INFORMS Journal on Computing
Vol. 8, No. 1, Winter 1996

0899-1499/96/0801-0001 $01.25

FEATURE ARTICLE

Toward an Experimental Method for Algorithm Simulation

CATHERINE C. MCGEOCH / *Department of Mathematics and Computer Science, Amherst College, Amherst, MA 01002; Email: ccm@cs.amherst.edu*

(Received: July 1994; revised: January 1995, May 1995; accepted: October 1995)

This feature article surveys issues arising in the design, development, and execution of computational experiments to study algorithms. An algorithm is viewed here as an abstract model of an implemented program: experiments are performed to study the model, and new insights about the model can be applied to predict program performance. Issues related to choosing performance measures, planning experiments, developing software tools, running tests, and analyzing data are considered. Some hazards and difficulties of computational research that arise particularly in the context of algorithmic problems are also surveyed.

A multitude of practical questions confronts the scientist who undertakes an experimental study of one or more algorithms. What should I measure? How many data points should I gather? How should I analyze the data? Can I trust the results? Will I learn anything useful?

The answers, of course, depend upon the particular problem being studied and upon the goals of the research project. Researchers in the natural sciences have available an extensive lore of laboratory techniques to guide the development of rigorous and conclusive experiments. This has not been the case in algorithmic research, however, perhaps because algorithm analysis has long been viewed as primarily a deductive science.

However, as the power of this kind of research has become evident in recent years, more sophisticated experimental studies have appeared, and interest in methodology has grown. In support of these trends, this feature article describes some aspects of a proposed methodology for experimentation on algorithms. The goal here is to collect and disseminate the experiences of scientists active in this field, and thereby to contribute to a growing lore of experimental research on algorithms. We can move forward by adopting the successful strategies—and avoiding the mistakes—of others.

This feature article is *not* about doing computational experiments to compare programs. The experiments considered here study properties of algorithms, which are abstract mathematical objects. However, these distinct problem areas are closely related. Figure 1 illustrates a common paradigm of simulation research. We have some complex *real-world process* (say an economic system), and we wish to predict how the process will behave in future situations. It is not possible to test the process directly, perhaps because we do not know what future situations will arise, but we do want to understand how it responds in various scenarios. We develop a *mathematical model* that captures important properties of the process. If we can analyze the model—that is, derive a mathematical statement that answers sufficiently the questions motivating the study—then we are done. If the model cannot be analyzed, we implement a *simulation program* and run computational experiments on the simulation program. The experimental results produce insights about the model and, by extension, about the real-world process.

For our purposes, the real-world process is an application program running in a particular computing environment and solving problems for real input instances. The abstract mathematical model is an algorithm: here we use the word algorithm in a very broad sense that allows wide variations in the kinds of properties that are incorporated (more about this later). When the algorithm cannot be analyzed sufficiently, a simulation program is developed and experiments are run. The simulation program may in fact be identical to the application program, but we shall treat them as conceptually different objects.

What do we gain by making this distinction between application programs and simulation programs? The power of simulation lies in the complete control the scientist has over the experimental environment. We can design experiments to test specific hypotheses, we can change performance measures as needed, and we can open up the black box of the simulation program and probe its internal mechanisms. In contrast, the scientist who studies application programs *in situ* must take care that the tests themselves do not affect the results, and must sacrifice some degree of control over the experiment.

Why do we base mathematical models of application programs on algorithms? Because algorithmic models capture most of the mechanisms and properties that determine program performance. Certainly it is widely recognized that an algorithm, in the traditional sense of the word, provides

Subject classifications: Computational experiments, algorithm design, algorithm development.

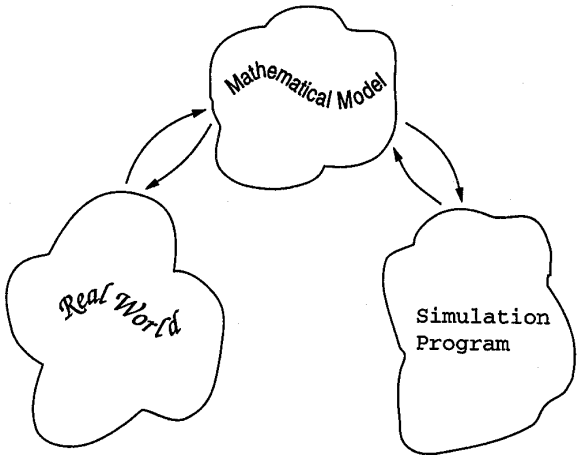

Figure 1. The Paradigm of Simulation Research.

only a very rough indication of program performance: low level implementation details can have a far greater impact on performance than algorithmic modifications. Therefore, in this feature article, we extend the definition of the word algorithm to include environment-specific details. That is, algorithmic models of programs can vary according to their level of *instantiation*.

A minimally instantiated algorithm (such as simulated annealing) contains few implementation details and is probably more properly called an algorithm template. A more instantiated algorithm may incorporate basic implementation strategies (such as whether to use a stack or queue for a given data structure); an even more instantiated algorithm may specify programming tricks that give constant-factor speedups. A highly instantiated algorithm might describe details specific to a particular programming language or computer architecture.

The discussion in this feature article concentrates on the right half of Figure 1. That is, we focus on problems of simulating algorithms: developing software tools, designing efficient experiments, avoiding errors, and analyzing the data. Issues arising in the left side of Figure 1 are not addressed here. In particular, the considerable problems of developing and validating sufficiently predictive models of performance are mentioned only briefly.

Section 1 surveys previous work, introduces some notation, and presents a small example study to illustrate several points raised in later sections. Section 2 considers issues arising at various stages in the simulation process. Section 3 surveys difficulties and hazards of simulation research on algorithms—in some cases the difficulties can be eliminated, and in some cases they can only be reduced. Section 4 presents concluding remarks.

No background in statistics or probability theory is assumed here beyond familiarity with terms such as *mean, expectation, density function,* and *correlation*. Definitions of these standard terms can be found in any introductory textbook on statistics (such as [31], [40], and [62]). Familiarity with the terminology and notation of algorithm analysis at about the level of an undergraduate course is also assumed (see, for example, [26] or [71]).

1. Computational Experiments in Context

In order to provide a context for further discussions, this section surveys related work, establishes some notation, and presents a small computational study of an algorithm to illustrate the notation and some ideas developed in later sections.

1.1. Building a Reference Shelf

Methodological issues of algorithm simulation intersect with several disciplines, including operations research, theoretical computer science, performance analysis, simulation, statistics, data analysis, and scientific visualization. There is no shortage of information sources; rather, the difficulty is in sorting through this enormous literature to find works most immediately applicable to algorithmic problems.

To assist with this winnowing process, this section surveys articles and texts most relevant to algorithmic simulation. This is a beginner's reading list that errs on the side of brevity rather than completeness. More extensive bibliographies can be found in the cited survey articles.

General resources. Until very recently there have been few mainstream publication outlets for research on methodological issues of algorithm simulation. Much of the lore gathered here has been developed by participants in mini-sessions and workshops such as [30], [45], [47], [48], [58], and [65]. Many of the anecdotes in the following pages were exchanged during coffee breaks at workshops like these.

A good source of ideas and practical advice for doing simulation experiments is the annual *Proceedings of the Winter Simulation Conference*, although this conference focuses primarily on discrete event simulation. The December 1994 issue of the *Annals of Operations Research* is entirely devoted to papers on simulation methodology. The *INFORMS Journal on Computing* has emerged as a leading source of articles on experimental methods for problems of interest to the operations research community. Many papers presented at the annual *ACM-SIAM Symposium on Discrete Algorithms* contain experimental studies of algorithms. The electronic *ACM Journal of Experimental Algorithmics* will contain refereed articles as well as programs, test instances, and instance generators.

For introductions to general topics in simulation see Bratley, Fox, and Schrage[18] and Kleijnen and Van Groenendaal.[52] Barton,[5] Lin and Rardin,[57] and Nance, Moose, and Foutz[63] discuss issues of model development and experimental design for computational testing of programs. Devroye[32] is the definitive reference on techniques for generating nonuniform random numbers. For a survey of techniques for generating uniform random variates see L'Ecuyer.[56] Park and Miller[66] discuss implementation issues for a specific uniform generator.

Survey articles. Hooker[44] laments the current state of computational research on algorithms and programs. He notes,

among other things, that it is notoriously difficult to extrapolate results from one class of input instances to another. Therefore, it is rarely possible for a researcher to claim that the experimental results are predictive of performance in practice. To finesse this problem, Hooker suggests that instance generators be developed to produce structured inputs based upon a very large number of parameters. Computational tests can be developed to study how performance depends on parameter settings, and practitioners can match parameter settings to their particular applications. Much of the discussion in Section 2 concerns this kind of testing.

Hooker also describes how *explanatory rationales* can be developed to support reliable predictions. These rationales are based on *empirical models* of experimental data, which make no assumptions about the process that produces the data. In contrast, the method proposed in this feature article is to build *mechanistic models* that are based on partial understanding of the process. Discussion of the relative merits of empirical models and mechanistic models can be found in textbooks on statistics, for example Box, Hunter, and Hunter[17] (Chapter 16) and Rawlings[68] (Section 7.1). To summarize very briefly, a good mechanistic model provides a much safer basis for extrapolation beyond the range of experiments, and can give more parsimonious descriptions of results. Sometimes, however, it is difficult to understand the interactions between programs, operating systems, and architectures that can drive performance. In such a case, the best strategy may be to develop an empirical model. Which approach is most suitable for a given research problem may be a matter of judgment.

Kelton[51] surveys current research trends in several areas of simulation. His focus is on applications of simulation to problems in operations research and management science, but many of his remarks apply to general algorithmic research as well. He traces the growth of interest in simulation over the past 25 years and provides a comprehensive bibliographic survey. Greenberg[43] discusses some broad philosophical issues.

Case studies. For testimonial descriptions of powerful interactions between theoretical and experimental approaches to algorithm analysis, see Bentley et al.,[13, 14] Cherkassky, Goldberg, and Radzik,[20] Fredman et al.[39] and Todd.[75]

A few case studies have appeared in which authors present experimental results together with detailed discussions of experimental techniques. Ahuja, Magnanti, and Orlin[1] describe methods for choosing combinatorial performance measures that are highly predictive of program running times, and illustrate their ideas using algorithms for network flows. Bentley[11] presents guidelines for developing software environments for experimental research. Bentley[10, 12] illustrates several methods of graphical data analysis in studies of heuristics for the TSP. Gent and Walsh[41] describe problems arising in their computational study of algorithms for satisfiability. Golden and Stewart[42] show how sophisticated statistical analyses can be applied to algorithms for the TSP. McGeoch[59] presents four case studies in a survey of methodological issues for algorithm analysis. A tutorial on application of variance reduction techniques to algorithmic problems appears in [61]. These and other case studies are discussed later in more specific contexts.

Data analysis. Graphical and exploratory methods of data analysis are highly appropriate for algorithmic problems. Exploratory data analysis (EDA) is a subfield of statistics in which few *a priori* assumptions are made about the data set. Instead, summarization and transformation techniques (like histograms and log-log plots but considerably more sophisticated) are used to discover underlying structures and patterns. Tukey's text on EDA[76] is the seminal work; another excellent introduction is given by Velleman and Hoaglin.[77]

Graphical data analysis is another subfield that concentrates on visual methods for exploring relationships in data. Graphs are especially good for handling the huge data sets that arise in algorithmic problems. For introductions to graphical methods see Chambers et al.,[19] Cleveland,[23] and duToit, Steyn, and Stumpf.[34] Jones[49] surveys current research trends in the intersection of visualization and optimization.

Parametric techniques of least-squares regression, and especially of regression diagnostics, figure prominently in research on algorithms because the frequent goal is to discover functional relationships between input parameters and performance measurements. For introductions to modern regression techniques see Atkinson[4] and Rawlings.[68]

The classic paper by Crowder, Dembo, and Mulvey[27] gives an excellent discussion of how to *report* experimental results in published work. The authors survey issues of experimental and statistical integrity, outline basic standards for carrying out and reporting computational experiments, and present a reviewer's checklist for evaluating the quality of experimental papers. Cleveland[22] discusses principles of graphical presentation.

1.2. Terminology and Notation

The following notation is used throughout the feature article. Suppose that the performance of algorithm A is to be studied for a given parameterized class of input instances. Algorithm A and the instance class comprise the *experimental model*. The simulation study is undertaken to learn how some *performance measure* X depends on certain *experimental parameters*, say n and d. (It should be pointed out that the word parameter has an entirely different meaning in statistics, where the terms factor or independent variable would be used instead. Various subareas of statistics use different terms for other concepts defined in this section. The terms used here are consistent with standard usage in algorithm analysis.) Performance measures are selected according to the open questions that motivate the experiment and the level at which the algorithm is instantiated. We assume throughout that the measures are *combinatorial* in nature—that is, they represent counts of basic operations, units of space, or some measure of solution quality. For more discussions of this point see Section 2.1.

Several types of parameters may be incorporated in a study. For example, instances may be generated according to some *input parameters* (such as problem size, graph density, or other structural properties). There may also be *algo-*

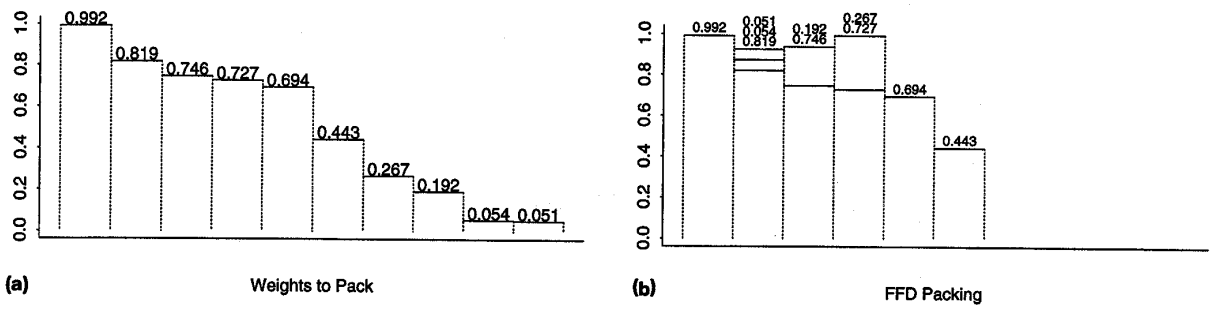

Figure 2. An FFD packing of a sorted list of 10 weights.

rithm parameters that correspond to specific implementation strategies (such as whether to use a queue or a stack for a basic data structure). Some parameters vary along a numeric scale, and some do not. For example, problem size $n \in I^+$ is a numeric parameter, but the data structure parameter $d \in$ {*queue, stack*} is not.

We develop experiments to answer open questions about the algorithm. Ideally, we fix those parameters that are irrelevant to the questions, control those parameters that are most important, and randomize those properties that have a small effect on performance, or that we do not understand or cannot control. Randomization may be present in the input model or in the algorithm.

A *simulation program* that mimics the behavior of the algorithm, and an input generator, are implemented. Experiments are carried out. A *design point* corresponds to fixed parameter settings (also called levels); for example, $n = 1000$ and d = *queue* may form a design point. There may be several *random trials* at each design point. Each trial corresponds to one run of the program on a single instance. If the algorithm is probabilistic then several trials may be run on a single instance. An *experiment* involves t random trials taken at several design points. A single trial i produces a value for an *output variate* $X_i(n, d)$ that corresponds to the performance measure X. Some other variates $W_i(n, d)$ may be recorded for comparison to the performance variate $X_i(n, d)$.

Because of randomization in the model, the distribution of X is described by an unknown density function $f_{n,d}(x)$, having unknown mean $\mu(n, d)$. Computational experiments often begin with an effort to understand the mean $\mu(n, d)$ by studying the sample mean $\bar{X}(n, d) = (1/t)\Sigma_{i=1}^{t} X_i(n, d)$. We say that $\bar{X}(n, d)$ is an *estimator* of $\mu(n, d)$. More detailed information about $f_{n,d}(x)$ can be obtained by studying other properties of $X_i(n, d)$. These terms are applied in an example presented in the next section.

1.3. An Example

The one-dimensional bin-packing problem is well known and easy to state. Given a list containing n weights each in the range (0, 1], arrange the weights in unit-capacity bins so as to minimize the total number of bins used. Since this problem is NP-hard, several heuristic algorithms have been proposed. The *First Fit Decreasing* rule (FFD) sorts the weight list in decreasing order and then packs each weight in the first bin that can contain it. Figure 2 shows an FFD packing of a list containing 10 weights.

How well does FFD pack in general? We will perform a small experimental study using lists L_n containing n weights drawn uniformly and independently from (0, 1]. Here n is an input parameter. Let the *total weight* in the list be denoted W, and note that $E[W] = n/2$ for this random model. One interesting measure of packing quality is the *bin ratio* $R = B/OPT$, where B denotes the number of bins used by FFD, and OPT denotes the number of bins in an optimal packing of the list L_n. The mean bin ratio is denoted $\rho(n)$.

Since bin packing is NP-hard, it is in general not feasible to compute OPT for a given list. As an alternative to measuring the bin ratio, we define two new performance measures, the *packing ratio* $P = B/W$, and *empty space* $S = B - W$, which have unknown means $\phi(n)$ and $\sigma(n)$, respectively. Note that empty space can be translated into packing ratio by the formula $P = (S + W)/W = S/W + 1$. Furthermore, for any list we have $\lceil W \rceil \leq OPT \leq B$. Therefore $B/OPT \leq B/W$, or equivalently $R \leq P$ and $\rho(n) \leq \phi(n)$.

Please note that the experimental study sketched below was selected to illustrate several ideas developed in later sections and not because of its impact on our state-of-the-art understanding of FFD. Analytical bounds are known for $\rho(n)$, $\sigma(n)$, and $\phi(n)$ for this model (see references at the end of this section). Nevertheless, many key theoretical advances on average-case analysis of FFD and other packing rules were directly stimulated by experimental research similar to that developed here.

The first experiment. Figure 3.a shows the observed packing ratios $P_i(n)$ in 25 random trials each at the design points n = 1000,2000, ... 128,000 (doubling n each time). The dotted line marks the average packing ratio $\bar{P}(n)$ observed at n = 128,000. From the trend toward decreasing means in this graph we might conjecture that as n grows larger, $\rho(n) \leq \phi(n) \leq c$ where c is near 1.0017.

Figure 3.b plots observed empty space $S_i(n)$ against n for the same experiment. Figure 3.c plots the data on a double-logarithmic scale. A least-squares regression fit is also shown. The superimposed line obeys the formula $\hat{y}(n) = 0.515 \ln n - 1.57$. Figure 3.d shows the residuals from this fit,

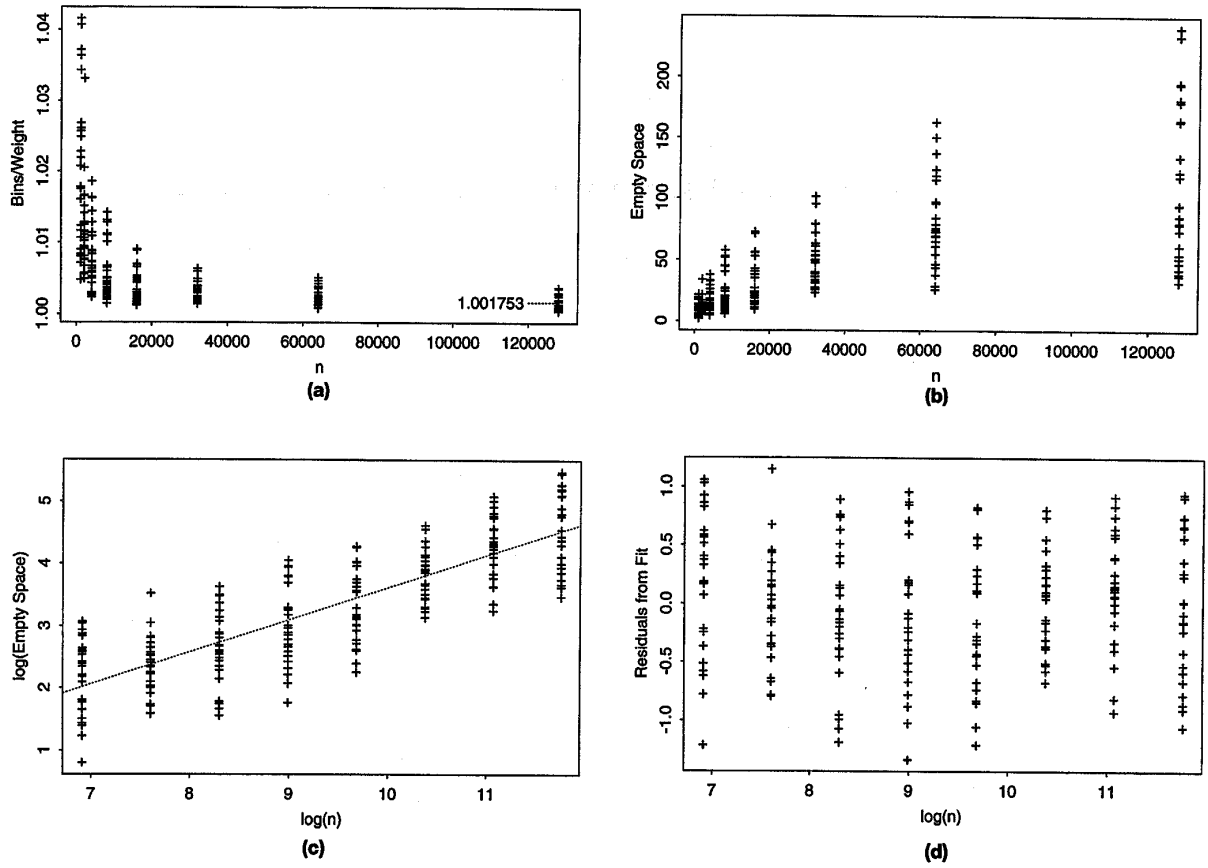

Figure 3. The First FFD Experiment. Graph (a) plots n vs. packing ratio. Graph (b) plots n vs. empty space. Graph (c) plots the data on a double-logarithmic scale; a regression line is also shown. Graph (d) shows the residuals from the regression fit.

defined by $\ln S_i(n) - \hat{y}(n)$. The residuals show no obvious curvature up or down, which suggests that the line provides a fairly good description of how the data grows with n. Other regression diagnostics (not shown here) indicate that the line may slightly overestimate growth in the data. (These statistical terms and techniques are discussed more fully in Section 2.4.)

Translating back to the linear scale we have an observation that $\bar{S}(n)$ grows no faster than $n^{0.515}/e^{1.57}$, and a conjecture that $\sigma(n)$ grows approximately as $cn^{0.515}$ for a constant c. This is a stronger conjecture than the earlier one: if $\sigma(n)$ does indeed grow more slowly than n, then $\phi(n) \to 1$. Since $\rho(n) \leq \phi(n)$, this would imply that FFD is asymptotically optimal for this random model.

Now we explore this conjecture: Why should empty space grow approximately as square-root n? We might hypothesize that empty space S reflects some random noise due solely to random variation in individual weights. If this were true then variations in observed empty space $S_i(n)$ would not be correlated with variations in, say, the total weight $W_i(n)$ of the list or the number of bins $B_i(n)$ used in the FFD packing.

The second experiment. To test this hypothesis we take 1,000 random trials at the design point n = 128,000. The graph in Figure 4.a suggests that in fact $S_i(n)$ is correlated with $W_i(n)$. The smoothed line on the plot shows a clear increasing trend at the right. Furthermore, as Figure 4.b indicates, a high value for empty space is associated with a high number of heavy weights (weights greater than 0.5) in a list. We try a new hypothesis: Most of the empty space in a packing will be found in bins containing surplus heavy weights, which are packed one-to-a-bin.

The third experiment. To investigate this hypothesis we look at the breakdown of empty space by number of weights per bin. Let a *gap* be the amount of empty space in a single bin. Table I shows how gaps are distributed according to counts of weights per bin in one trial at n = 128,000. Table entries correspond to weight counts per bin (rows) and ranges of gap values (columns). Each entry shows how many bins fit the category. For example, there were 63,493 bins containing 2 weights, and 59,583 of them had gaps in the range [0, .00005). In this FFD packing, no bin contained more than 5 weights.

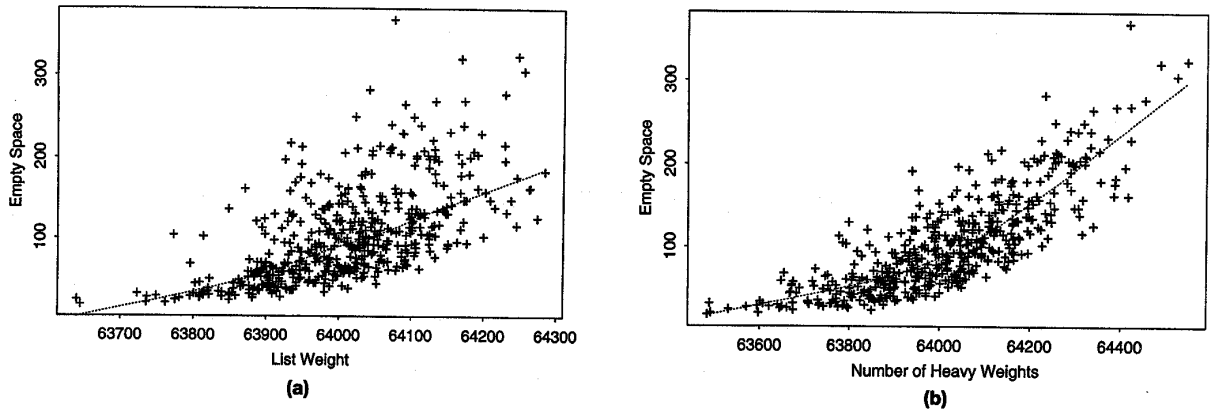

Figure 4. The Second FFD Experiment. Graph (a) shows list weight vs. empty space. Graph (b) plots number of heavy weights vs. empty space.

Table I. Breakdown of Bins According to Number of Weights Per Bin and Gap

Weights Per Bin	Total Bins	Gap Ranges		
		[0, .00005)	[.00005, .0001)	[.0001, 1)
1	259	14	1	244
2	63493	59583	1335	2575
3	242	16	3	223
4	40	3	4	33
5	1	0	0	1
Total	64035	59616	1343	3076

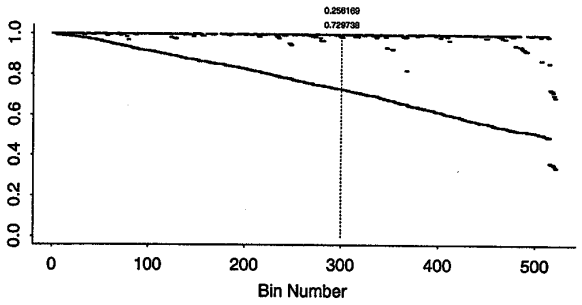

Figure 5. The Fourth FFD Experiment. This is an FFD packing of 1,000 weights. The tops of weights in bins are marked by horizontal bars. The weights in bin number 300 are highlighted.

The hypothesis fails, since it appears that empty space occurs mostly in bins containing two weights, not one weight. Since weights are uniformly distributed on (0, 1], it is not surprising that most bins contains two weights, and that most of those paired bins have gaps in the smallest range. What is perhaps surprising is the large number of paired bins having gaps in the middle and largest range.

The fourth experiment. An obvious next step is to focus on how weights get paired in bins. Figure 5 shows a diagram of every bin and weight in an FFD packing of a list of size n = 1,000. The tops of weights in bins are marked by tiny horizontal bars. For example, bin number 300 is highlighted here with a dotted line. This bin contains a pair of weights, of sizes 0.729738 and 0.256169, giving total weight 0.985907 and a gap of 0.014093. The rough diagonal line across the figure marks the initial packing of the heavy weights in the list. The first (largest) weight goes into bin 1, the second weight (a little bit smaller) goes into bin 2, and so forth. It is not difficult to visualize the rest of the process and to develop a new hypothesis about how weights are paired in bins, but at this point we abandon the study.

Further investigation along these lines is described in [13] and [59]. Interestingly, the observation that overall empty space depends primarily on the quality of packings in bins containing two weights can be extended to other heuristics and other input models. Algorithms that improve the way in which weights get paired in bins would probably outperform these heuristics for these classes of inputs.

It is now known analytically that $\sigma(n) = O(n^{0.5})$ and therefore $\rho(n)$ is asymptotically one for this random model.[38] Related theoretical results and more conjectures about bin packing rules appear in [14], [24], [37], [38], and [72]. An excellent survey of bin packing can be found in [25].

2. Steps In the Process

The bin packing study illustrates an important property of simulation research on algorithms—experimentation is an evolving and developing process. The performance measures, parameters, and design points were all modified as insight grew and new questions were raised. This evolutionary process, alternating hypothesis and experiment, is a crucial factor in successful computational research on algorithms. Also note

that in this (contrived) example, hypotheses were more often contradicted than supported by subsequent experiments. Any sufficiently ambitious computational study on algorithms is likely to contain several false leads and blind alleys.

The experimental process begins well before, and continues well after, the actual computational tests are run. This section surveys methodological issues arising at several stages in this process.

2.1. Planning the Experiments

Computational experiments are undertaken when research questions cannot be answered by direct means. These open questions suggest which performance measures are interesting and what experimental parameters should be incorporated, but other factors should also guide the planning stage.

What to measure. It is possible to find research questions and opportunities for performance improvements at all points along the instantiation scale described earlier. For example, Goldberg's maximum flow algorithm can have $O(n^2m)$ or $O(n^2m^{1/2})$ worst-case performance, depending on how a basic data structure is implemented.[3] If constant factors are similar, this represents a speedup by a factor of 1,000 on moderate-size problems (say $n = 10^3$, $m = n^2$). Furthermore, the choice of a relabeling heuristic for Goldberg's algorithm can affect average-case performance by a factor of about 100 for this problem size.[3] For some classes of inputs, a parallel implementation on a Connection Machine is faster by factors of 40 to 50 than a sequential implementation on a Sun SPARC-2 workstation (for other input classes the SPARC-2 is faster).[2]

Performance measures can be suggested by the open questions motivating the research. To study big-oh performance, count the dominant operation identified in theoretical analyses. To compare strategies for data structures, use combinatorial measures—perhaps number of structure updates or number of nodes touched or both—that are affected by the strategies and that are meaningful indicators of overall performance. An excellent discussion of techniques for finding combinatorial measures that are highly correlated with program running times is given by Ahuja, Magnanti, and Orlin.[1] The authors provide guidelines for discovering bottleneck operations that drive real performance (as opposed to the dominant operations suggested by theoretical analysis), and they illustrate their technique using several algorithms for network flow problems.

For highly instantiated algorithms, it can be difficult to find combinatorial measures that predict running times to a sufficiently fine degree, and it may be tempting just to report the running times. However, this approach severely limits the generality of the experimental results. It is very difficult to scale running times for different computing environments. For example, Fredman et al.[39] observe very different performance profiles for identical programs and inputs depending on whether they are run on RISC or CISC architectures. This situation is exacerbated with the recent growth in variety of parallel architectures and operating systems.

Maybe someday we can predict running times reliably using linear combinations of a small set of key operations, much as Knuth does for the hypothetical, very simple, MIX architecture.[53] For now, however, it remains an open research problem to find a concise and accurate prediction scheme that spans different varieties of computing systems.

The set of combinatorial measures that might be adopted in a given computational study can be quite large. Some measures are better than others in terms of the statistical analyses and insights they support. For example, in the bin packing experiments, measurements of empty space S generally provided clearer views and stronger conjectures than did measurements of the packing ratio P. While this experience is difficult to generalize, some guidelines for finding good measures are listed below.

- Experts on exploratory data analysis point out (for example [19], [76]) that data representing differences (like empty space S) have different properties than data representing ratios (like packing ratio P). Since the insights gained can be significantly different for these two classes of data, it is useful to look at both kinds.
- In general, good measures exhibit small variance within a design point compared to the amount of variation seen between distinct design points.
- A statistician with expertise in variance reduction techniques (VRTs) might recognize that list weight W is a control variate for bin count B, and therefore empty space, defined by $S = B - W$, might have less variance than B (and possibly less than P). Variance reduction techniques can suggest better measures even when they are used informally as is done here. A nice technical introduction to VRTs appears in [18]. A tutorial description of VRTs applied to an algorithmic problem appears in [61].
- A biased estimator is an output variate having expectation that does not equal the quantity it is supposed to be estimating. In the bin packing study, for example, the sample mean $\bar{P}(n)$ overestimates the expected bin ratio $\rho(n)$ because $R < P$ for all lists. Therefore, $\bar{P}(n)$ is a biased estimator of $\rho(n)$. When an unbiased estimator cannot be computed efficiently (as is the case with $\bar{R}(n)$), or when an unbiased estimator has too much variance, it may be possible to find a biased estimator that provides a good upper or lower bound. See Section 3.2 for further development of this idea.
- Too early summarization of the performance measure should be avoided. The simulation program should report output variates from each trial, rather than, say, just the sample means and standard deviations obtained over t trials. This can be especially important when data points are bimodal or have unusual distributional properties. Mechanistic explanations are developed by looking at the entire data set, not just the summary statistics.
- Of course, the measure can and should change as new hypotheses are formed and understanding grows. In the bin packing example, the trend was from less detail to more detail in the output variates. In other studies a reverse of this trend may be more appropriate.

Modes of experimentation. Computational experiments can be developed to target various kinds of questions about

performance. The bin packing study illustrates three modes of experimentation, described below.

- In a *dependency study* the goal is to discover functional relationships between parameter settings and performance measurements. The first bin packing experiment, for example, concentrated on how P and S vary with problem size n. In this experiment, the focus was on average performance. Deviation from average was of secondary interest, used primarily for checking the quality of the regression fit. The function that results from this type of study may be descriptive, reflecting the best fit to the data found, or it may be mechanistic, incorporating some partial understanding of underlying structure. The first bin packing experiment produced a descriptive formula of $cn^{0.515}$, and subsequent experiments were developed to find mechanisms to explain this observation.
- The second bin packing experiment focused on how packing quality correlates with variates such as list weight W and the number of heavy weights in a list. Notice that together with the additional output variates reported in each trial, the second experiment incorporated fewer design points but more trials per design point. In general, such trade-offs may be necessary to keep the total size of the data set manageable.

  A *robustness study* such as this looks at distributional properties observed over several random trials, rather than average behavior. Robustness studies address such questions as: How much deviation from average is there? What is the range in performance at a given design point? Does the amount of variation depend on the parameters? Are there unusually high or low values in the measurements? How does the median compare with the mean? Is the variation in performance correlated with certain properties of the input or of the algorithm?

  Robustness studies can reveal ways to eliminate algorithmic sensitivities that produce unusually bad performance. Correlations between variates can suggest ways in which performance depends on randomized properties. They can also suggest new parameters for future dependency studies. For example, since the quality of FFD packings appears to be correlated with the number of heavy weights, we might develop a new round of experiments incorporating both n and the number of heavy weights in the list-generation routine.
- The last two experiments were *probing studies*, which involved opening up the simulation program and inserting probes to study components of the performance measures. In the third experiment, empty space was recorded in categories according to weight counts and gap ranges. Because of the greater amount of data reported, this experiment was performed for only one trial. The fourth experiment, showing every weight and bin in a packing, represents the penultimate in the amount of detail reported (the ultimate would be achieved by a real-time animation of the packing). This kind of close-up view can be extremely useful for developing the insights needed for building mechanistic explanations, but it is easy to be overwhelmed by too much detail. Visualization tools can be very useful here.

2.2. The Pilot Study

A good first step is to implement a basic simulation program and input generator based on simple assumptions about the algorithmic model. This *pilot program*, developed separately from the more ambitious simulation program, has several uses.

First, the pilot implementation can be used for a preliminary exploration to determine the scope and dimensions of the larger simulation project. The pilot program can be used to: (1) suggest the number of trials necessary to obtain desired accuracy in estimators; (2) indicate the largest problem size that can be reasonably accommodated; (3) narrow a large collection of potential parameters down to the most important ones; (4) suggest better performance measures; and (5) suggest some initial hypotheses about performance.

The pilot study may generate publishable information about the performance of an algorithm. Especially when an algorithm is new and complex, news about its general properties and characteristics can be immediately useful to other researchers. However, the strongest computational experiments are obtained by deliberate designs that exploit some partial information about the problem. The pilot study produces a wealth of information that can be applied in more ambitious experiments.

Second, a pilot program can be used as backup implementation for validating experimental results. The pilot program can be ported to other computing environments and fitted with alternative random number generators to evaluate the impact of environmental factors on the results. Replication of experiments can play a crucial role in ensuring the integrity of the study (see Section 3.1).

The pilot program should be carefully tested for correctness. Statistical tests of the random number generator may be appropriate. Standard program verification and validation techniques should be applied. Sometimes the computational results can be checked against known analytical formulas (in one case such a check revealed an error in the published theoretical work[60]).

2.3. Developing and Running Experiments

Once the scope and dimension of the project are outlined by the pilot study, the next step is to develop a simulation program and a testing environment. After this, the tasks of running experiments and analyzing data can begin.

Implementing the simulation program. Of course the simulation program should be a correct and accurate representation of the algorithmic model. Some sources of difficulties in achieving this goal are outlined in Section 3.1.

Much can depend upon the computational efficiency of the simulation program. The conclusions obtained in a study can depend dramatically on the size of the largest problem examined (see Section 3.3). This quantity, together with the number of design points and the number of trials per design point, are all constrained by the speed of the simulation program. Furthermore, the amount of exploration and evolution possible in a study depends on program efficiency: whether a given hypothesis is pursued may depend entirely upon whether the experiment takes an hour or a weekend to

run. Bentley[7] describes strategies for writing efficient programs. Examples of *simulation speedups*—ways to exploit the simulation problem to obtain results more efficiently than could be obtained in a straightforward implementation of an algorithm—appear in [9], [59], and [61].

Bentley[11] provides an excellent survey of software development issues arising in algorithmic simulation. With illustrations from experimental studies of binary search trees and heuristics for the TSP, he gives several practical hints and programming techniques to support flexible and rigorous experiments. He mentions several principles for input and output design in this context. For example, output files should be self-describing. That is, they should contain human-readable specifications of the parameter settings producing the data; otherwise the experimenter can quickly lose track of which files correspond to which experiments. Another principle is that output files should be directly readable by whatever statistical package, plotting system, or visualization tool is used to analyze the data. (Since these two principles conflict in my own software environment, my approach is to place all parameters and human-readable comments in the input file, and to link it to an output file containing unannotated data. It is also very helpful to use an online notes program to keep track of experiments.)

Basics of experimental design. Where do you place the design points in a dependency study? Familiarity with the powerful formal methods of experimental design can be helpful in answering this question. Statistics texts such as [17], [40], and [62] provide good introductions to this topic. Strategies for developing experimental designs for computational testing of programs are presented by Lin and Rardin,[57] and Nance, Moose, and Foutz.[63]

These formal methods give rise to several informal rules for choosing design points. A good pilot study can provide the rough understanding needed to apply these rules effectively.

- Suppose there are four parameters in the experiment, and each can be set at three levels (say high, medium, and low). A complete factorial design would require $3^4 = 81$ design points for all possible combinations of the parameter levels. If the design produces too many design points to be tractable, then it becomes necessary either to eliminate some parameters or reduce the number of levels for some parameters. Carefully developed incomplete designs can support rigorous statistical analyses. Introductory statistics textbooks such as Freund[40] or Moore and McCabe[62] contain discussions of these topics.
- Partial understanding of the functional relationships between parameters and performance can be exploited in the experimental design. For example, if measure X is known to be linear in parameter p (when all other parameters are held constant), then two design points (at high and low values of p) are sufficient to determine the slope of the line. If the function is not known to be linear, then three levels of p are enough to tell whether the function is curving up or down. More levels give more detailed views. It is important to note, however, that a linear relationship is not the same as an asymptotically linear relationship. Assumptions that ignore low-order terms in functions can produce misleading results.
- Cyclical and threshold behaviors (showing up as oscillations and step-functions in data) are probably best detected with a design that uses many design points spaced evenly or randomly throughout the range of the parameter settings (with perhaps only a few trials per design point). There seems to be little information available about whether this approach is best in general, although some statistical arguments for stratification in field experiments (see for example [18]) tend to justify the use of fewer design points and more trials per design point.
- A large amount of variance in the data makes it difficult to obtain reliable estimates of means. Variance can be reduced by taking more trials; in general variance is inversely proportional to the square root of the number of trials per design point.
- Some experience suggests that the quality of the experiment is enhanced by taking the largest problem sizes possible (see Section 3.3).
- For several reasons described in Section 2.4, data analysis may begin with a transformation of data values, often by taking logarithms. Transformation may be applicable for parameters that have theoretically unbounded growth (like problem size) or that cause large changes in the performance measurements.

  When logarithmic transformation is appropriate, plan ahead and use design points that appear evenly spaced on a logarithmic scale. These points are obtained by increasing the parameter level by a constant factor rather than a constant increment (for example, doubling the problem size each time as was done in the bin packing experiments).

2.4. Statistics and Data Analysis

The bin packing study suggests the power of graphical techniques for providing insights into algorithmic processes. Certainly many of the observations in that example would have been nearly impossible to obtain from, say, large tables showing sample means and standard deviations.

Several general properties of algorithmic problems combine to suggest that methods of graphical and exploratory data analysis, and of scientific visualization, are most useful. For example, simulation programs tend to be very fast and to generate huge data sets. It is not unusual to obtain hundreds of data points in a few seconds (although experiments requiring several hours for one data point are not unknown). Graphs and visualization techniques provide the only practical means of handling such large quantities of data. Also, we are interested in developing mechanistic explanations of observations, and graphical and visual techniques allow us to see directly the patterns produced by underlying mechanisms. Finally, a common goal in dependency studies is to discover the form of the function that relates input parameters and performance measures, not to make predictions based on *a priori* assumptions about the functional form.

Methods of exploratory data analysis are highly suited to this goal.

The remainder of this section discusses the data analysis techniques applied in the bin packing study of Section 1.3. Additional techniques and technical discussions can be found in books by Atkinson,[4] Chambers et al.,[19] Cleveland,[23] du Toit, Steyn, and Stumpf,[34] Rawlings,[68] Tukey,[76] and Velleman and Hoaglin.[77] Many statistical software packages are available to support the analyses described in this section.

The graphs in Figures 3 and 4 are examples of the classic scatterplot representation of data pairs (x,y). Figure 3 shows data in slices and Figure 4 shows clouds of points. A technique known as jittering reduces the problem of overlapping points in slices by adding a small amount of random noise to each x coordinate.[19] Several graphical methods (such as draftsman displays, coded plots, casement displays, and plot matrices) are available for handling higher-dimensional point sets.[19]

The linear least-squares regression fit in Figure 3.c is a standard technique for investigating (assumed) linear relationships in data. Regression is a powerful tool when formal assumptions about the data set are valid. In this case, we assume that the (x,y) pairs are related by the formula $y = ax + b + \epsilon$, where ϵ is a random variate representing deviation from the mean. The variate ϵ is assumed to be independently and normally distributed, with mean 0 and standard deviation independent of x and y. When these assumptions hold, linear regression produces an estimate of the coefficients a and b together with a reliable probabilistic statement about the quality of that estimate.

It is rarely the case that all these regression assumptions hold for data from algorithmic problems. For example, when an output variate represents some count of basic operations, it is likely to have upper and lower bounds that depend on the parameter level, which is not consistent with the assumption of a normal distribution in the error variate ϵ. Nevertheless, regression analysis can be fairly robust even when some assumptions fail. Furthermore, regression-based techniques can be used in informal ways to guide the search for true functional forms. An important tool for checking the regression assumptions is *residuals analysis*. The residuals from the fit, shown in Figure 3.d, correspond to the vertical distances between each y value and the regression line. Other techniques are available to assess the quality of regression fits and to develop improved regression models. Atkinson,[4] Rawlings,[68] and Chambers et al.[19] provide discussions of these techniques.

If residuals analysis (or a quick look at the scatterplot) indicates that the data do not obey a linear relationship, it may be possible to find a linearizing transformation for which the relationship does hold. The original data set in Figure 3.b undergoes a transformation that takes logarithms of both x and y, with the result shown in Figure 3.c. In general, if the data obey the relationship $y = ax^b\epsilon$, then the transformed data $y' = log(y)$, $x' = log(x)$ obey a linear relationship $y' = bx' + log(a) + log(\epsilon)$. Results of linear regression on this transformed scale can be translated back to the original scale to obtain estimates of a and b. This is one of about a half-dozen well-known transformation techniques (see Rawlings[68] for an excellent discussion). Transformation is also useful for making the sample variance constant across the design points, which can improve some kinds of analysis.

Tukey[76] uses the term *re-expression* to describe this same general idea. He points out that data sets tend to fall into a few broad categories, such as counts and amounts, ratios, and balances. He argues that exploratory analysis of counts and amounts (which have lower bounds of zero and no fixed upper bounds) is usually best done after the data have been transformed by taking logarithms (or sometimes square roots).

The technique of *smoothing* shown in Figure 4 is generally more appropriate for clouds than for slices. Smoothed lines are developed by taking running averages in overlapping groups of data points. Smoothed lines can be used to highlight trends and correlations in data, as is done here. They can also be used to assess the quality of regression fits, or to find symmetries in plots of distributions. Smoothing is described by Chambers et al.,[19] Tukey,[76] and Velleman and Hoaglin.[77]

Figures 2 and 5 are very primitive visualization techniques showing two snapshots of FFD packings. The bin packing research cited at the end of Section 1.3 made use of animations showing FFD packings in progress; it is likely that such visualization tools will be more widely used in the future.

3. Hazards of Algorithm Simulation

A basic premise here is that the quality of an experimental study is maximized when the simulation program represents the algorithmic model exactly.

This property does not always hold in published computational research on algorithms, sometimes because the distinction between application programs and simulation programs is not recognized. For example, three studies of heuristics for self-organizing sequential search present theoretical results for a standard analytical model together with some experimental measurements.[15, 21, 74] In all three studies, the simulation programs incorporate variations from the analytical model. Some modifications are clearly intended to make the experiments more reflective of realistic situations. Others were probably developed to reduce the time required to run the tests (for details, see [61]). However, in none of the cases do the authors mention that the data cannot be used for making precise inferences about the model—some experimental measurements overestimate the theoretical quantity, some underestimate it, and for some it is hard to tell.

Other difficulties associated with developing a simulation program that accurately reflects an algorithmic model are surveyed in this section.

3.1. Implementation Artifacts

There are several ways in which a simulation program can give an inaccurate view of algorithmic properties. Consider the following anecdotes (those with no citations are from my

own experience or are personal communications from colleagues).

- Some intriguing cyclical behavior observed in an algorithm for self-organizing search, which prompted about two days worth of fruitless theoretical research, disappeared when the random number generator was changed.
- In a study to estimate a constant in the Held-Karp lower bound on the optimal tour length for the TSP, D. S. Johnson checked his results using different random number generators. Although the estimates agreed to three decimal places, estimates of the fourth decimal place had 95% confidence intervals that were disjoint. (Numerical analyses indicated that estimation of the fourth decimal place should have been well within the range of his experiments.)
- A backup implementation was developed for the bin packing project cited in Section 1.3. The main body of experiments were run on a VAX/750, and a second program was developed on a Radio Shack TRS-80 Model III computer. At one point in the experiments, the two programs reported different values for empty space when run on identical algorithms and identical (large) input instances. An investigation of this discrepancy revealed that the smaller machine precision on the TRS-80 was causing some small weights to be rounded to zero. Larger problem sizes would have introduced the same problem on the VAX implementation.
- A colleague reports that two separately developed implementations of a heuristic algorithm produced different results on identical input instances. This could have been due to randomization in the algorithm. However, it turned out that some strategies introduced to speed up the code had combined to make one of the implementations diverge from the theoretical model. Several experimental tests had to be redone.
- Gent and Walsh[41] describe the following experience in their study of a randomized algorithm for satisfiability. "We discovered a bug in one of the most frequently called subroutines which biased the picking of a random element from a set. We noticed this bug when we observed very different performance running the same code under two different compilers . . . ." Several experiments had to be redone. They also report that the code with the bug sometimes gave better performance, suggesting that the algorithm might be improved by incorporating some kind of systematic bias in the subroutine.

Note that all of these problems were caught because the researchers were able to compare different versions of their simulation programs. The lesson is clear. Key experiments should be replicated to reduce the possibility of program bugs and implementation-dependent factors producing erroneous results.

Two kinds of replication are suggested. First, check for discrepancies due to pseudo-random number generators. Even the best modern generators can exhibit non-random patterns that interact badly with simulation programs. Second, check for program bugs and numerical precision problems by comparing implementations developed separately and running on different machines. In the latter case, identical input instances, generators, and random number seeds should be used. Of course, like debugging, replication only demonstrates the presence of error and not its absence.

3.2. Intractable Problems

Certainly a large proportion of experimental research on algorithms concerns heuristics for NP-hard problems. Approximation and heuristic algorithms are natural candidates for experimental research because they are of great interest to practitioners and are usually too complex to analyze for any but the simplest of models.

One question often asked about an approximation algorithm is: How well does it perform in comparison to an optimal algorithm? The obvious difficulty with experimental research here is that the performance of an optimal algorithm cannot be efficiently computed (unless $P = NP$). Without the optimal solution cost, it can be hard to interpret experimental results: Is a heuristic solution of 35.78 good or bad? How much effort should we spend trying to reduce it to 33? Ideas for sidestepping (but not surmounting) this problem, gleaned from the literature on approximation algorithms for the TSP, are surveyed in this section.

We begin by establishing some notation. For a given input graph G, let A denote the cost of the TSP tour obtained by some approximation algorithm, and let OPT denote the cost of the optimal tour for that instance. Graphs G are drawn randomly from some class $\mathcal{G}$ according to probability distribution P. The set of input parameters for this class and distribution is denoted by p. We are interested in the performance ratio $R = A/OPT$. This ratio has unknown mean $\rho(p)$ that depends on the parameter set p.

The straightforward experimental approach would be to generate t random graphs according to parameters p and to calculate the mean $\bar{R}(p) = (1/t)\Sigma_{i=1}^{t} A_i(p)/OPT_i(p)$. However, this is not possible because $OPT_i(p)$ cannot be computed efficiently. Alternative strategies may involve changing the class $\mathcal{G}$ to other input classes, finding a more tractable analytical measure than $\rho(p)$, and developing new (perhaps biased) estimators of $\rho(p)$.

- Use well-known test sets for which optimal solutions have been published. This corresponds to restricting the input class to a handful of instances. The TSPLIB problems[69] have provided widely used TSP benchmarks for several years, and optimal tours are known for several instances in that set. Of course, it can be difficult to generalize results from small classes. Even worse, researchers may be tempted to tune their algorithms to produce good performance on the benchmark problems rather than on general problem classes. Another difficulty is that testbeds become obsolete. The largest problem in TSPLIB has about 11,000 vertices, while recent TSP heuristics can handle instances with a million vertices ([10], [12]).
- Generate instances for which the optimal solution is known by construction but is somehow hidden from the

heuristics. For these kinds of instances *OPT* can be found exactly. Input instances arising in worst-case analyses often have this property, but interesting random classes are more difficult to construct. Pilcher and Rardin[67] describe random instances of this type for the TSP. Sanchis[70] discusses complexity issues of constructing such test cases for NP-hard problems.

- Restrict the experiments to an instance class and probability distribution for which the expected optimal solution cost $\omega(p) = E[OPT]$ can be calculated. For example, the expected cost of an optimal tour through n points uniformly distributed in the unit square is known to be of the form $K\sqrt{n} + o(n)$ for a constant K. Computational tests have bounded K by $0.765 \leq K \leq 0.765 + 4/n$ (see [6], [42], and [73]).

  With this input model, report the mean $\bar{R}_0(p) = (1/t)\sum_{i=1}^{t} A_i(p)/\omega(p)$, which is an estimator of $\bar{\rho}(p) = E[A_i(p)]/\omega(p)$. In general, there is no reason that $\bar{\rho}(p)$ should be close to $\rho(p)$. That is, the ratio of expected values need not equal the expected value of the ratio. Nevertheless, $\bar{\rho}$ can provide a useful baseline for comparison. See [8] for an example of this approach.
- Use the distribution of costs of heuristic solutions $A_i(p)$, $B_i(p)$ . . . to estimate $OPT_i(p)$. Golden and Stewart[42] show that costs of solutions to certain TSP problems can be reasonably expected to obey a Weibull distribution. They show how to obtain a point estimate and a confidence interval for the minimum value (i.e., the optimal tour cost) of a set of Weibull-distributed data. Their results are impressive. For a class of problems with known optimal solutions their point estimates are always within 2.8% of the optimal cost, and their $(100 - \epsilon)$% confidence intervals always contain the optimal cost (where ϵ is approximately 10^{-22}). Golden and Stewart also apply this technique to estimate $\omega(p)$ for the model using n points uniform in the unit square. They report that an $(100 - \epsilon)$% confidence interval (with $\epsilon \approx 10^{-14}$) always includes $0.765\sqrt{n}$ (which agrees with the computational estimate of K mentioned earlier). This idea is intriguing but has seen limited application.
- Instead of restricting the instance class, find an easily computed lower bound $L_i(p) \leq OPT_i(p)$ and compute the ratio $R_{L,i}(p) = A_i(p)/L_i(p)$. The sample mean $\bar{R}_L(p)$ is an upper bound on $\bar{R}$. Although $\bar{R}_L(p)$ overestimates $\rho(p)$ to a degree that may be unknown, it has the advantage over $\bar{R}(p)$ of being efficient to compute. For applications of this approach to studies of TSP algorithms see [12], [28], and [39]. (This idea was also applied in the bin packing example.)
- Karp[50] uses a novel approach in his evaluation of a TSP heuristic for planar point sets. He applies the heuristic to the problem of computing a minimum spanning tree (MST) of the point set, and compares the cost of the heuristic MST to the cost of the true MST (which is easy to compute). He argues that the performance ratio on the MST problem is predictive of the performance ratio on the TSP problem for these inputs. It would be interesting to see if this approach can be generalized to other problems.
- A final strategy is to forget about performance ratios and just use solution quality $A_i(p)$ to estimate $\alpha(p) = E[A_i(p)]$ for a variety input classes. Use the same test sets that others have used in published work, so that comparison to a broad range of other heuristics is possible. Graphical testbeds other than TSPLIB and Netlib ([33], [69]) are available. Bentley[10, 12] has developed a set of scalable geometric problems that appear to be hard for many TSP algorithms. Several graph and network generators are available in a directory supported by DIMACS.[47] Knuth's extensive Stanford GraphBase[55] contains random graphs and networks as well as several instance classes that have more structure.

Many of these approaches can be applied to problem domains other than the TSP, but it is difficult to say which is the most useful in general. Strategies involving restrictions of the input class may give results that cannot be generalized. The lower-bound strategy is attractive because there is no need to restrict the input class, but good lower bounds on optimal solutions may not always be available. Results can be hard to interpret if the lower bound L is far from *OPT* or is negatively correlated with *OPT*. Some of the more advanced ideas are not yet well understood.

Perhaps the best way to proceed is to search for better estimators, tractable performance measures, and interesting problem classes, when beginning any new study of algorithms for intractable problems. It is also a good idea to produce results that can be compared with previous work.

3.3. Coping with Infinity

Analytical models are developed for infinite-size classes of inputs, and the usual focus is on describing performance as problem size approaches infinity. Unfortunately simulation experiments are finite. With rare exceptions (see [35]), it is not feasible to design experiments that directly reveal asymptotic behavior or that span the entire spectrum of input classes.

It is dangerous to extend observations about performance to situations outside the range of experiments. The hazards of extrapolating to other instance classes are fairly well recognized, and researchers tend to be conservative. However, the dangers of extrapolating to larger problems sizes may be less widely recognized. Three cautionary tales are presented here.

Random binary search trees. The internal path length (IPL) of a binary search tree is a measure of how balanced the tree is (good trees are well balanced and have low IPLs). Start with a random binary search tree of n nodes. What happens to the IPL after m iterations of the operations (delete a random node, insert a random node)? Knuth[54] (Section 6.6.2), citing experimental results by Knott, remarks, "empirical evidence suggests strongly that the path length tends to decrease after repeated deletions and insertions, so the departure from randomness seems to be in the right direction." Later experiments by Eppinger,[36] however, generated a conjecture that the IPL tends to increase to about $\Theta(\log^2 n)$ times the initial cost.

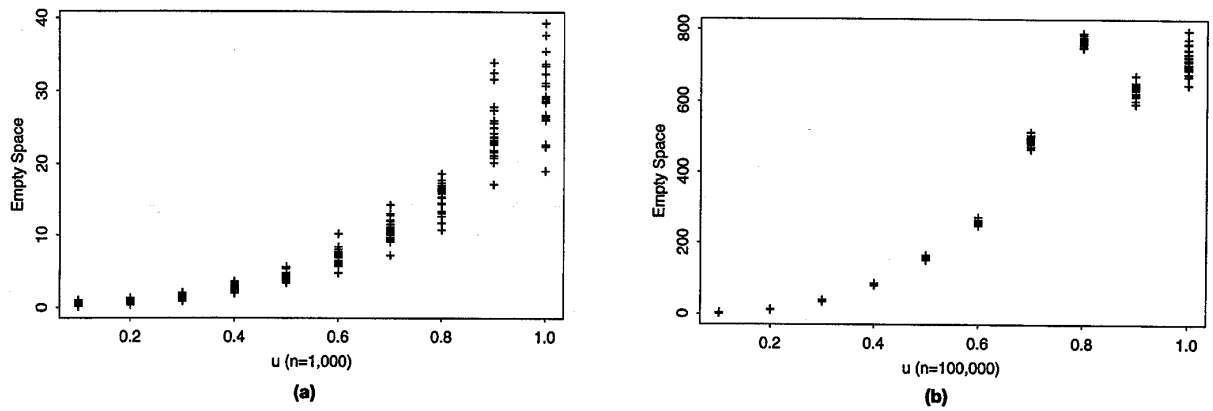

Figure 6. Two Experiments for the First Fit Rule. The parameter u is an upper bound on weight sizes in the list. Graph (a) shows u vs. empty space for n = 1,000. Graph (b) plots u vs. empty space for n = 128,000.

The reason for these contradictory conclusions is problem size. Eppinger's experiments use much larger values for n and m, and his results clearly indicate that the IPL initially decreases, but later increases as m grows. (Later results by Culberson and Munro[29] confirm Eppinger's observations and suggest that the asymptotic IPL is closer to $\Theta(n^{3/2})$. Their results are based partly on experiments and partly on new analytical insights.)

More bin packing. The First Fit heuristic for bin packing is the FFD rule without the sorting step. This rule takes the original weight list and packs each weight into the first bin that can contain it. Suppose the n weights are drawn uniformly from $(0, u]$, for $0 < u \leq 1$. How does empty space vary with u when n is fixed?

Figure 6.a shows observed empty space for 25 random trials at each of several values of u, with n = 1,000. Figure 6.b shows the same measurements with n = 100,000. The non-monotonic behavior is striking. First Fit packings of weights drawn from (0, .9] leave much less empty space than packings of weights from (0, .8].

Experiments on bin packing rules done in the late 1970's ([46], [64]) produced conjectures implying that empty space is linear in u. (Those experiments did not measure empty space, but inferences can be drawn from the published data.) These conjectures were contradicted by experiments done in the late 1980's with n as large as 128,000. Again, a major reason for the contradictory conclusions was problem size. The pattern in Figure 6.b is not visible until n is near 8,000.

Self-organizing search. Another example arises in studies of heuristic rules for self-organizing sequential search. Two input parameters for these problems are n and m. A paper[74] appearing in 1978 describes experiments comparing seven rules with n set to values between 25 and 250, and with m = 12,000. A paper[21] published in 1989 presents experimental results with n at 100 and 200, and with several settings of m up to 600,000. The conclusions and the relative performance rankings of the seven rules are very different in these two papers.

How big is a big problem? These examples clearly indicate the difficulties inherent in extending experimental results to conjectures about asymptotic performance. In all three of these problem domains, the later experiments were performed at least 10 years after the early ones. Experiments using larger problem sizes were possible because of technological improvements in the intervening 10 years. In each case, larger problem sizes in the later studies produced results that contradicted conclusions drawn from earlier studies. Note, however, that the early experiments were worth doing. All of the studies, old and new, produced useful results that could not have been obtained by purely analytical methods.

Recent progress on TSP algorithms has been even more rapid. In 1985, Golden and Stewart[42] described experiments on graphs having at most 350 nodes. In 1987, Bland and Shallcross[16] reported results for problems of 5,000 to 30,000 nodes. In 1990, Bentley[10] described experiments on graphs having up to 1,000,000 nodes. Here problem sizes have increased by four orders of magnitude (seven, if you count the edges) in the space of five years.

In general, it does appear to be worthwhile to make every effort to study the largest problem sizes possible. Even if no better understanding of asymptotic behavior is gained, results on very large problem instances will extend the useful lifetime of an experimental paper.

4. Final Remarks

A final story. While at a conference a few years ago, I was introduced to a colleague who asked me about my research interests. I told him I was working on developing experimental methods for algorithm analysis. "How can you do research in that area?" he asked. "All you have to do is implement the algorithms and measure them—doing computational studies on algorithms is too easy to be called research." About a half-hour later I had a similar conversation with another colleague. "How can you do research in that area?" he asked, "You can't learn anything useful about

algorithms that way—doing computational studies on algorithms is too hard."

Both were right, of course. It is conceptually easy to develop a simulation program that represents a given well-specified algorithm. On the other hand, it is difficult to obtain simulation results that will suggest new mechanistic models of performance and support extrapolations to scenarios outside the scope of the experiments. These goals are not easy to achieve, but with care the special features of algorithmic problems can be exploited and the special hazards of algorithmic research can be circumvented.

Although the potential power of simulation research on algorithmic problems is evident, many difficulties and methodological issues remain unresolved. The ideas presented in this feature article are meant to promote better experiments as well as more discussion as this field continues to develop.

Acknowledgments

I thank Jon Bentley, Bill Eddy, David Johnson, and Jim Orlin, who contributed many anecdotes and suggestions. The ideas presented here were developed through conversations with them and numerous other researchers over too many cups of coffee.

References

1. R.K. AHUJA, T.L. MAGNANTI and J.B. ORLIN, 1993. *Network Flows: Theory, Algorithms, and Applications*, Prentice Hall, Englewood Cliffs, NJ.
2. F. ALIZADEH and A.V. GOLDBERG, 1991. Implementing the Push-Relable Method for the Maximum Flow Problem on a Connection Machine, in *Network Flows and Matching: First DIMACS Implementation Challenge*, D.S. Johnson and C.C. McGeoch (eds), American Mathematical Society, Philadelphia, pp. 65–95.
3. R.J. ANDERSON and J.C. SETUBAL, 1991. Goldberg's Algorithm for Maximum Flow in Perspective: A Computational Study, in *Network Flows and Matching: First DIMACS Implementation Challenge*, D.S. Johnson and C.C. McGeoch, (eds), American Mathematical Society, Philadelphia, pp. 1–18.
4. A.C. ATKINSON, 1987. *Plots, Transformations, and Regression*, Clarendon Press, Oxford.
5. R.R. BARTON, 1987. Testing Strategies for Simulation Optimization, in *Proceedings of the 1987 Winter Simulation Conference*, A. Thesen, H. Grant, and W.D. Kelton, (eds.), Society for Computer Simulation, pp. 391–401.
6. J. BEARDWOOD, J.H. HALTON and J.M. HAMMERSLEY, 1959. The Shortest Path Through Many Points, *Proceedings of the Cambridge Philosophical Society 55*, 299–327.
7. J.L. BENTLEY, 1982. *Writing Efficient Programs*, Prentice Hall, Englewood Cliffs, NJ.
8. J.L. BENTLEY, 1984. A Case Study in Applied Algorithm Design, *Computer 17:2*, 75–88.
9. J.L. BENTLEY, 1988. *More Programming Pearls: Confessions of a Coder*, Addison-Wesley, Reading, MA.
10. J.L. BENTLEY, 1990. Experiments on Traveling Salesman Heuristics, in *Proceedings of the First ACM-SIAM Symposium on Discrete Algorithms*, ACM-SIAM, New York and Philadelphia, pp. 91–99.
11. J.L. BENTLEY, 1991. Tools For Experiments on Algorithms, in *CMU Computer Science: A 25th Anniversary Commemorative*, R.F. Rashid, (ed.), ACM Press, New York.
12. J.L. BENTLEY, 1992. Fast Algorithms for Geometric Traveling Salesman Problems, *ORSA Journal on Computing, 4:4*, 387–411.
13. J.L. BENTLEY, D.S. JOHNSON, F.T. LEIGHTON and C.C. MCGEOCH, 1983. An Experimental Study of Bin Packing, in *Proceedings of the 21st Annual Allerton Conference on Communication, Control, and Computing*, University of Illinois at Urbana-Champaign, pp. 51–60.
14. J.L. BENTLEY, D.S. JOHNSON, F.T. LEIGHTON, C.C. MCGEOCH and L.A. MCGEOCH, 1984. Some Unexpected Expected Behavior Results for Bin Packing, in *Proceedings of the Sixteenth Symposium on Theory of Computing*, ACM, New York, pp. 279–288.
15. J.R. BITNER, 1979. Heuristics that Dynamically Organize Data Structures, *SIAM Journal of Computing 8:1*, 82–110.
16. R.G. BLAND and D.F. SHALLCROSS, 1987. Large Traveling Salesman Problems Arising from Experiments in X-Ray Crystallography: A Preliminary Report on Computation. Technical Report 730, School of Operations Research and Industrial Engineering, Cornell University, Ithaca, NY.
17. G.E.P. BOX, W.G. HUNTER and J.S. HUNTER, 1978. *Statistics for Experimenters*, John Wiley & Sons, New York.
18. P. BRATLEY, B.L. FOX and L.E. SCHRAGE, 1983. *A Guide to Simulation*, Springer-Verlag, New York.
19. J.M. CHAMBERS, W.S. CLEVELAND, B. KLEINER and P.A. TUKEY, 1983. *Graphical Methods for Data Analysis*, Duxbury Press, Boston.
20. B.V. CHERKASSKY, A.V. GOLDBERG and T. RADZIK, 1994. Shortest Paths Algorithms: Theory and Experimental Evaluation, in *Proceedings of the Fifth ACM-SIAM Symposium on Discrete Algorithms*, ACM-SIAM, New York and Philadelphia, pp. 516–525.
21. H.T. CH'NG, B. SRINIVASAN and B.C. OOI, 1989. Study of Self Organizing Heuristics for Skewed Access Patterns, *Information Processing Letters 30*, 237–244.
22. W.S. CLEVELAND, 1985. *Elements of Graphing Data*, Wadsworth, Monteray, CA.
23. W.S. CLEVELAND, 1993. *Visualizing Data*, Hobart Press, Summit, NJ.
24. E.G. COFFMAN, JR., D.S. JOHNSON, P.W. SHOR and R.R. WEBER, 1993. Markov Chains, Computer Proofs, and Average-Case Analysis of Best Fit Bin Packing, in *Proceedings of the Twenty-Fifth Symposium on the Theory of Computing*, ACM, New York, pp. 412–421.
25. E.G. COFFMAN, JR. and G.S. LUEKER, 1991. *Probabilistic Analysis of Packing and Partitioning Algorithms*, Wiley InterScience, New York.
26. T.M. CORMEN, C.E. LEISERSON and R.L. RIVEST, 1990. *Introduction to Algorithms*, MIT Press, Cambridge, MA.
27. H.P. CROWDER, R.S. DEMBO and J.M. MULVEY, 1978. Reporting Computational Experiments in Mathematical Programming, *Mathematical Programming 15*, 316–329.
28. H.P. CROWDER and M.W. PADBERG, 1980. Solving Large-Scale Symmetric Travelling Salesman Problems to Optimality, *Management Science 26:5*, 495–509.
29. J. CULBERSON and J.I. MUNRO, 1989. Explaining the Behavior of Binary Search Trees Under Prolonged Updates: A Model and Simulations, *Computer Journal 32:1*, 68–75.
30. N. DEAN and G. SHANNON, 1992. DIMACS Workshop on Computational Support for Discrete Mathematics, DIMACS Center, Rutgers University, Piscataway, NJ.
31. M.H. DEGROOT, 1975. *Probability and Statistics*, Addison-Wesley, Reading, MA.
32. L. DEVROYE, 1986. *Non-Uniform Random Variate Generation*, Springer-Verlag, New York.
33. J.J. DONGARRA and E. GROSSE, 1987. Distribution of Mathematical Software Via Electronic Mail, *Communications of the ACM 30:5*, 403–407. Available through the mail server netlib@research.att.com.
34. S.H.C. DU TOIT, A.G.W. STEYN and R.H. STUMPF, 1986. *Graphical Exploratory Data Analysis*, Springer-Verlag, New York.

35. W.F. EDDY and A.A. MCINTOSH, 1989. Determining Properties of Minimial Spanning Trees by Local Sampling, in *Computer Science and Statistics: Proceedings of the 20th Symposium on the Interface*, pp. 538–545.
36. J. EPPINGER, 1983. An Empirical Study of Insertion and Deletion in Binary Trees, *Communications of the ACM 26:9*, 663–669.
37. S. FLOYD and R.M. KARP, 1991. FFD Bin Packing for Item Sizes with Distributions on [0,1/2], *Algorithmica 6*, 222–240.
38. G.N. FREDERICKSON, 1980. Probabilistic Analysis for Simple One and Two-Dimensional Bin Packing Algorithms, *Information Processing Letters 11:4-5*, 156–161.
39. M.L. FREDMAN, D.S. JOHNSON, L.A. MCGEOCH and G. OSTHEIMER, 1993. Data Structures for Traveling Salesmen, in *Proceedings of the Fourth ACM-SIAM Symposium on Discrete Algorithms*, ACM-SIAM, New York and Philadelphia, pp. 145–154.
40. J.E. FREUND, 1973. *Modern Elementary Statistics*, Prentice Hall, Englewood Cliffs, NJ.
41. I.P. GENT and T. WALSH, 1994. How not to do it. Manuscript.
42. B.L. GOLDEN and W.R. STEWART, 1985. Empirical Analysis of Heuristics, in *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*, E. Lawler, J.K. Lenstra, A. Rinnooy Kan, and D. Shmoys, (eds.), Wiley InterScience, New York.
43. H. GREENBERG, 1990. Computational Testing: Why, How, and How Much, *ORSA Journal on Computing 2*, 94–97.
44. J. HOOKER, 1993. Needed: An Empirical Science of Algorithms, *Operations Research 42:2*, 201–212.
45. J. HOOKER, 1993. Session on Empirical Evaluation of Algorithms, TIMS/ORSA Joint National Meeting, Chicago.
46. D.S. JOHNSON, 1973. Near-Optimal Bin Packing Algorithms, Ph.D. Thesis, Project MAC TR-100, MIT, Cambridge, MA.
47. D.S. JOHNSON and C.C. MCGEOCH (EDS.), 1991. *Network Flows and Matching: First DIMACS Implementation Challenge*, Vol. 12 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, American Mathematical Society, Philadelphia. Programs and test instances are available via anonymous ftp in directory pub/netflow at dimacs.rutgers.edu.
48. D.S. JOHNSON and M. TRICK (EDS.), 1995. *Graph Coloring, Vertex Cover, and Satisfiability: Second DIMACS Implementation Challenge*. In preparation.
49. C.V. JONES, 1994. Visualization and Optimization, *ORSA Journal on Computing 6:3*, 221–257.
50. R.M. KARP, 1977. Probabilistic Analysis of Partitioning Algorithms for the Traveling-Salesman Problem in the Plane, *Mathematics of Operations Research 2:3*, 209–224.
51. W.D. KELTON, 1994. Perspectives on Simulation Research and Practice, *ORSA Journal on Computing 6:4*, 318–328.
52. J. KLEINJNEN and W. VAN GROENENDAAL, 1992. *Simulation: A Statistical Perspective*, John Wiley & Sons, New York.
53. D.E. KNUTH, 1973. *The Art of Computer Programming: Vol. 1, Fundamental Algorithms*, Addison-Wesley, Reading, MA.
54. D.E. KNUTH, 1973. *The Art of Computer Programming: Vol. 3, Sorting and Searching*, Addison-Wesley, Reading, MA.
55. D.E. KNUTH, 1993. *The Stanford GraphBase: A Platform for Combinatorial Computing*, ACM Press and Addison-Wesley, New York and Reading, MA. Available by anonymous ftp at labrea.stanford.edu.
56. P. L'ECUYER, 1994. Uniform Random Number Generation, *Annals of Operations Research 53*, 77–120.
57. B.W. LIN and R.L. RARDIN, 1980. Controlled Experimental Design for Statistical Comparison of Integer Programming Algorithms, *Management Science 25:12*, 1258–1271.
58. B. LISKOV, 1991. Workshop on Research in Experimental Computer Science, Palo Alto, CA.
59. C.C. MCGEOCH, 1986. Experimental Analysis of Algorithms, Ph.D. Thesis, Technical Report CMU-CS-87-124, Department of Computer Science, Carnegie Mellon University, Pittsburgh.
60. C.C. MCGEOCH, 1986. An Experimental Study of Median Selection in Quicksort, in *Proceedings of the 24th Allerton Conference on Communication, Control, and Computing*, University of Illinois at Urbana-Champaign, pp. 19–28.
61. C.C. MCGEOCH, 1992. Analyzing Algorithms by Simulation: Variance Reduction Techniques and Simulation Speedups, *ACM Computing Surveys 245:2*, 195–212.
62. D.S. MOORE and G.P. MCCABE, 1988. *Introduction to the Practice of Statistics*, W.H. Freeman and Co., San Francisco.
63. R.E. NANCE, JR., R.L. MOOSE and R.V. FOUTZ, 1987. A Statistical Technique for Comparing Heuristics: An Example from Capacity Assignment Strategies in Computer Network Design, *Communications of the ACM 30:5*, 430–442.
64. H.L. ONG, M.J. MAGAZINE and T.S. WEE, 1984. Probabilistic Analysis of Bin Packing Heuristics, *Operations Research 32:5*, 983–999.
65. J.B. ORLIN, 1990. Session on Computational Experiments in Network Optimization. ORSA/TIMS Joint National Meeting, Philadelphia.
66. S.K. PARK and K.W. MILLER, 1988. Random Number Generators: Good Ones are Hard to Find, *Communications of the ACM 31:10*, 1192–1201.
67. M.G. PILCHER and R.L. RARDIN, 1987. A Random Cut Generator for Symmetric Travelling Salesman Problems with Known Optimal Solutions. Technical Report CC-87-4, Purdue University, West Lafayette, IN.
68. J.O. RAWLINGS, 1988. *Applied Regression Analysis: A Research Tool*, Wadsworth and Brooks/Cole, Pacific Grove, CA.
69. G. REINELT, 1991. TSPLIB-A Traveling Salesman Problem Library, *ORSA Journal on Computing 3*, 376–384. Available through the mail server netlib@research.att.com.
70. L.A. SANCHIS, 1990. On the Complexity of Test Case Generation for NP-Hard Problems, *Information Processing Letters 36*, 135–140.
71. R. SEDGEWICK, 1983. *Algorithms*, Addison-Wesley, Reading, MA.
72. P.W. SHOR, 1986. The Average-Case Analysis of Some On-Line Algorithms for Bin Packing, *Combinatorica 6:2*, 179–200.
73. D. STEIN, 1977. Scheduling Dial-a-ride Transportation Systems: An Asymptotic Approach, Ph.D. Thesis, Harvard University, Cambridge, MA.
74. A. TENENBAUM, 1978. Simulations of Dynamic Sequential Search Algorithms, *Communications of the ACM 21:9*, 790–791.
75. M.J. TODD, 1994. Theory and Practice for Interior-Point Methods, *ORSA Journal on Computing 6:1*, 28–31.
76. J.W. TUKEY, 1977. *Exploratory Data Analysis*, Addison-Wesley, Reading, MA.
77. P.F. VELLEMAN and D.C. HOAGLIN, 1981. *Applications, Basics, and Computing of Exploratory Data Analysis*, Duxbury Press, Boston.