n)$ on average, his doubling experiment revealed the following runtimes (in units of seconds):

```
$ time a.out 2000
real   5.85s
$ time a.out 4000
real 21.65s
$ time a.out 8000
real 85.11s
```

This clearly quadratic behavior was caused by "organ-pipe" inputs of the form $123 \ldots nn \ldots 321$ and was subsequently repaired.

An example of a doubling experiment that incorporates two parameters n and m appears in Section 3.1.1.

Guideline 2.7 *Try a doubling experiment for a quick assessment of function growth.*

Another question that arises early in some experimental studies is to determine when the algorithm has converged. In the context of iterative-improvement heuristics, convergence means, informally, that the probability of finding further improvements is too small to be worth continuing. Another type of convergence arises in stochastic algorithms, which step through sequences of states according to certain probabilities that change over time: here convergence means that the transition probabilities have reached steady state, so that algorithm performance is no longer affected by initial conditions. In this context the problem of determining when steady state has occurred is sometimes called the *startup problem*.

A *stopping rule* is a condition that halts the algorithm (i.e., stops the the experiment) when some event has occurred. Experimental designs for incremental and stochastic algorithms require stopping rules that can terminate trials soon after – but no sooner than – convergence occurs.

A poorly chosen stopping rule either wastes time by letting the algorithm run longer than necessary or else stops the algorithm prematurely without giving it a chance to exhibit its best (or steady-state) performance. The latter type of error can create *censored data*, whereby a measurement of the (converged) cost of the algorithm is replaced by an estimate that depends on the stopping rule. See Section 7.1.1 for more about the problem of data censoring.

Good stopping rules are hard to find: here are some tips on identifying promising candidates.

- Avoid stopping rules based on strategies that cannot be formally stated, like "Stop when the cost doesn't appear to change for a while." A good stopping rule is precisely articulated and built into the algorithm, rather than based on hand tuning.
- To ensure replicability, do not use rules based on platform-specific properties, such as "Stop after 60 minutes have elapsed."
- If the total number of states in a stochastic process is small, or if a small number of states are known to appear frequently, consider implementing a rule based on state frequencies: for example, stop after every state has appeared at least k times.
- A related idea is to assign a cost to every state and to compute running averages for batches of b states in sequence – stop the algorithm once the difference in average cost $C(b_x..b_{x+i})$ and $C(b_y..b_{y+i})$ is below some threshold. A graphical display of batch measurements may show a "knee" in the data where the transition from initial states to steady state occurs.
- Sometimes it is possible to implement a test of some property that is a precondition for the steady state. For example, it may be known that a given stochastic graph coloring algorithm does not reach steady state until after every vertex has changed color at least once.

We next consider designs for fitting and modeling algorithmic cost functions.

Analyzing Trends and Functions

The central problem of algorithm analysis is to describe the functional relationship between input parameters and algorithm performance. A doubling experiment can give a general idea of this relationship but often we expect more precision and detail from the experiment.

Suppose we want to analyze time performance of two implementations of Random. This algorithm depends on how functions `G.checkColor(c,v)` and `G.assignColor(c,v)` are implemented. Let k be the maximum color used by the algorithm in a given trial: `checkColor` is invoked at most nk times, and `assignColor` is invoked n times. Two implementation options are listed below.

Option a. Each vertex v has a color field: check for a valid coloring by iterating through the neighbors of v. The total number of comparisons in `checkColor` is at most mk, once for each edge and each color considered; the cost per call to `assignColor` is constant. Therefore, total cost is $O(mk+n)$.

Option b. Each vertex has a color field and a "forbidden color" array that is updated when a neighbor is assigned a color. Each call to `checkColor` is constant time, and each call to `assignColor` is proportional to the number of neighbors of v. Total cost is $O(nk+m)$.

The experimental design includes factors $Option = (a,b)$, input sizes n, m, and iteration count I. The goal is to develop a function to describe the comparison cost of Random in terms of these four factors. Since *Option* is categorical, we use two functions $f_a(n,m,I)$ and $f_b(n,m,I)$. Since the algorithm iterates I times, we know that $f(n,m,I)$ is proportional to I; let $g_a(n,m)$ and $g_b(n,m)$ equal the average cost per iteration of each option.

The experimental design problem boils down to how to choose levels for n and m to give the best views of function growth. One idea is to use a grid approach, with $n = 100, 200, \ldots max\_n$, and $m = 100, 200, \ldots max_m$, omitting infeasible and uninteresting combinations: for example, m must be at most $n(n-1)/2$, and coloring is trivial when m is small. Another idea is to select a few levels of m that are scaled by n, for example, $m_1 = n(n-1)/2$ (complete graphs), $m_2 = m_1/2$ (half-full), and $\ldots m_3 = m_1/4$ (quarter-full). Scaled design points are more informative than grid-based designs whenever the scaled functions are expected to have similar shapes – in this case, similar shapes would arise from a property that is invariant in the ratio m/n.

Guideline 2.8 *The problem of analyzing a multidimensional function can be simplified by focusing on a small number of one-dimensional functions, ideally with similar shapes.*

This rough analysis, which ignores low-order terms and relies on some untested assumptions, yields the following preliminary cost formula:

$$W(n,m) = qn\log_2 n + bm\log_2 n + rmn. \tag{3.1}$$

To check the validity of this model we use a simple doubling experiment as described in Section 2.2.2. Since this is a randomized algorithm running on real-world inputs, we expect a reasonably close correspondence, but not a perfect match between the model and the data.

Guideline 3.2 *Use a doubling experiment to perform a quick validation check of your cost model.*

The validation experiment runs MC on a file called `total` that contains three volumes of English text described in the table that follows. All text files mentioned in this section were downloaded from Project Gutenberg [20].

File	Text	n
`huckleberry`	*Huckleberry Finn*, by Mark Twain	112,493
`voyage`	*The Voyage of the Beagle*, by Charles Darwin	207,423
`comedies`	Nine comedies by William Shakespeare	337,452
`total`	Combined `comedies`, `huckleberry`, `voyage`	697,368

The experiment reads the first n words of `total` and measures `qcount`, `bcount`, and `rcount` in one trial each at design points with $k = 1$, $n = (10^5, 2\times10^5, 4\times10^5)$, and $m = (10^5, 2\times10^5, 4\times10^5)$.

First we check whether `qcount` is proportional to $n\log_2 n$. Since doubling n increases $\log_2 n$ by 1, we expect `qcount`$/n$ to increment by a constant at each level. The following table shows the results.

	$n = 10^5$	2×10^5	4×10^5
`qcount`$/n$	12.039	13.043	14.041

The data behave as expected, so we accept $qn\log_2 n$ to model the cost of initialization. Next we consider `bcount`, which should grow as $m\log_2 n$.

	$n = 10^5$	$n = 2\times10^5$	$n = 4\times10^5$
$m = 10^5$	133,606	143,452	153,334
$m = 2\times10^5$	267,325	287,154	306,553
$m = 4\times10^5$	534,664	574,104	613,193