Experimental Analysis of Algorithms

Catherine Cole McGeoch

August 1986

Submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Computer Science at Carnegie-Mellon University.

This research was sponsored in part by the National Science Foundation under Graduate Fellowship Grant No. SPE-8350019, and in part by the Office of Naval Research under Contract N00014-85-k-0512.

Carnegie-Mellon University

DEPARTMENT OF COMPUTER SCIENCE

THESIS

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

TITLE Experimental Analysis of Algorithms

PRESENTED BY Catherine McGoech

ACCEPTED

for Louis Bentley — MAJOR PROFESSOR — 25 June 1986 — DATE

Alfred Aberman — DEPARTMENT HEAD — 5/21/87 — DATE

APPROVED

SENIOR VICE PRESIDENT FOR ACADEMIC AFFAIRS — 11 Sept. 1987 — DATE

Abstract

This thesis examines the application of experimental, statistical, and data analysis tools to problems in algorithm analysis. Note that algorithms, not programs, are studied: "results" in algorithm analysis generally refer to abstract cost functions, are independent of particular machines or implementation strategies, and express functional relationships between input parameters and measures of algorithmic performance. The study of algorithms presents special problems and opportunities for experimental research.

The following research goals are set:

1. To demonstrate that simulation can provide a useful, general tool for developing new understanding of algorithms.

2. To identify common problems and to assess the applicability of this approach.

3. To develop principles for successful experimental research in this domain.

4. To promote more general use of this approach by giving a "handbook" of useful tools and techniques.

Part I of the thesis introduces an experimental approach to algorithm analysis and discusses the context for this research. Examples from the current algorithms literature serve to illustrate issues and problems that can arise.

Part II presents experimental results for case studies in four algorithm domains (one-dimensional bin packing, greedy matching, median-selection strategies for Quicksort, and self-organizing sequential search).

Part III discusses the case studies, guidelines for successful simulation research, and useful tools and techniques for simulating algorithms.

Acknowledgements

This is the place where I get to say all those things that I've always meant to say.

To Jon Bentley: as my advisor you have provided encouragement, constructive criticism, and laughs far beyond the call of duty. This thesis is to a great extent a reflection of your abilities as a teacher. Thanks.

To the other members of my committee, Bill Eddy, Jay Kadane, and Ravi Kannan: your patience with my naive questions and your encouragement throughout this research is much appreciated. Thanks.

To the many people that I've learned from in technical discussions, including Al Aho, Andrew Appel, Bill Cleveland, Rex Dwyer, Guy Jacobson, David Johnson, Brian Kernighan, Mike Langston, Tom Leighton, Glen Manacher, Jim Saxe, Danny Sleator, Peter Shor, Mike Steele, Daryl Tennenbaum, Jim Wendorf, Roli Wendorf, Bob Wilber, and many others: thank you for your time and your interest in this research.

It is impossible to complete a graduate degree without the support and encouragement of friends. To all my friends, whether currently, formerly or never members of the CMU CSD community: thanks for the memories.

The support and love of my family has been invaluable to me. To my parents, my grandparents, my inlaws, and my siblings: I couldn't have gotten where I am today without you. Thank you.

I can't say enough about a husband who sympathized with my frustrations, proofread drafts, discussed technical issues, and cleaned the apartment, all while working on his own thesis. You know who you are: thanks, pal.

Table of Contents

PART I:
An Experimental Approach to Algorithm Analysis 1

1. Introduction 3
   - 1.1. A Context for Experiments 3
   - 1.2. Previous Work: Examples 6
   - 1.3. Research Goals 13
   - References 18

PART II:
Case Studies 23

2. One-Dimensional Bin Packing 25
   - 2.1. Previous Work 26
   - 2.2. The Simulation Study 27
   - 2.3. First Fit 28
     - 2.3.1. Nonmonotonicity in u 30
     - 2.3.2. Measurements at fixed u 36
   - 2.4. Best Fit 38
   - 2.5. First Fit Decreasing 40
     - 2.5.1. u Below 0.5 43
     - 2.5.2. u above 0.5 47
   - 2.6. Best Fit Decreasing 51
   - 2.7. Future Work 52
   - References 53
3. Greedy Matching in One Dimension 55
   - 3.1. Introduction 55
   - 3.2. The Study 56
   - 3.3. Experimental Results 58
   - References 65
4. Comparisons in Quicksort 67
   - 4.1. Introduction 67
   - 4.2. Simulation Issues 68
     - 4.2.1. The Model 68
     - 4.2.2. The Simulation Program 72
   - 4.3. Fixed-T Strategies 76
   - 4.4. Choosing T to Minimize Comparisons 81
   - 4.5. Insertion Sort 83

4.6. Conclusions 86
References 87
5. Self-Organizing Search 89
5.1. Previous Work 90
5.2. Measures of Search Rules 92
5.3. Experimental Results 95
5.3.1. Zipf's Distribution 95
5.3.2. Varying Lambda 100
5.4. Properties of Search List Permutations 100
References 107

PART III:
Experiments and Algorithm Analysis 109

6. Experiments and Algorithms 111
6.1. Why Do Experiments? 111
6.2. Applications and Limitations of Experimental Analysis 113
6.3. Principles 117
References 119
7. Tools and Techniques 121
7.1. Choice of Measure 122
7.2. Ensuring Correct Results 123
7.3. Variance Reduction Techniques 124
7.4. Placement of Sample Points 127
7.5. Pilot Studies 130
7.6. Simulation Shortcuts 132
7.7. The Simulation Environment 134
7.8. Analyzing Simulation Results 137
7.8.1. Looking at Distributions 140
7.8.2. Comparing Sets of Data 143
7.8.3. Assessing Functional Relationships 145
7.9. Summary 149
References 151
8. Conclusions 155
8.1. Contributions of the Thesis 156
8.2. Future Work 156
References 162

Part I

An Experimental Approach to Algorithm Analysis

The experiment serves two purposes, often independent one from the other: it allows observation of new facts hitherto either unsuspected, or not yet well defined; and it determines whether a working hypothesis fits the world of observable facts.

– Rene J. Dubois

This section introduces an experimental approach to algorithm analysis and discusses the context for this research. Examples from the current algorithms literature illustrate issues and problems that can arise. The specific goals of this thesis and the research strategy are described.

Chapter 1
Introduction

This thesis investigates the application of experimental methods to problems in algorithm analysis: specifically, techniques of simulation and data analysis are used to gain new understanding of combinatorial algorithms. A number of terms other than "experimental analysis" have been used to describe the same general idea. The term *simulation* is certainly appropriate, since the object is to represent and measure the behavior of one system (an algorithm) by use of another system (a computer). *Monte Carlo study* also applies since inputs are sampled from specified probability distributions. For this thesis, *statistical analysis* was rejected because techniques other than the purely statistical are considered. *Empirical* is defined by Webster as "relying on experience or observation alone often without due regard for system and theory". Both system and theory are highly regarded here.

The following section introduces this research in the context of algorithm analysis and experimental statistics. Section 1.2 surveys experimental studies from the algorithms literature and illustrates problems that can arise. Section 1.3 presents the specific goals and scope of the thesis. The primary vehicle for this research is the case study: simulation results in four algorithm domains are described in Part II. Principles and techniques for powerful, correct, and efficient experimental studies are developed in Part III, which also presents conclusions and open problems.

1.1. A Context for Experiments

Much of the current research in the area of overlap between computer science and statistics involves the development of better tools for statisticians. Conferences such as *Statistics and Computer Science: The Interface*, *COMPSTAT*, and *Frontiers in Computational Statistics* provide forums for research on design of statistical packages, fast and stable algorithms for computing statistical formulas, and database tools for managing and analyzing data sets. Equally important is the use of statistical methods to develop more powerful tools

for computer scientists; a variety of applications are possible. Weide [50], for example, used concepts from probability theory to develop techniques for probabilistically analyzing algorithms. Kadane and Wasilkowski [30] demonstrated the equivalence between Bayesian experimental design and certain complexity problems. Bentley, Haken, and Hon [3] presented a statistical characterization of VLSI designs. Experimental statistical methods have been applied in many areas of computer science: for examples, see Borenstein's investigation of user help systems [11], Brent's evaluation of techniques for dynamic storage allocation [12], or Stritter's study of file migration strategies [48].

Continuing this trend, this thesis examines experimental and statistical techniques in the context of algorithm analysis. As Exhibit 1-1 suggests, there are many ways to analyze an algorithm, and many forms that a "result" may take. The analysis problems considered here are quite traditional: the usual goal is to characterize an algorithm's performance, on an abstract model of computation, as a function of its input. In particular, the expected performance, which is determined by a specified probability distribution on input instances, is the quantity to be characterized. "Performance" generally refers to some measure such as the number of comparisons required, the number of iterations performed, or the quality of a heuristic solution.

Note that algorithms, not programs, are studied here; the principle of *abstraction* is maintained throughout. Abstraction is fundamental to traditional algorithm analysis for a number of good reasons. By maintaining abstraction in the cost function (number of comparisons, say, rather than running time), one obtains results that are implementation-independent and therefore useful in a variety of situations. Abstraction can produce deeper understanding of underlying mechanisms and discovery of algorithmic paradigms. Mathematical models of algorithms and input allow consistent and well-defined manipulation of parameters, provide a standard vocabulary for communicating results, and promote generalization of algorithms and analysis techniques.

Although the algorithmic problems studied are traditional, this research represents something of a departure from familiar uses of simulation in computer science. Because the experimental results are expected to correspond to abstract models and theoretical statements, the procedural issues tackled here are different from those that arise, say, in a benchmark study to compare various compilers. The following tasks usually associated with simulation research are *not* considered here: identifying appropriate benchmarks, monitoring

- Type of Measure.
  - Time: number of significant operations, or running time of a program.
  - Space.
  - Time-space tradeoffs.
  - Heuristic solution quality.
  - Communication cost.
  - Numerical stability.
- Domain of Analysis
  - Worst-case input.
  - Best-case input.
  - Pathological input.
  - Expected-case: probability distribution on input, or randomized algorithm.
  - Input from real applications.
  - Models of typical input.
  - Classic problem instances.
- Model of Computation.
  - Random Access Machines: word or bit operations, straight-line or branching programs.
  - Specific machine models: e.g. MIX implementation.
  - Parallel and distributed models.
  - Real programs and real machines.
- Precision of Analysis.
  - Tractability: establishing polynomial halting time.
  - Order of Magnitude bound: usually asymptotic analysis (as problem size $\rightarrow \infty$).
  - Exact formulas: usually concrete analysis (for all problem sizes).
  - Probabilistic analysis.

Exhibit 1-1: Analyzing Algorithms

and modeling typical input, developing and justifying simulation models (that is, arguing that they are realistic), and developing statistical models for analysis.

Instead, experiments are to be used to study what are essentially mathematical objects: combinatorial algorithms operating on well-defined input distributions. The motivation for applying experimental research in this context is clear: completely analyzing an algorithm is difficult, and purely mathematical approaches don't always give desired results. This is especially true in studies of expected-case performance, where the analytical results that have been obtained are often limited to very simple distributions on input instances. Ideally,

experimental studies could be used to suggest theorems, to support or to refute conjectures, and to characterize performance in terms of input parameters. Success with such an approach depends on understanding of experimental techniques, familiarity with practical issues of algorithm simulation, and knowledge of appropriate analytical tools. Clearer understanding of these topics in the context of algorithm analysis is a primary goal of this research.

The following terminology is used throughout the thesis. Suppose the expected-case performance of algorithm *A* is of interest: this is the *simulation model*. A distribution on input instances is established that can be described by a small number of *parameters*. A *sample point* is determined by a fixed setting of the parameters. A *trial* corresponds to a single input instance randomly generated[1] at a fixed sample point. For example, taking 50 trials at the sample point ($n = 1000$, $p = 1/2$) might correspond to generating 50 binary strings of length 1000 according to a binomial probability distribution with parameter $p = 1/2$. In performing an experiment, a *simulation program* that mimics the performance of algorithm *A* is implemented and applied to the input instances. For each trial, values for one or more *measures* – such as number of comparisons, number of nodes examined, or solution quality – are recorded. The goal of the experimental analysis is usually to characterize the *measurements* (the values taken for the measures) in terms of the input parameters.

Throughout the thesis, the notation H_n refers to the n^{th} harmonic number, defined by $H_n = \sum_{i=1}^{n} 1/i$. The base-2 logarithm of n is denoted by $\lg n$. The natural logarithm is denoted by $\ln n$. The notation "log" is used when the base is irrelevant, as in order-of-magnitude formulas.

1.2. Previous Work: Examples

Given an algorithm whose theoretical analysis is elusive, it is conceptually easy to implement the algorithm, generate appropriate inputs, and gather measurements. In practice, however, difficulties can arise in matching the simulation program to the model, in ensuring correctness of simulation results, and (especially) in using the measurements to gain real insight into the algorithm's structure. To illustrate problems that can arise, this section surveys a number of experimental studies from the algorithms literature. It is clear that none of the studies surveyed here were intended to illustrate sophisticated analysis techniques or

[1] The term "randomly generated" is used as shorthand for "pseudo-randomly generated".

innovations in experimental method: except for a few dissertations entirely devoted to simulation research, the experimental results are usually presented in the final section (or appendix) of a paper largely devoted to theoretical analysis. Although this is not an exhaustive survey, these studies are representative of the current level and scope of experimental research in algorithm analysis.

Simulation Models and Simulation Programs

Usually the goal of an experimental study is to shed light on open problems suggested by partial theoretical characterization of an algorithm. One impediment to achieving this goal is found when simulation results do not correspond to the analytical description of the problem.

This is certainly the case when asymptotic performance is studied: how can measurements at finite input sizes be extended to inferences about asymptotic behavior? Experimental results can be greatly dependent on input size; consider the problem of determining the expected internal path length I_n of a binary tree under a random series of insert/delete operations (see Knuth [34], Section 6.2.2 for a detailed discussion of the problem). Experiments performed by Knott [33] in 1975 suggested that (for certain deletion algorithms) I_n tends to decrease as a random sequence of insertions and deletions is applied. Knott's studies took sequences of up to 24 insertion/deletion operations and trees with fewer than 100 nodes. Eppinger's [21] 1981 study with n as high as 2048 and insert/delete sequences as large as 9,000,000 indicate that I_n decreases at first and then increases as the sequence length grows.

Another common difficulty with obtaining measurements that accurately reflect the simulation model arises in the study of heuristics for NP-hard problems. Bounds on heuristic solution quality are often expressed in terms of the optimal solution, which cannot be determined experimentally. The one-dimensional bin packing problem, for example, is to pack a list of n items with weights from a subrange of (0,1] into unit-sized bins so as to minimize the number of bins used (the *bin count*). Since this problem is NP-hard, heuristic rules for bin packing are of interest; a common analytical measure is the *bin ratio*, the ratio of the heuristic bin count to the optimal bin count.

In general, the true bin ratio cannot be measured experimentally because the optimal bin count is not known (if it were known, there would be no need for a heuristic). A common solution is to find an easily-measured lower bound on the optimal bin count and to estimate

the bin ratio using this lower bound, which gives an upper bound on the true bin ratio. For example, since there must be enough bins to contain all of the items, the *sum of the weights* is a lower bound on the optimal bin count. Also, the optimal bin count is bounded below by the number of items with weight greater than 1/2, since no two of these items can fit in the same bin. Johnson's [29] 1973 simulation study of various packing rules used the measure max(⌈weight-sum⌉, number of items > 1/2) as a lower-bound approximation to the optimal bin count. Ong, Magazine and Wee [38] used ⌈weight sum⌉ as the lower-bound estimate in their 1984 study; noticing that the heuristic "BFD" nearly always achieved the lower bound, they also presented bin ratios using the BFD bin count to estimate the optimal bin count.

Using the solution ratio to characterize the quality of a heuristic is a common analytical technique. One drawback to using a lower-bound estimate of the optimal solution in simulation studies is that the estimate may be a poor approximation to the optimal solution and therefore give little information about the true solution ratio. While this is not a problem in bin packing (Karmarkar [31] showed in 1982 that under the standard expected-case model the weight sum is very near the optimal bin count), finding useful lower bounds is a nontrivial task in many domains. An alternative approach is to generate input instances with known optimal solutions. Helfrich [27], for example, generates random integer lattices with known shortest vectors and uses them to study heuristics for finding the smallest vector in a lattice. Pilcher [39] describes techniques for generating graphs for which optimum traveling salesman tours are known. Although this approach is promising, it can be difficult to develop generation schemes that preserve interesting properties of the input.

Other examples of disparity between simulation model and simulation program have appeared; some could have been avoided. For example, a self-organizing sequential search rule maintains a list of items under a sequence of requests, keeping frequently requested items near the front of the list so that the average cost of searching for requested items is low. Since the rules do not know the true request frequencies, they are allowed to reorder the search list according to the requests seen so far.

The usual expected-case model is that the N items in the search list are requested according to a specified probability distribution. The standard analytical model assumes that all initial orderings of the search list are equally likely. In real applications, however, it is more likely that lists are initially empty and that new items are added to the back of the list if not found. Tenenbaum's [45] simulations of search rules used a combination of these two

assumptions: the lists are initially empty, but accumulation of search costs does not begin until the lists are of size N. Since the search lists are in not in random order when the cost accounting begins, the observed convergence properties for the rules do not correspond to the analytical model. (In fact, Rivest [41] showed that one of the rules (MF) has achieved its asymptotic performance by the time the list is of size N). Tenenbaum's discussion of convergence properties for the analytical model (based on his experimental results) is inappropriate.

Franklin [23] presents an algorithm for performing hidden-line elimination. The algorithm is conjectured to run in time linear in N, the number of overlapping circles randomly generated within the unit square. Franklin presents timing statistics to support this conjecture, and notes that the timings are linear except for a slight increase at larger N; he remarks that the observed super-linearity is probably due to increased paging activity. Ohuyad, Iam and Murasoto [37], similarly, use timing statistics to support the conjectured linear running time of their cell-based algorithm for computing Voronoi diagrams and to find optimal program parameters under various input models.

The above authors have a legitimate interest in the running times of their algorithms, which have great practical value. Their discussions of algorithmic bounds and optimal parameter settings would be stronger, however, if they were based on abstract operations: this could be easily accomplished by simple bookkeeping mechanisms embedded into the implementation. Although runtime statistics can give a rough idea of algorithmic time complexity, a number of factors interfere with accurate measurement. Van Wyk, Bentley, and Weinberger [46] and Wendorf [48] observe that timings of a single program can vary by as much as 20% under Unix[2] timing protocols, even when it is the only user process running on the system. On any large operating system, variation due to paging, multiple users, and cacheing can add significant "noise" to the timings. Implementation details and variation in optimization levels can mask the behavior of the underlying algorithm.

Obtaining Correct Results

Even when the implementation accurately reflects the simulation model, it can be difficult to ensure that experimental results are correct. For example, theoretical models are likely to assume properties of real numbers, but experiments are performed on finite-precision machines. Eddy [20] presents a fast convex hull algorithm for planar point sets and measures

[2] Unix is a trademark of AT&T Bell Laboratories.

its performance for five distributions on point sets. He notes that when the convex hull has a large number of vertices, adjacent sides are nearly parallel and roundoff errors significantly affect the measurements.

The general problem of verifying that a program performs as specified is well known. Direct validation of simulation programs is rarely possible: if the measurements can be accurately predicted, there is little need for a simulation study. Exceptions do occur: Bloch, Daniels, and Spector [10] use Markov analysis to characterize their algorithm for maintaining directory information in a distributed system. Because the size of the state space makes direct computation tedious, they use simulation as an efficient way to describe performance over a wide range of sample points. The authors are able to validate their simulation results by spot-checking against the correct formula.

Since most researchers are not as fortunate, an important assurance of experimental integrity is *replication*. Eppinger [21], for example, replicates his experiments for insertion and deletion in binary trees on a secondary system. The two simulation environments differ in machine architecture (a Vax 11/750 vs. a Perq personal workstation), random number generator, implementation strategy, and programming language. The consistency of results between these two environments gives strong assurance that the results are not artifacts of the implementation.

Otherwise, replication by the author appears to be nonexistent, or at least unreported. Usually, however, authors provide enough details so that the reader can duplicate the experiments. Kernighan and Lin [32], and Coffman, Kadota and Shepp [16] present listings of the simulation programs as well as fairly detailed descriptions of the random number generators. Cameron and Thomas [15] discuss significant implementation details, give the code for the random number generator used, and offer to send a list of random number seeds to interested readers. Many authors report the sample points and the number of trials per sample point and only mention implementation details that differ significantly from the model.

Analyzing the Data

The most difficult task of the simulation study is to draw conclusions about the algorithm based on the experimental results. In most of the studies surveyed here, the "analysis" consists of tables (or graphs) displaying average measurements for each sample point, accompanied by an informal discussion of the results. This is the format used, for example, in studies of sequential search rules by Bellow [1], Bitner [8, 9], Rivest [41], and Tenenbaum

[45]. Other examples of this presentation format are found in Bui [14], Kernighan and Lin [32], Crowder and Padberg [18], Culberson [19], and Friedman, Bentley and Finkel [24].

A few instances of more formal data analyses have appeared. Golden and Stewart [25] apply Wilcoxon signed rank tests, Friedman tests, and other statistical tests to compare TSP heuristics. Eddy [20] estimates standard deviation in his study of a convex hull algorithm. Weide [47] presents confidence intervals in his studies of search structures. Hart [26] establishes confidence intervals for his results on insertion in binary search trees and applies hypothesis testing.

In general, extensive statistical analyses appear relatively infrequently in the algorithms literature. This may be because the answers produced by standard analysis techniques appear to be at odds with many questions posed in algorithm analysis. For example, the standard procedure in regression analysis is to assume an underlying functional form describing the relationship between experimental values and to determine the function parameters that best fit the observed relationship. In the study of algorithms, on the other hand, determining or bounding the true functional relationship is often the primary goal of the analysis; in order-of-magnitude analyses the actual parameter values are not a part of the model.

A partial solution is found when strong arguments for a particular function form are available, although difficulties can still arise. Hart [26] uses regression in his study of insertion in binary trees: after n insertions, the average height $H(n)$ of a tree is known to satisfy $H(n)=C\ln n+o(\ln n)$, and may be of the form $H(n)=C_1\ln n+C_2\ln\ln n+o(\ln\ln n)$. It is known that C is in the range [3.634, 4.311], and specific values of 4 and 4.311 have been conjectured. As part of a thorough statistical analysis, Hart performs a least-squares regression using the model $H(n)=C_1\ln n+C_2\ln\ln n$, which gives $C_1=4.4037$ and $C_2=-4.1001$. Although C_1 is nearer to 4.311 than to 4, it is also larger than its known upper bound. This fact as well as standard analysis suggests that the model is not appropriate for this range of input sizes.

Culberson [19] studies internal tree height after a series of insert/delete operations and fits regression curves using the model $E[I_n]=a(n^{1/2})+b$, where I_n denotes the internal height of a tree with n nodes. The correlation coefficients (for unweighted and weighted regressions) are 0.99894 and 0.9956, and the R^2 errors are 0.997 and 0.99124, suggesting that the model gives a very good fit to the data. In accompanying graphs, however, the data points clearly curve

upward relative to the fit. Culberson remarks: "any such [regression] results must be treated with skepticism, unless some theoretical reason can be found to support them."

Ong, Magazine and Wee [38] use regression when studying heuristics for bin packing. Estimating $H(n)$, the bin count for rule H, they show that if certain reasonable assumptions hold, then $E[H(n)]$ is of the form $bn + \psi(n)$, for constant b and $\psi(n) = o(n)$. Therefore $E[H(n)]$ is a nearly linear function of n for large n. The authors apply least-squares regression using the model $E[H(n)] = a + bn$. They note that the correlation coefficient for all experiments is equal to 1, and that the percentage of variation explained by the model is 99.99: by these measures the model provides a very good fit to the data. Although the regression model is well justified and fits the data very well, some of the results are (unavoidably) misleading, primarily because the model is for asymptotic n and the measurements are taken at n between 40 and 1000. For example, they estimate the bin ratio for the "FFD" rule as 1.018, although Leuker [35] had proved that the true bin ratio is asymptotically 1. It was later shown in [5] and [43] that the bin ratio for two other rules is asymptotically 1 although their estimates suggest otherwise.

Authors have used techniques other than statistical summarization and analysis to convey the results of their simulations. Coffman, Kadota and Shepp [16], for example, study a strategy for dynamic storage allocation. They present "snapshots" of memory over time to illustrate their observations. Culberson [19] presents snapshots of binary search trees as insertions/deletions are performed. Brown and Sedgewick [13], and Bentley and Kernighan [6] have developed systems for "animating" algorithms; perhaps in the future these methods will play a larger role in the analysis and presentation of simulation results.

In addition to these *examples* of experimental study in algorithm analysis, some previous work has appeared *about* using experiments in this context. Purdom and Brown [40] devote a chapter of their text, *The Analysis of Algorithms*, to a discussion of probabilistic tools for analysis and a review of Eppinger's work. Golden and Stewart [25] present data analysis tools for benchmark studies of heuristics for the Traveling Salesman Problem. Crowder, Dembo, and Mulvey [17] discuss issues in the presentation of computational experiments in mathematics; they give a critical survey of previous experimental studies and propose a checklist of criteria for reporting computational results. Hoaglin and Andrews [28] also propose guidelines for presentation of computer-based experimental results.

1.3. Research Goals

While there is strong motivation for using experimental techniques in algorithm analysis, it appears that not much progress has been made. Experimental results have been published that contradict known theoretical bounds, lead to erroneous conclusions about algorithmic performance, and do not correspond to the underlying analytical model. Perhaps because formal statistical analyses have not been generally successful in lending new insight, researchers tend to limit their exposition to methods more suitable to benchmark studies: the most common format for presentation of experimental results is a table giving average measurements at various sample points, accompanied by informal discussion of the table entries. Very little discussion of experimental methods and techniques for research in this domain has appeared.

Nevertheless, the thesis motivating this research is that simulation and data analysis can provide a powerful tool for obtaining new insight about combinatorial algorithms. This hypothesis was prompted by experience with a simulation study begun as joint work with J. L. Bentley, D. S. Johnson, and F. T. Leighton (reported in [4]). The object of the study was to measure the expected performance of two heuristics for bin packing. Under the expected-case model, n items with weights drawn from the uniform distribution on $(0,u]$, $0 < u \leq 1$, are to be packed into a minimum number of unit-sized bins. The amount of *empty space* in a packing – the number of bins used minus the sum of the weights – was the measure of packing quality recorded in our simulations of the packing rules *First Fit* (FF) and *First Fit Decreasing* (FFD) (see Chapter 2 for a detailed discussion of the problem). Prior to the study, very little was known about the expected-case behavior of the rules; our goal was to characterize mean empty space as a function of n and u. The following observations were among those reported in [4].

- When $u = 1$, mean empty space in FFD packings is $\approx 0.3n^{1/2}$. Prior to this observation (first noted by Bentley and Faust [2] in 1980) it was widely conjectured that empty space is $\Omega(n)$. Prompted by experimental results, Leuker [35] subsequently proved the $\Theta(n^{1/2})$ bound.

- When $u \leq 0.5$, mean empty space in FFD packings is $O(1)$. This remarkable observation – that empty space does not grow in n when u is small – was subsequently proved in [5]. The proof gives an upper bound of roughly 10^{10} bins; Floyd and Karp [22] have recently reduced this to 10 bins under a slightly different average-case model. Simulation results suggest that the true expectation is nearer to 0.7.

- When $u \leq 0.5$, empty space is less than 1 (and therefore the packing is optimal) in over 75% of FFD packings.

- When $0.5 < u < 1$, mean empty space is $\Theta(n^{1/3})$. This bound was subsequently proved (see [5]).

- There appears to be a critical point u_n such that when u is less than u_n, empty space in FFD packings is very small; above the critical point, empty space is quite large and outliers (corresponding to very bad packings) are observed. The critical point appears to increase slowly as n grows.

- Mean empty space appears to grow linearly in u when u is below the critical point and to increase rapidly in u above the critical point. When $u \leq 0.5$, empty space appears to be constant in u as well as n. (This early observation is modified somewhat in Chapter 2). Theoretically characterizing empty space as a function of u remains an open problem.

- When $u = 1$, mean empty space in FF packings appears to grow approximately as $0.22n^{0.7}$. This observation contradicts previous widely-held intuition, which predicted that empty space would grow at least linearly in n. A bound of $O(n^{4/5}\log n)$ was subsequently proven (see [5]); this was tightened to $O(n^{2/3}\log^{2/3} n)$ and $\Omega(n^{2/3})$ by Shor [42].

- When $u < 1$, empty space in FF packings is nonmonotonic in u; for example, packings of weights drawn from (0, .9] give *less* empty space than packings of weights from (0, .84] or from (0, 1]. This nonmonotonic behavior becomes more pronounced as n grows.

- The nonmonotonicity suggests that empty space grows more rapidly in n when $u = .84$ than when $u = 1$. Experimental results give the tentative conjecture that empty space is linear in n at some values of u.

This experimental study significantly influenced theoretical analysis of the two bin-packing algorithms. First, the simulation results in some cases contradicted previously held conjectures, prompting a redirection of theorem-proving effort. Not only did the experimental results suggest the theorems to be proved, but detailed and varying views of the data as well as animations of the algorithms as they packed were essential to the development of the proof techniques appearing in [5]. The study went beyond simple measurements: new insight into packing structure, new conjectures about the performance of the heuristics, and more efficient heuristics were a direct result of the simulations. The simulation results have "opened up" what had previously been a fairly closed area for expected-case results.

Limited simulation studies by Johnson [29], Maruyama, Chang, and Tang [36], and Ong, Magazine and Wee [38] had appeared previously. Why was this study much richer in

conjectures, insights, and theorems? The following factors probably contributed to the success of our study.

- Larger problem size. We simulated packings with lists of up to 128,000 items, while previous studies used lists of up to 200 and 1000 items. Some of the observations, particularly the nonmonotonicity in FF packings, were not visible at lower n.

- Change of measure. We measured *empty space* rather than the *bin ratio* (the ratio of the number of bins used by the heuristic to the number used in an optimal packing) as had been done previously. This measure allowed a much clearer picture of packing quality; because the ratios are very near 1 and tend to converge slowly in n, the small changes in growth are overwhelmed by the variance in the data. Empty space has much smaller variance relative to its growth in n.

- Departure from benchmark-style reporting of results. Rather than presenting tables of measurements for the packing rules at various sample points, we tried to characterize empty space as a function of n and u. We examined the raw data over all trials, rather than just average measurements, to gain insight into distributional properties. We made extensive use of graphical analysis tools.

- Many of the experiments were replicated on a personal computer, which varied the implementation of the packing heuristics, the type of random number generator, and the machine word size. The consistency of the results between the two environments, combined with program validation and hand-checking of results at small n, gave us confidence in the (often nonintuitive) results.

- Finally, and perhaps most importantly, the study was not finished after a single round of experiments; we *iterated* theoretical and experimental analyses of the heuristics. The two approaches interacted in many ways. Certainly theoretical work was guided by experiments; just as importantly, experimental work was directed by theoretical insight. In some cases new insights suggested shortcuts in the data-gathering process, or eliminated the need to gather new data. Growing insight suggested more precise measures of packing quality. In turn, later experimental results gave more insight and produced more detailed understanding of the algorithms.

The simulation study of bin packing had a significant impact on theoretical analysis by contributing new theorems, new insights, new conjectures, and precise, accurate measurements. In addition, a number of procedural questions were prompted by the success of the study. Can the above principles be generalized to other algorithmic domains? Could we have learned more from the data? Are more powerful data analysis tools available? Could the same information have been gained with less programming and analysis effort? In general, what types of results can be gained from experimental studies of algorithms? What are the limitations of this approach?

An obvious first step in answering such questions is to see what has been accomplished by others. As Section 1.2 suggests, experimental research in this area has not been extensively applied or addressed. Compare this to experimental research in, say, the physical sciences: entire papers are devoted to experimental results, and topics such as the justification of experimental models, experimental design, applications of analysis tools, and issues of graphical presentation are regularly and rigorously discussed.

A vast literature of general simulation and statistical analysis techniques exists. Texts such as *The Art of Scientific Investigation* [7] contain a great deal of collected "lore" of good experimental technique and discussion of the scientific process (which iterates theoretical and experimental analyses). The application of these techniques and tools in the context of algorithm analysis is the topic addressed in the following chapters.

The following research goals are set:

1. To demonstrate that simulation can provide a useful, general tool for developing new understanding of algorithms.

2. To identify common problems and assess the applicability of this approach.

3. To develop principles for successful experimental research in the domain of algorithm analysis.

4. To promote more general use of this approach by giving a "handbook" of useful tools and techniques.

Part II, comprising Chapters 2 through 5, presents four "case studies" of experimental analyses of algorithms. The case study approach is adopted here for a number of reasons. First, the studies allow an accurate assessment of the usefulness of this approach. The problem domains are well known to computer scientists: partial theoretical characterization already exists, and the open problems have been the subject of extensive previous theoretical and experimental research. If the experiments give new insight in these problem domains, then Goal 1 will be established. Second, the problem domains provide realistic testbeds for simulation and analysis techniques. Third, studying a variety of problems allows identification of common problems and useful general techniques. Fourth, the case studies serve as examples of the experimental process, which may be of use to future researchers. Finally, the experimental results themselves contribute to open problems the algorithm domains.

Part III contains a discussion of the case studies. Chapter 6 discusses applications,

principles, and goals of experimental research in the domain of algorithm analysis. Chapter 7 presents techniques and tools that proved useful in the case studies. Chapter 8 assesses the contributions of the thesis and discusses future work.

References

[1] M. E. Bellow.
Performance of Self-Organizing Sequential Search Heuristics under Stochastic Reference Models.
PhD thesis, Department of Statistics, Carnegie-Mellon University, Pittsburgh, PA, November, 1983.

[2] J. Bentley, J. Faust.
Unpublished notes on simulations of FFD.
1980.

[3] J. L. Bentley, D. Haken, R. W. Hon.
Statistics on VLSI Designs.
Technical Report CMU-CS-80-111, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213, April, 1980.

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

[5] J. L. Bentley, D. S. Johnson, F. T. Leighton, C. C. McGeoch, L. A. McGeoch.
Some unexpected expected-behavior results for bin packing.
In *Proceedings, 16th Symposium on Theory of Computation*. ACM, April, 1984.

[6] J. L. Bentley and B. W. Kernighan.
A system for algorithm animation (draft manuscript).
December , 1986.

[7] W. I. B. Beveridge.
The Art of Scientific Investigation.
Vintage Books, New York, 1957.

[8] J. R. Bitner.
Heuristics that Dynamically Alter Data Structures to Reduce Their Access Time.
PhD thesis, University of Illinois, July, 1976.

[9] J. R. Bitner.
Heuristics that dynamically organize data structures.
SIAM Journal of Computing 8(1):82-110, February, 1979.

[10] J. J. Bloch, D. S. Daniels, and A. Z. Spector.
Weighted Voting for Directories: A Comprehensive Study.
Technical Report CMU-CS-84-114, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, April , 1984.

[11] N. S. Borenstein.
The Design and Evaluation of On-line Help Systems.
PhD thesis, Department of Computer Science, Carnegie-Mellon University, April, 1985.

[12] R. P. Brent.
Dynamic Storage Allocation on a Computer with Virtual Memory.
Technical Report CMA-R37-84, Centre for Mathematical Analysis, Australian National University, Canberra ACT 2601, Australia, February, 1984.

[13] M. H. Brown and R. Sedgewick.
Techniques for algorithm animation.
IEEE Software 2(1):28-39, January, 1985.

[14] T. N. Bui.
On Bisecting Random Graphs.
Master's thesis, MIT, January, 1983.

[15] J. Cameron and G. Thomas.
An heuristic graph partitioning and coloring algorithm.
1984.
Draft manuscript.

[16] E. G. Coffman, Jr., T. T. Kadota, and L. A. Shepp.
A stochastic model of fragmentation in dynamic storage allocation.
Manuscript, Bell Laboratories, Murray Hill, NJ 07974, 1983.

[17] H. P. Crower, R. S. Dembo, and J. M. Mulvey.
Reporting computational experiments in mathematical programming.
Mathematical Programming 15:316-329, 1978.

[18] H. Crowder and M. W. Padberg.
Solving large-scale symmetric traveling salesman problems to optimality.
Management Science 26(5):495-509, May, 1980.

[19] J. C. Culberson.
Updating Binary Trees.
Master's thesis, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, March, 1984.

[20] W. F. Eddy.
A new convex hull algorithm for planar sets.
ACM Transactions on Mathematical Software 3(4):398-403, December, 1977.

[21] J. Eppinger.
An empirical study of insertion and deletion in binary trees.
Communications of the ACM 26(9), September, 1983.

[22] S. Floyd and R. Karp.
FFD bin-packing for distributions on [0, 1/2].
In *Proceedings, 27th Symposium on Foundations of Computer Science*. IEEE, October, 1986.

[23] W. R. Franklin.
An exact hidden sphere algorithm that operates in linear time.
Computer Graphics and Image Processing 15:364-379, February, 1981.

[24] J. H. Friedman, J. L. Bentley, R. A. Finkel.
An algorithm for finding best matches in logarithmic expected time.
ACM Transactions on Mathematical Software 3(3):209-226, September, 1977.

[25] B. L. Golden and W. R. Steward.
Chapter 7: Empirical Analysis of Heuristics.
The Travelling Salesman Problem.
1985, Chapter 7.

[26] R. R. Hart.
The Average Height of Binary Search Trees.
Master's thesis, University of California at Irvine, 1983.

[27] B. Helfrich.
Reduktionsalgorithmen fuer Gitterbasen.
Diplomarbeit, Frankfurt, Germany, 1984.

[28] D. C. Hoaglin and D. F. Andrews.
The reporting of computation-based results in statistics.
The American Statistician 29(3):122-126, August, 1975.

[29] D. S. Johnson.
Near-Optimal Bin Packing Algorithms.
PhD thesis, Department of Mathematics, Massachussetts Institute of Technology, Cambridge MA, June, 1973.

[30] J. B. Kadane and G. W. Wasilkowski.
Average case epsilon-complexity in computer science: a Bayesian view.
In *Second Valencia International Meeting on Bayesian Statistics*. September, 1983.

[31] N. Karmarkar.
Probabilistic analysis of some bin-packing algorithms.
In *Proceedings, 23rd Symposium on Foundations of Computer Science*, pages 107-111. IEEE Computer Society, 1982.

[32] B. W. Kernighan and S. Lin.
An efficient heuristic procedure for partitioning graphs.
The Bell System Technical Journal 49(2):291-307, February, 1970.

[33] G. D. Knott.
Deletion in Binary Storage Trees.
PhD thesis, Stanford University, May, 1975.

[34] D. E. Knuth.
The Art of Computer Programming: Volume 3, Sorting and Searching.
Addison-Wesley Publishing Company, Reading, MA, 1973.

[35] G. S. Leuker.
Bin packing with items uniformly distributed over intervals [a,b].
In *Proceedings, 24th Symposium on Foundations of Computer Science*, pages 289-297. IEEE Computer Society, 1983.

[36] K. Maruyama, S. K. Chang, and D. T. Tang.
A general packing algorithm for multidimensional resource requirements.
International Journal of Computer and Information Sciences 6(2):131-149, 1977.

[37] T. Ohya, M. Iri, and K. Murota.
Improvements of the incremental method for the Voronoi diagram with computational comparison of various algorithms.
Journal of the Operations Research Society of Japan 27(4):304-336, December, 1984.

[38] H. L. Ong, M. J. Magazine, T. S. Wee.
Probabilistic analysis of bin packing heuristics.
Operations Research 32(5):983-998, September-October, 1984.

[39] M. G. Pilcher.
Development and Validation of Random Cut Discrete Optimization Test Problem Generators.
PhD thesis, Purdue University, 1985.

[40] C. W. Purdhom and C. Brown.
The Analysis of Algorithms.
Holt, Reinhart & Winston, 1985.

[41] R. Rivest.
On self-organizing sequential search heuristics.
Communications of the ACM 19(2):63-67, February, 1976.

[42] P. W. Shor.
The average-case analysis of some on-line algorithms for bin packing.
In *Proceedings, 25th Symposium on Foundations of Computer Science*, pages 193 200. IEEE, October, 1984.

[43] P. W. Shor.
Average-case Analysis of Some Online Heuristics for Bin Packing.
PhD thesis, Massachussetts Institute of Technology, May, 1985.

[44] E. P. Stritter.
File Migration.
(Ph.D. Thesis) SLAC-200, UC-32, STAN-CS-77-594, Stanford Linear Accelerator Center, Stanford University, Stanford California, January, 1977.
Available from National Technical Informations Service, U S Department of Commerce, 5285 Port Royal Road, Springfield, VA 22161.

[45] A. Tenenbaum.
Simulations of dynamic sequential search algorithms.
CACM 21(9):790-791, September, 1978.

[46] C. J. Van Wyk, J. L. Bentley, and P. J. Weinberger.
Efficiency Considerations for C Programs on a VAX 11/780.
Technical Report CMU-CS-82-134, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, August, 1982.

[47] B. W. Weide.
Statistical Methods in Algorithm Design and Analysis.
PhD thesis, Carnegie-Mellon University, August, 1978.

[48] J. Wendorf.
Unpublished notes.
October, 1985.

Part II

Case Studies

Just the facts, m'am.

– Sgt. Joe Friday

This section presents four studies in experimental analysis of algorithms. Chapter 2 examines four heuristics for the one-dimensional bin packing problem. The solution quality and time complexity of a greedy algorithm for matching are studied in Chapter 3. Chapter 4 compares strategies for selecting partition elements in Quicksort. Finally, Chapter 5 studies a family of self-organizing sequential search rules under various distributions on request probabilities.

Chapter 2
One-Dimensional Bin Packing

The one-dimensional bin packing problem is well known: given a set of items with weights from the interval (0, 1], pack the items into a minimum number of unit-capacity bins. Since the problem is NP-complete, a variety of approximation algorithms have been proposed. The following have received considerable attention.

- **First Fit (FF):** Inspect the bins sequentially and place each item into the first bin that can contain it. Items are packed in the order in which they are presented as input.
- **First Fit Decreasing (FFD):** Sort the items in decreasing order by weight, then apply First Fit to the sorted list.
- **Best Fit (BF):** Place each item in the fullest bin that can contain it; that is, the bin into which it fits most tightly. As with First Fit, the ordering of the input list is preserved.
- **Best Fit Decreasing (BFD):** Sort the items in decreasing order by weight, then apply Best Fit to the sorted list of weights.

All of these algorithms can be implemented to run in $O(n \log n)$ time, where n is the number of items to be packed. Note that FF and BF are on-line algorithms, whereas FFD and BFD require that the entire list be available before packing begins.

For a given list L_n of n weights, the *bin count* $A(L_n)$ – the number of bins used by algorithm A to pack list L_n – is the usual analytical measure of packing quality. The packing rule that minimizes the bin count for any list has the name OPT. The *sum of weights* in L_n, denoted by $\Sigma(L_n)$, is a lower bound on $OPT(L_n)$. Another common measure is the *bin ratio*, the ratio of the number of bins used by the algorithm to the number used by OPT when packing L_n. Since bin packing is NP-hard, $OPT(L_n)$ cannot be easily determined experimentally. The measure *empty space*, which is the sum of the gaps remaining in partially packed bins, is therefore adopted here. Empty space for algorithm A when packing list L_n is denoted by $\Delta A(L_n)$.

Note that empty space is equal to the difference between the bin count and the weight sum: that is, $\Delta A(L_n) = A(L_n) - \Sigma(L_n)$. Since $\Sigma(L_n)$ is a lower bound on $OPT(L_n)$, an upper bound on empty space can be used to derive an upper bound on the bin ratio by the following argument. Suppose it is established that $\Delta A(L_n) = A(L_n) - \Sigma(L_n) \le f(u,n)$, for some function $f(u,n)$. Then

$$(A(L_n) - \Sigma(L_n)) / OPT(L_n) \le f(u,n)/OPT(L_n).$$

and therefore

$$A(L_n)/OPT \le f(u,n)/OPT(L_n) + \Sigma(L_n)/OPT(L_n).$$

where the last term is at most 1.

The expected performance of the bin-packing algorithms is studied in this chapter. Under the standard model, weights are drawn independently at random from the uniform distribution on the interval $[0, u]$, for $0 < u \le 1$. The list $L_{u,n}$ is therefore a random variable generated according to parameters n and u, as are the measures bin count $A(L_{u,n})$, and empty space $\Delta A(L_{u,n})$. Note that $E[\Sigma(L_{u,n})] = un/2$.

2.1. Previous Work

Approximation algorithms for bin packing have received considerable attention; for an extensive review of work in this area, see Coffman, Garey and Johnson [5]. Some results related to this work are surveyed below.

Johnson [9] established the following worst-case bounds on the bin ratio. These bounds are tight in the sense that no better ratio can be found.

$$FFD(L_n) \le 11/9 \cdot OPT(L_n) + 4$$

$$BFD(L_n) \le 11/9 \cdot OPT(L_n) + 4$$

$$FF(L_n) \le 17/10 \cdot OPT(L_n) + 1$$

$$BF(L_n) \le 17/10 \cdot OPT(L_n) + 1$$

Brown [4] showed that no *on-line* algorithm can achieve an asymptotic bin ratio better than 1.536. On the other hand, Fernandez de la Vega and Leuker [6] and Karmarkar and Karp [11] have presented *off-line* algorithms for which the worst-case bin ratio approaches the optimal value of *1*.

Karmarkar [10] showed that in the expected-case model any u allows a *perfect packing*: that is, the ratio of $E[OPT(L_{u,n})]$ to $E[\Sigma(L_{u,n})]$ approaches 1 as $n \to \infty$. How do the heuristics compare to the optimal packing? Expected-case results for the rules studied here have appeared only for the case $u = 1$. Frederickson [8] and Leuker [13] have studied FFD and BFD. Their analyses show that the expected bin ratio for these algorithms converges to 1 and that expected empty space is $\Theta(n^{1/2})$.

More recently, Shor [16] showed that any on-line algorithm that does not know n in advance (including BF and FF) will leave $\Omega(n^{1/2}(\log n)^{1/2})$ empty space in the expected case. He also showed that $E[\Delta BF(L_{1,n})]$ is $O(n^{1/2}\log n)$ and $\Omega(n^{1/2}(\log n)^{1/4})$, and that $E[\Delta FF(L_{1,n})]$ is $O(n^{2/3}(\log n)^{2/3})$ and $\Omega(n^{2/3})$.

Previous simulation studies of bin packing heuristics have estimated the bin ratio for varying n and u. Johnson [9] simulated a number of heuristics for $u = 0.25, 0.5, 1$ and n up to 200. He computed max(⌈weight-sum⌉, number of weights > 1/2) (a lower bound on OPT) for each list, and reported the bin ratio using this approximation. His measurements therefore give an upper bound on bin ratios for $n \le 200$. Ong, Magazine and Wee [15] applied regression analysis with the model $E[A(L_{u,n})] = bn + a$. The regression fits give estimates for the asymptotic bin ratio, using $\Sigma(L_{u,n})$ and $BFD(L_{u,n})$ to estimate $OPT(L_{u,n})$. Their simulations took $u = .25, .5, .75, 1$ and n up to 1000. Maruyama, Chang, and Tang [14] considered a spectrum of packing rules and input parameters to determine which rules dominate under a variety of circumstances.

2.2. The Simulation Study

The following sections present new results for the expected-case behavior of the heuristics. Parts have been published as joint work with J. Bentley, D. Johnson, T. Leighton, and L. McGeoch, in [2] and [3]. Earlier work is extended here in a number of ways, primarily by a closer examination of performance when u is less than 1, identification of new measures that give more precise characterization of the packings, and new arguments to explain observed behavior.

The initial goal of the simulation study was to measure packing quality for the four rules as a function of list size and the upper bound on the item weights. Each trial therefore corresponds to a list of n weights generated independently from the uniform distribution on $(0,u]$. For efficiency, integer computation was used in the simulation: bin capacity was $2^{30}-1$, and weights were generated from appropriate integer ranges.

The primary experiments were performed on a VAX 11/750, (some on a VAX 11/780), using 32-bit integers and 64-bit (55-bit mantissa) double-precision reals. The random number generator was the cyclic feedback method described by Knuth [12] (Algorithm A, Section 3.2.2, 2nd Edition). To check the primary results, a number of experiments were replicated on a TRS-80 Model III computer (using the system's linear congruential random number generator), with 32-bit real arithmetic.

In most of the graphs presented in this chapter, the abscissa corresponds to either n or u and the ordinate corresponds to a measure of packing quality such as empty space (in units of bins). Sample points were taken at n doubling from 125: that is, at 125, 250, . . . 128000. The parameter u takes values in the range (0, 1]. Unless otherwise specified, the number of trials at each sample point is 25.

2.3. First Fit

A long-standing open problem has been to determine the asymptotic bin ratio for First Fit under the expected-case model when u is fixed at 1. Since First Fit is an on-line algorithm and does not reorder its input list, it has been widely conjectured (see [9, 3]) that the heuristic is not asymptotically optimal, implying that the asymptotic bin ratio is some constant strictly greater than 1. Johnson's [9] worst case bound implies that the expected bin ratio cannot be more than 1.7; from simulations he conjectured that the expected bin ratio is near 1.07. Experimental results of Ong, Magazine and Wee [15] (with higher n) suggest that the ratio is between 1.038 and 1.056.

Exhibit 2-1-a shows empty space for 25 trials each at the sample points $u = 1$ and n doubling from 125 to 128000 The near linear growth on a log/log scale suggests a power law: linear least-squares regression on this scale yields an estimated slope of 0.7012, indicating that mean empty space grows approximately as $n^{0.7}$. The residuals to this fit (Graph 2-1-b) suggest that the variance increases in n; no significant curvature in the residuals is apparent.

It appears, therefore, that empty space grows sublinearly in n. This leads to the surprising conjecture that the expected bin ratio of First Fit is asymptotically optimal, by the following argument: the expected sum of the weights (equal to $n/2$) grows linearly in n and therefore $\mathrm{OPT}(L_{1,n})$ is $\Omega(n)$. Since, $\Delta\mathrm{FF}(L_{1,n}) \approx n^{0.7} = o(\mathrm{OPT}(L_{1,n}))$, the asymptotic expected bin ratio is 1.

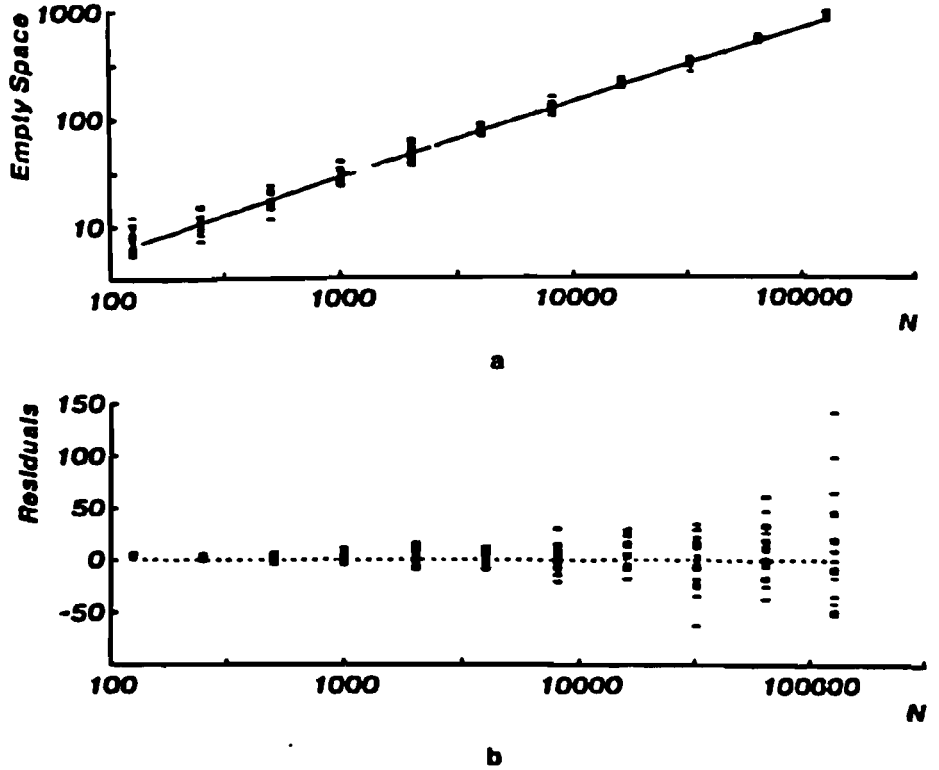


Exhibit 2-1: First Fit, $u = 1$

This new conjecture is proved in [3]; specifically, it is shown that $E[\Delta FF(L_{1,n})] = O(n^{0.8})$. The bound is actually derived for a simplified version of FF, called 2FF, which performs exactly as FF except that a maximum of two items may be placed in any bin. (It is easy to show that FF never uses more bins than 2FF.) An obvious question is whether the gap between observed growth in empty space, $\approx n^{0.7}$, and the theoretical bound, $O(n^{0.8})$, is due to differences between the packing efficiency of 2FF and FF. Graph 2-2 suggests that there is no such gap between the two algorithms: on identical lists, the average ratio of empty space in a 2FF packing to that in the FF packing appears to approach a constant near 1.2 rather than increasing in n.

Prompted by these observations, Shor [16] obtained bounds of $\Omega(n^{2/3})$ and $O(n^{2/3}\log^{2/3} n)$ for expected empty space in FF packings. His technique was to establish an analogy between bin packing and certain planar matching problems and to show that bounds in one domain imply bounds in the other. Note that although our regression fit ($\Delta FF(L_{1,n}) \approx n^{0.7}$) is greater

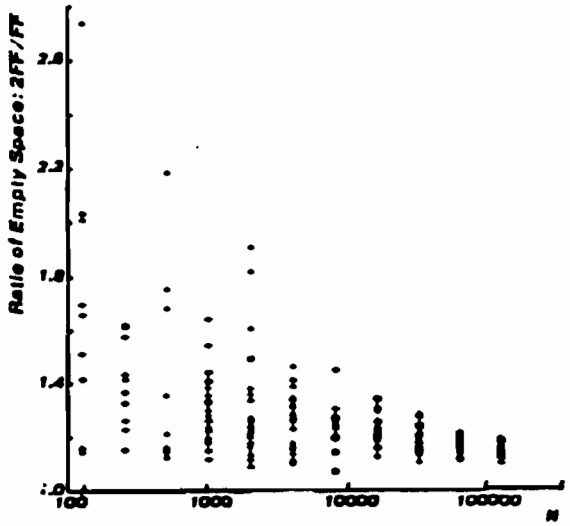


Exhibit 2-2: Ratio of 2FF to FF

than Shor's upper bound, the fit was adequate to justify the conjecture of asymptotic optimality.

2.3.1. Nonmonotonicity in u

Analytical characterizations of First Fit for values of $u < 1$ remain elusive. Graph 2-3-a presents empty space for 25 trials at each sample point, for $0.2 \leq u \leq 1.0$ and $n = 128000$. Graph 2-3-b gives a "closeup" of the measure for $0.7 \leq u \leq 1$. Graph 2-3-c presents mean empty space in this range for four values of n. The graphs reveal a surprising phenomenon: at large n, empty space is not monotonic in u. Packings of items with weights from $(0, 0.8]$, for example, give *more* empty space than packings of items with weights from $(0, 0.9]$.

Experiments taking 50 trials each for u in the neighborhood of the local minima (in increments of 0.005) suggest that they occur near $u^- = 0.9$, 0.925, and 0.94 respectively, for n = 32000, 64000, and 128000. Doubling n produces a declining increment in u^-, suggesting that the local minimum u^- increases slowly in n. It would be interesting to determine if u^- has an asymptotic upper bound that is strictly less than 1.

Similar estimation of u^+, the u value giving the local maximum, is more difficult because of the shallow curve and relatively large variance of data points in this range. With u taken in increments of 0.01, the largest means are found at $u^+ = 0.82$, 0.82, and 0.8 for n = 32000,

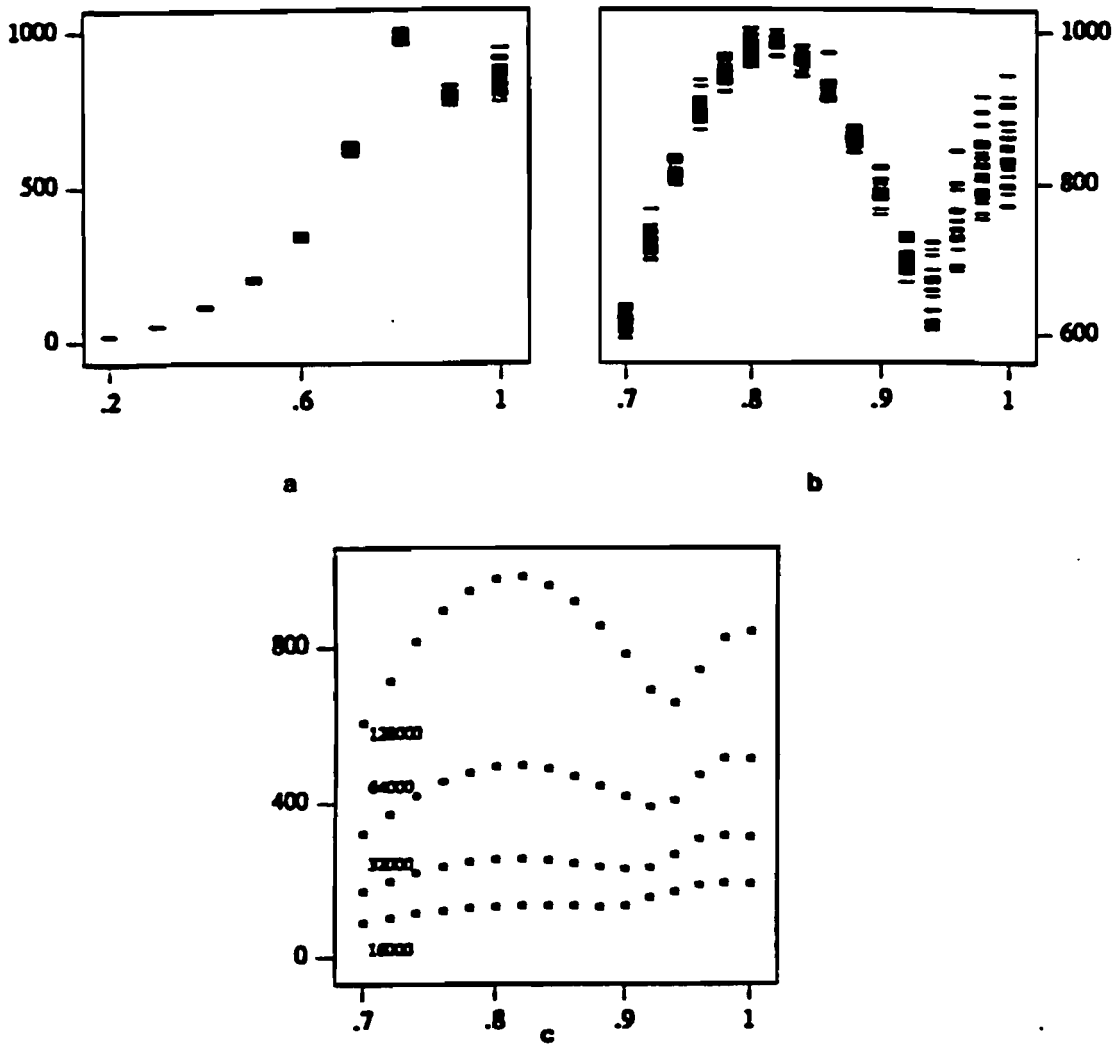


Exhibit 2-3: Empty Space for Varying u

64000, and 128000. These estimates do not inspire confidence because the means do not move smoothly in u at fixed n.

An alternative to comparing means at each sample point is to use a statistical test on each set of measurements. Student's t test, for example, evaluates the hypothesis that two point sets from a normal distribution have the same mean: the measurements at the sample point $n = 128000, u = 0.8$, for example, can be compared to those from $n = 128000, u = 0.81$. (Informal graphical analysis indicates that the assumption of normality is not unreasonable.)

Unfortunately, while the graphical observation that the three curves are moving up at $u = 0.79, 0.80$ and down at $u = 0.83, 0.84$ is confirmed, the test detects no significant difference between adjacent point sets in this range. The conjecture remains, therefore, that $0.8 \leq u^+ \leq 0.82$, and that u^+ moves very slowly, if at all, in n.

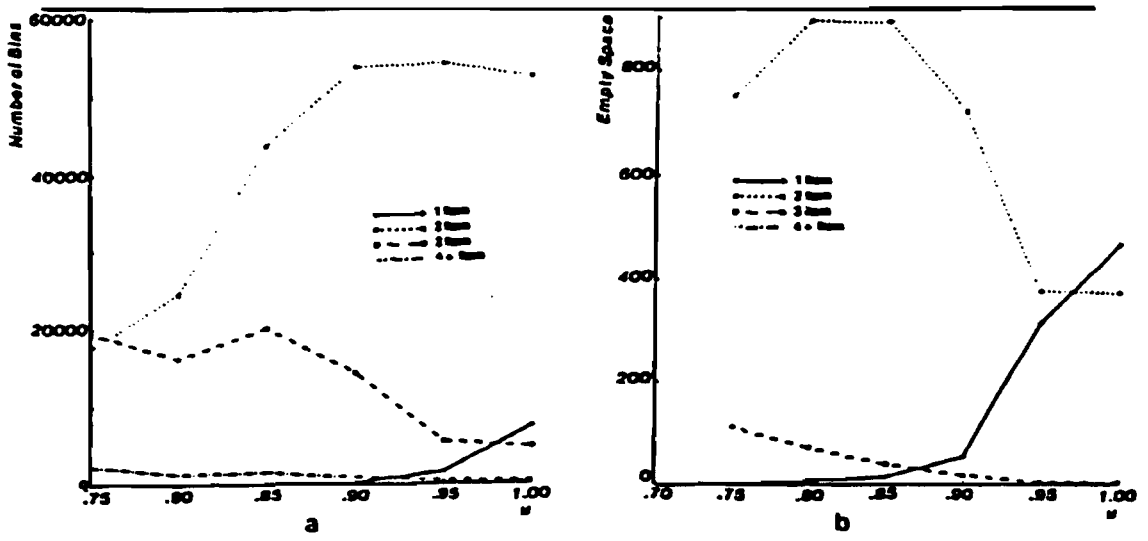


Exhibit 2-4: Distribution by Item Class

A more detailed look at the packings gives a better understanding of the nonmonotic behavior. Define a *gap* to be the amount of empty space in a single bin, and define a *k-item bin* to be a bin with k items in it after the packing is finished. Graph 2-4-a shows the distribution of k-item bins, averaged over five trials each at $n = 128000$ and $0.75 \leq u \leq 1$. The four curves represent the average number of 1-item, 2-item, 3-item, and 4-or-more-item (denoted by 4+ item) bins in the packings. Over most of this range the number of 1-item bins is small, except for a rapid increase as u nears 1. The 2-item bins are by far the most common: the number of 2-item bins increases quickly in the range $0.8 \leq u \leq 0.9$ and then levels off as u nears 1. The number of 3-item and 4+ item bins generally decrease as u increases. On a logarithmic y-scale the number of 1-item bins is nearly linear in u, suggesting approximately exponential growth. Graph 2-4-b depicts the amount of empty space in each bin class. Most of the empty space lies in 2-item bins for u in this range. As u nears 1, empty space in 2-item bins decreases and empty space in 1-item bins increases; compare.

Now, divide the interval (0, 0.25] into 25 *gap ranges*, each of the form $((i-1)/100, i/100]$, for $i \leq 25$, and label each bin according to the index i of its gap range. (Bins with gap greater

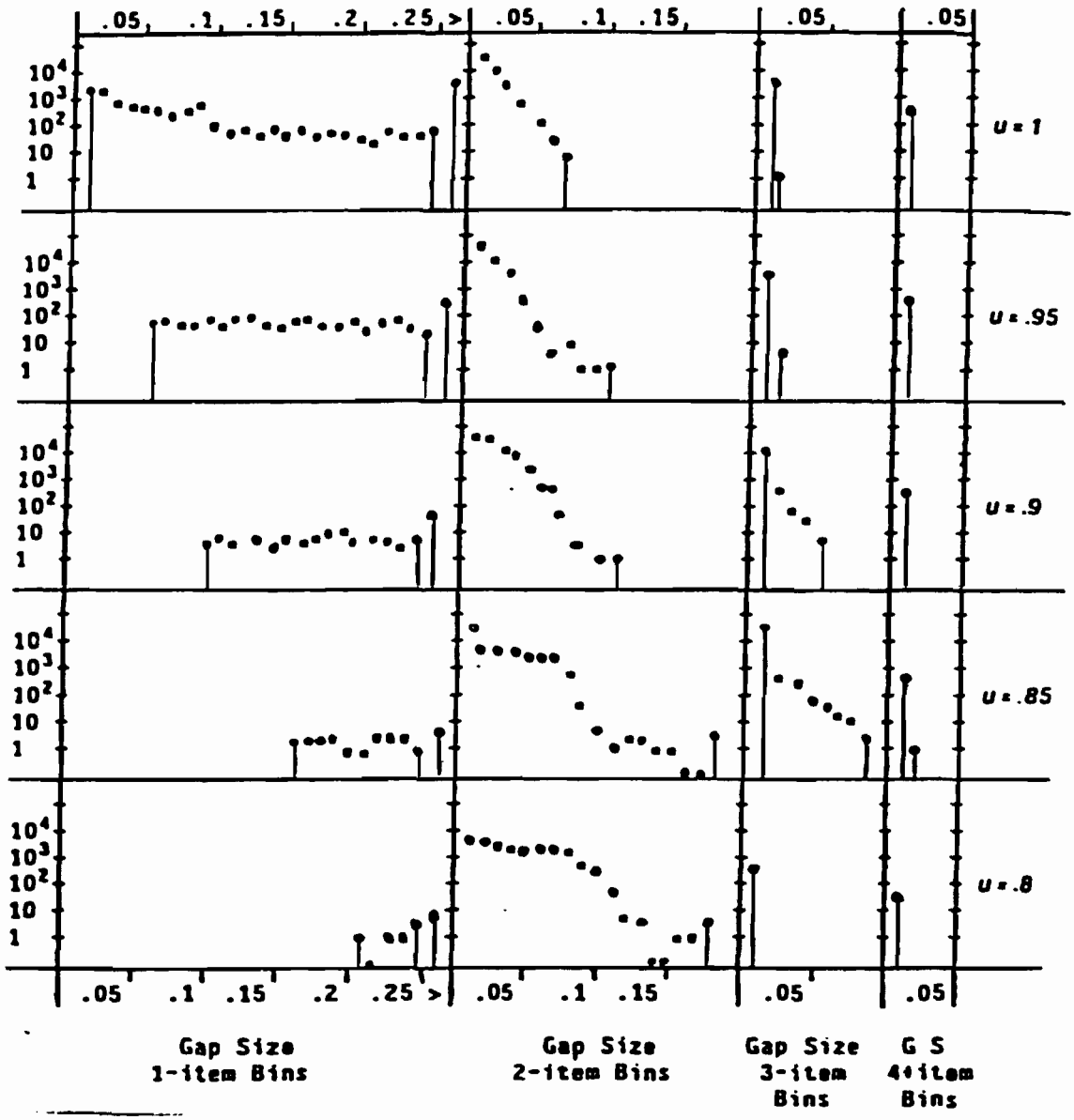


Exhibit 2-5: Distributions of Gaps

than 0.25 are counted in a single category.) Exhibit 2-5 presents the distribution of bins by k-item category as well as by gap range, with $n = 128000$ and five values of u, for a single trial at each sample point. In this multigraph, a *row* of panels corresponds to a u value, and a *column* of panels corresponds to k-item bins, for k = 1, 2, 3, and 4-or-more. Each panel

displays the distribution of bins with respect to gap ranges. In the upper left panel, for example, a point with coordinates (0.05, 400) would indicate that in the packing at $n = 128000$, $u = 1$, there were 400 1-item bins having gaps in the range (.04, .05].

Because the counts range over four orders of magnitude, a logarithmic y-scale is used in the panels. In each panel, the horizontal bars mark the right and left edges of the gap ranges having non-zero counts. In the 1-item column (leftmost panels), bins with gaps larger than 0.25 are counted as a single category at the right side of each panel.

Exhibit 2-5 allows a number of observations. For example, 1-item bins cannot have gaps smaller than $1 - u$. Only 1-item bins ever have gaps larger than 0.25. Gaps in 1-item bins are fairly uniformly distributed, whereas the distribution of gaps in 2-item (and in 3-item bins) declines quickly as gap size increases.

The largest gap observed in 2-item bins is 0.18 (at $u = .85$), and 3-item bins have gaps smaller than 0.09. For some reason, gaps in 3-item bins are generally larger at $u = 0.85$ than at $u = 0.9$ or at $u = 0.8$. The largest gap observed for 4+ item bins is 0.02. In general, at fixed u, k-item bins have smaller gaps than j-item bins, for $k < j$.

Finally, note that as u increases to 1 the *number* of 1-item bins increases (observed from Graph 2-4-a), but the average *gap* in those bins decreases (since the distribution shifts left at higher panels of 2-5). Similarly, the number of 2-item bins also increases in u, leveling off near $u = 0.9$, and the average gap in 2-item bins decreases as u nears 1. Recall that empty space in 2-item bins is nonmonotonic over this range, and that these bins dominate the packing: the 2-item bins account for the "hump" observed earlier for total empty space. As u nears 1, empty space in 1-item bins increases rapidly, which accounts for the rise in total empty space when u is greater than u^-. The number of 3 and 4+ item bins generally decreases as u increases, but they are few and their gaps are small. Although they do not greatly affect total empty space in this range, it is likely, that they would dominate the packings at smaller u.

We conjectured in [2] that there exist values of u for which expected empty space grows linearly in n. Exhibits 2-4 and 2-5 suggests an argument for this conjecture when, say, $u = 0.8$. Suppose there is a positive constant f that gives a lower bound on the expected fraction of 1-item and 2-item bins in an FF packing. Now, the gap in 1-item bins is at least $1-u$; suppose also that the average gap in 2-item bins is at least some fixed constant e, where $0 < e \leq 1-u$. Total empty space is therefore bounded below by $ef\cdot FF(L_{u,n})$. Since the number of bins is at

least linear (bounded below by the sum of the weights) this gives a linear lower bound for empty space.

It is easy to formalize the following argument for the desired lower bound on f, the fraction of 1-item and 2-item bins in the packing. The argument works by giving an upper bound on the number of bins that can contain at least 3 items. This number is maximized when as many bins as possible have exactly three items.

Recall that $\Sigma(L_{u,n})$ is a lower bound on the number of bins used; when $u = 0.8$ we have $E[\Sigma(L_{u,n})] = 2n/5$. Suppose a packing leaves $2n/5$ bins. Not *all* of these bins can contain 3 or more items, since this would give only $n/3$ bins, which is smaller than $2n/5$. The way to leave $2n/5$ bins and also maximize the number of 3-item bins is to pack as many as possible 3-item bins and then to fill out the packing with 1-item bins. If a is the number of items packed 3-to-a-bin, we want to maximize a subject to $a/3 + (n-a) = 2n/5$. This equality holds when $a = 9n/10$; therefore at most $9n/10$ of the items can be packed 3-to-a-bin if there are to be $2n/5$ total bins in the packing. The expected number of items packed 1- or 2-to-a-bin is therefore at least $n/10$.

Deriving a lower bound on e appears to be difficult. An easy lower bound on gap sizes exists for 1-item bins: since u bounds the weight size, there can be no 1-item bin with gap smaller than $1-u$. If the 2-item bins were formed by random pairings of uniform variates with upper bound 0.8, then one could easily show that the gaps are (with high probability) greater than a small constant $e \leq 0.2$. Unfortunately, the pairings produced by First Fit are likely to give gaps consistently smaller than those produced by random pairings.

Graph 2-6-a gives observed average values for f for three trials each at $n = 128000$ and $0.2 < u \leq 1$. The curves corresponds to k-item bins, for k between 1 and 7: each curve is labeled near the point where it reaches its highest value. For example, at $u = 0.65$ the 3-item bins comprise about 61% of the bins in the packings, and this is the highest fraction ever obtained by 3-item bins. The highest fraction achieved for 2-item bins is approximately 81%, seen when $u = .95$. The highest fraction for 1-item bins is about 1%, when $u = 1$.

Graph 2-6-b gives the distribution of empty space in 2-item bins for $n = 128000$, $u = .8$. The x-coordinate of each point corresponds to a gap range in increments of .001. The y-coordinate gives the fraction of 2-item bins with gaps in this range. This graph can be used to suggest appropriate values for e: the leftmost points show that about 3/100 of the 2-item bins

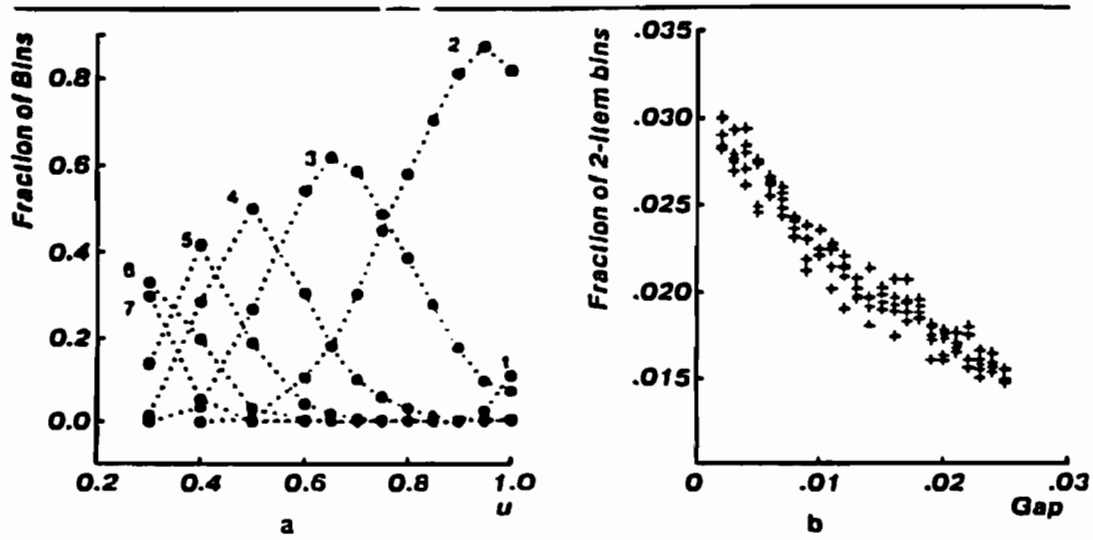


Exhibit 2-6: Measuring f and e

have gaps in the range [0, .001]. Therefore 97/100 of the 2-item bins have gaps at least .001. About half of the bins have gaps greater than .025 (not shown on this graph). Similar graphs measuring gap distribution at smaller n are almost identical in appearance to Graph 2-6-b, although variance tends to decrease in n.

The graphs of Exhibit 2-6 show a great deal of structure; it would be interesting to characterize this behavior analytically by bounding the number of k-item bins and the gaps in those bins for any value of u. These could be combined to obtain bounds on empty space as well as the bin ratio in First Fit packings.

2.3.2. Measurements at fixed u

The previous subsection measures growth in u for fixed values of n; this subsection examines growth in n for certain fixed values of u. An interesting open question is whether empty space is linear in n for any value of u. Early simulation results (reported in [2]) suggest that this is the case when $u = 0.8$, and the previous subsection gives an argument for this conjecture.

Graph 2-7-a depicts mean empty space as a function of n for 25 trials each at four values of u; Graph 2-7-b presents results of 25 trials at $u = .2, .8$. Both graphs are on log-log scales. At $u = .8$, a linear least-squares fit to the data in 2-7-b corresponding to the five highest values of

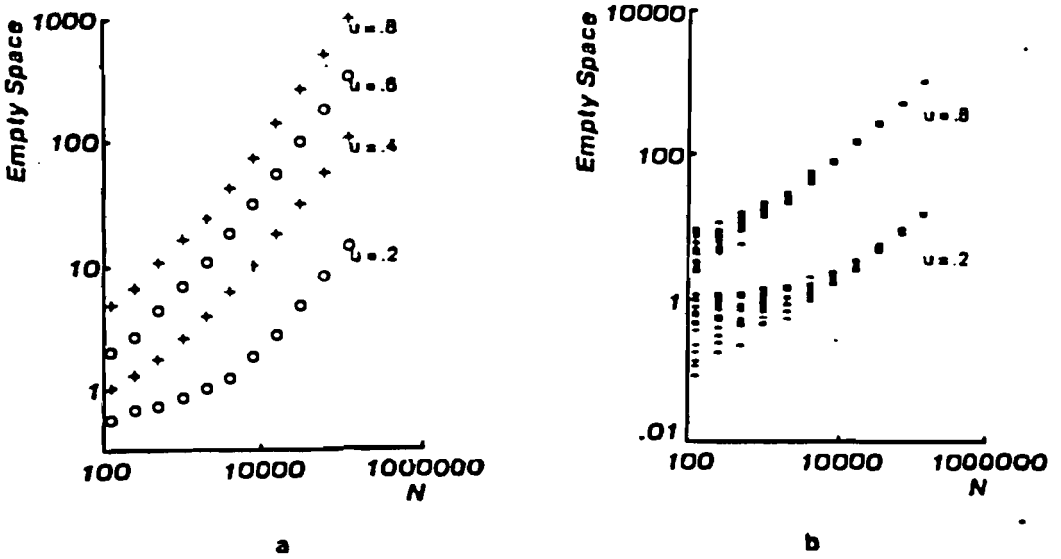


Exhibit 2-7: Growth in n

n (that is, for sample points with $8000 \leq n \leq 128000$) has slope near 0.938. Similar fits to the four, three, and two highest values give slopes of 0.943, 0.954, and 0.966, respectively.

Although the increasing slopes and upward curvature in the residuals (not shown) provide some evidence for asymptotic linearity, the results are not conclusive: the steady change in slopes suggests that n is too small for an accurate assessment of asymptotic behavior. Of course, it is likely that a function form other than a power law, perhaps $n/\lg n$, is more appropriate. A regression fit using this model at $u = 0.8$ (and $8000 \leq n \leq 128000$) also leaves an upward curve in the residuals, although the curve is more shallow. At $u = 0.2$, a least-squares fit at the five highest n values gives a slope of approximately 0.75, and a fit at the two highest values gives 0.78. The lower slope suggests that either empty space actually grows sublinearly at this u value (as was the case with $u = 1$), or that the curve is very slow in approaching its asymptotic form.

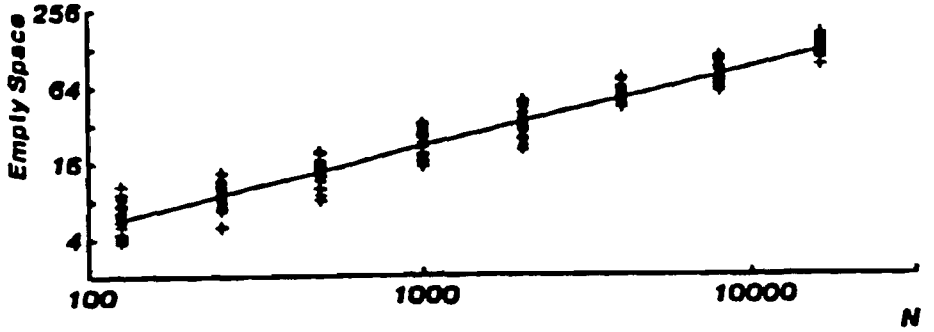
2.4. Best Fit

The Best Fit algorithm is similar in many ways to First Fit. Both are on-line algorithms, and both can be implemented to run in $O(n \log n)$ time. First Fit needs only a simple heap data structure to find the first bin that can contain a given item, however, while Best Fit requires some sort of balanced tree mechanism to find the bin into which the item fits most tightly.

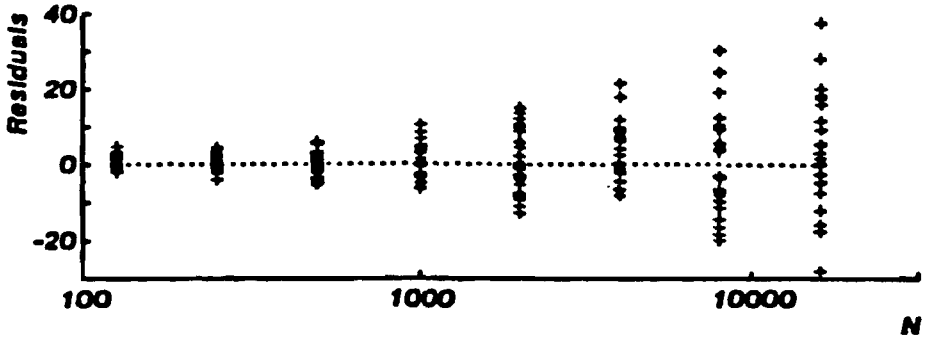
Partly because of the implementation requirements, the simulation program for Best Fit uses linear search to find the best-fitting bin and therefore requires $\Theta(n^2)$ time to pack. Simulations of Best Fit packings were only taken for n up to 16000 rather than to 128000. Although the program efficiency was not great, the efficiency of the experimentation – in terms of human and computer time spent in the search for useful measurements – was enhanced by the fact that the First Fit study had been done first. The understanding of packing structures gained from that study allowed similar analysis of Best Fit with much less exploration time. Similarly, since the results are analogous to those for First Fit, the pace of this section is faster.

Graph 2-8-a depicts empty space in Best Fit packings as a function of n with $u = 1$, and Graph 2-8-b shows the residuals from this fit. A linear least-squares fit on this log/log scale has slope 0.619, which is better than the corresponding value of 0.701 observed for First Fit. The regression results suggests that empty space is sublinear in n; as before, this leads to the conjecture that Best Fit is asymptotically optimal for this input model. Prompted by these observations, Shor [16] proved bounds of $O(n^{1/2} \log n)$ and $\Omega(n^{1/2}(\log n)^{1/4})$ for empty space in Best Fit packings when $u = 1$.

Because Shor's results imply that empty space grows more slowly for Best Fit than for First Fit, Best Fit must produce better packings asymptotically when $u = 1$. It is clear from Exhibit 2-9 that Best Fit produces better packings at smaller n and u as well. Graph 2-9-a shows the ratio of empty space in First Fit packings to that for Best Fit packings of identical weight lists, for 10 trials each at $u = 1$ and $125 \le n \le 8000$. In all cases the ratio is greater than 1, indicating that BF gives better performance. Graph 2-9-b compares empty space for the two algorithms on identical weight lists at $n = 16000$ and $0.2 \le u \le 1$. Best Fit generally gives better packings throughout, although the differences are negligible at small u. Although Best Fit and First Fit are both online algorithms, there is no *a priori* reason to suspect that similar nonmonotonic behavior will be exhibited. Graphs similar to 2-9-b indicate, however, that the nonmonotonicity is even more pronounced with Best Fit, since it becomes apparent at smaller n and is more sharply defined as n grows.



a



b

Exhibit 2-8: Best Fit, $u = 1$

The obvious question is whether the bins are distributed by item classes and by gap ranges as they are for First Fit. Preliminary experiments measuring f, the fraction of k-item bins in a packing, suggest behavior almost identical to that displayed for First Fit (in Graph 2-6). The only significant difference discovered so far is that when u is above 0.9, Best Fit tends to give a slightly higher fraction of 1-item bins and slightly lower fractions of 2-item and 3-item bins than First Fit.

The arguments for nonmonotonic behavior in First Fit can be adapted to Best Fit. Showing that e (a lower bound on the gap in 2-item bins) is bounded away from zero seems even more difficult in this case, however, since Best Fit finds pairings that minimize this gap. Complete analytical characterization of online packings remains a formidable open problem.

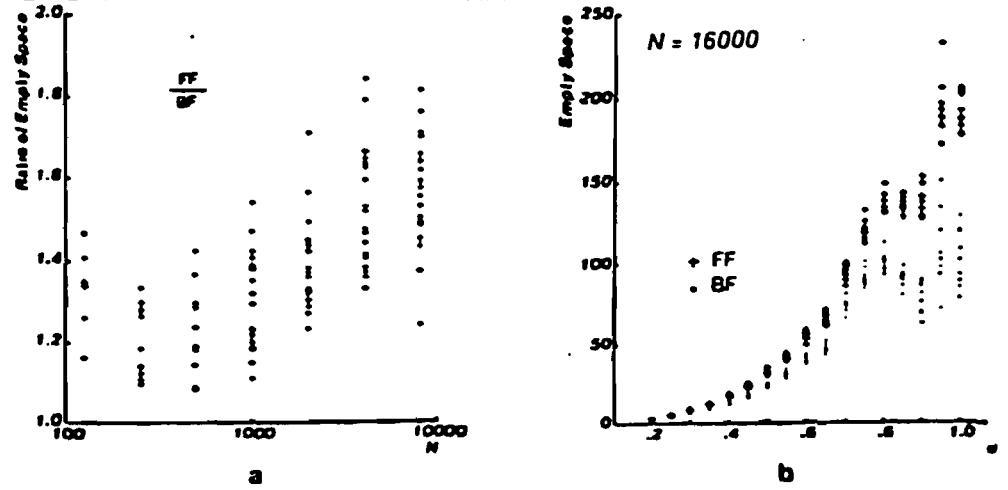


Exhibit 2-9: Comparing Best Fit to First Fit

2.5. First Fit Decreasing

The First Fit Decreasing algorithm sorts the items by decreasing weight before applying First Fit to the list. Prompted by early simulation results of Bentley and Faust [1], Leuker [14] showed that $\Delta FFD(L_{1,n}) = \Theta(n^{1/2})$ when $u = 1$.

Exhibit 2-10 depicts empty space as a function of n for five values of u. Each panel of Graph 2-10-a presents empty space for 25 trials at each sample point; for comparison of scales, a horizontal line in each panel is drawn at one bin. In Graph 2-10-b, each curve presents mean values for the corresponding panel of Graph 2-10-a. Note that the n values double as they increase and that the abscissa is on a logarithmic scale.

The most striking observation, from the bottom two panels of 2-10-a, is that empty space does not appear grow in n when u is small. The conjecture that empty space is constant in n when $u \leq 0.5$ was first made in [2] and subsequently proved in [3]. Subsection 2.5.1 presents a closer look at the packings when u is less than 0.5.

The presence of outliers – which indicate very bad packings – when u is large (at 0.7 and 0.8) and n is small (less than 1000) is also of interest. Exhibit 2-11 gives another view of the outliers. Graph 2-11-a presents empty space as a function of u for three fixed values of n (as before, note the differences in scale among the panels). The bottom two graphs present mean empty space for the corresponding panels above.

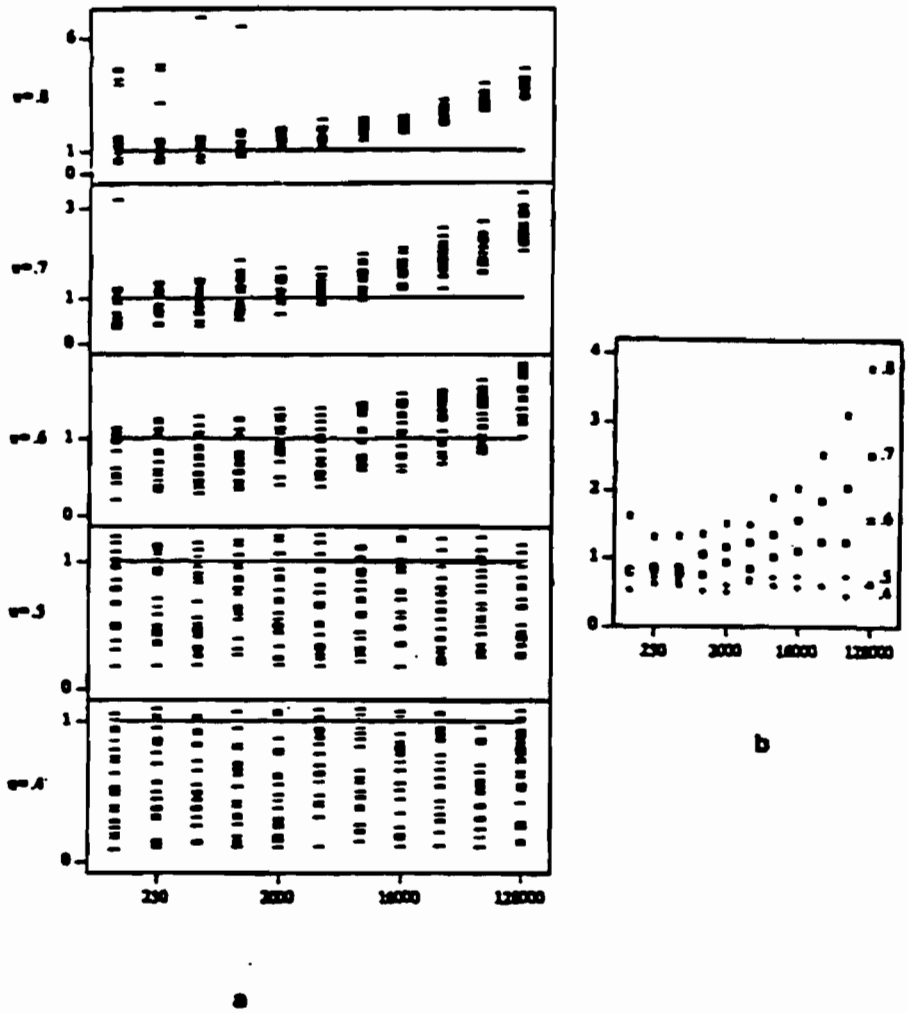


Exhibit 2-10: First Fit Decreasing

When u nears 1 (the panels on the right), a number of outliers appear and empty space suddenly displays a large variance as well as a rapid increase in the mean. As n increases, this *critical region* (where the bad packings occur) appears to shift to the right. This behavior is also observed in Graph 2-10-a: the top panel suggests that $u = 0.8$ is no longer in the critical region when $n = 2000$, since outliers are no longer seen.

The left panels of Exhibit 2-11 suggest that behavior at $u \leq 0.5$ is quite different from that at $u > 0.5$. In fact, the bottom left graph suggests that empty space grows linearly in u when u is

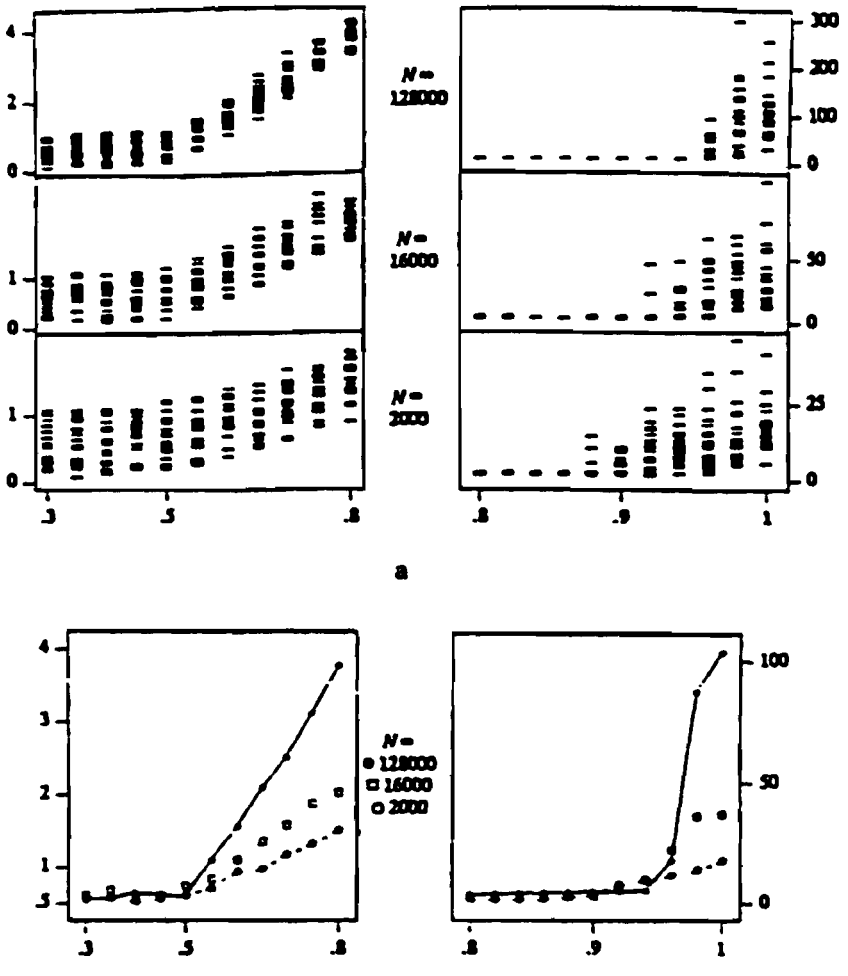


Exhibit 2-11: First Fit Decreasing

between 0.5 and the critical region. Subsection 2.5.2 examines the critical region and gives a partial characterization of the causes of bad packings; behavior when u is below the critical region but above 0.5 is also explored.

2.5.1. u Below 0.5

The proof (in [3]) that empty space is constant in n when $u \leq 0.5$ gives an upper bound of at least 10^{10} bins; Floyd and Karp [7] recently improved this bound to 10 bins, using a slightly different model of input probabilities. The bottom panel of Graph 2-10-a suggests that the mean is in fact nearer to 0.7. Moreover, empty space is less than 1 over 75% of the time, in which case the packing must be optimal (because the optimal packing cannot use fewer bins and still contain the entire weight list). This section examines behavior at $u \leq 0.5$ more closely.

Since the minimum number of bins used is $\lceil \Sigma(L_{u,n}) \rceil$, there must be at least $\lceil \Sigma(L_{u,n}) \rceil - \Sigma(L_{u,n})$ empty space in even an optimal packing: the last bin in the packing represents a sort of "spillover" bin, whose gap is more an artifact of the weight sum than of the packing quality. Let the *partial empty space* of a packing refer to the empty space in all but the last bin.

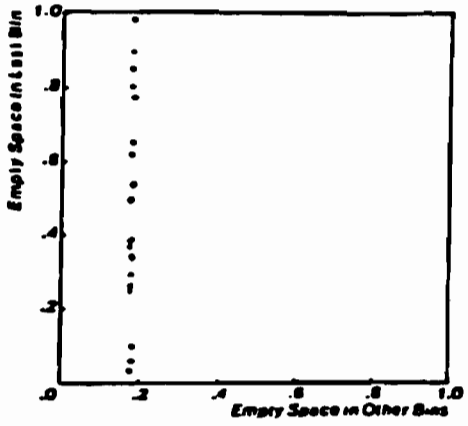


Exhibit 2-12: Last Bin vs Partial Empty Space

Exhibit 2-12 plots empty space in the last bin against empty space in all the other bins (partial empty space), for 25 trials at the sample point $n = 128000$, $u = 0.5$. Empty space in *all* the other bins (approximately 32000 of them at this sample point) remains between 0.17 and 0.18, while empty space in the last bin ranges between 0 and 1. Total empty space (the sum of these two quantities at each trial) is therefore almost completely dominated by variation in the last bin.

Graph 2-13-a compares total empty space (left panels) to partial empty space (right panels) for sample points at $n = 128000$, $0.1 \le u \le .5$. Note the dramatic increase in precision: empty space has an observed range of approximately 1 bin, while partial empty space is usually less than ± 0.001 bin from its mean. The growth of partial empty space in u was completely obscured by the last bin.

Graph 2-13-b shows partial empty space for $n = 128000$ and u in increments of .01. Graph 2-13-c shows the same information on a log/log scale, with a linear regression line superimposed. Partial empty space is nearly linear on this scale, except for an increase at the high end. The fit has slope of 2.11, indicating that partial empty space grows approximately as $u^{2.11}$. (Fits at smaller n are consistent with this). The residuals from this fit provide an interesting pattern: in Graph 2-13-d, peaks appear at $u = 1/2, 1/3, .1.4, \ldots$, suggesting that a cyclic component exists. Graph 2-13-e shows residuals from a fit to a degree-3 polynomial. Similar peaks appear with this model and in fits with as high as degree-5 polynomials.

An obvious question is whether this cyclic behavior is somehow an artifact of the simulation. When n is very large and u very small, the average difference between successive weights is small, as are the gaps in the bins: perhaps errors due to machine precision are propagated in some fashion to cause this pattern. Evidence exists to suggests that the cyclicity is not an artifact of machine precision or implementation. First, the smallest item weight ever generated has expected size about one millionth of a bin (this occurs when $n = 128000$ and $u = 0.1$), and partial empty space is near 0.005, giving an average gap of one millionth of a bin. The weights are represented by 30-bit integers, which can represent one billionth of a bin, so machine precision is not overwhelmed. Second, the cyclic behavior is observed at smaller n, which would presumably not be the case if machine precision were the problem. Third, the results were replicated on a secondary simulation environment, with differences in random number generator (linear congruential vs. cyclic feedback), machine precision (30-bit integers to represent the weights vs 16-bit reals), implementation, programmer, and programming language. The only differences in partial empty space between the two are predicted by analysis of the differences in precision between the two systems.

It is possible that the cyclicity in the residuals can be explained by an argument similar to that for the nonmonotonic behavior of First Fit: that is, the observed behavior is a result of the interaction between the fraction of k-item bins and the gaps in k-item bins. Analysis of FFD packings suggests that k-item bins have a great deal of structure.

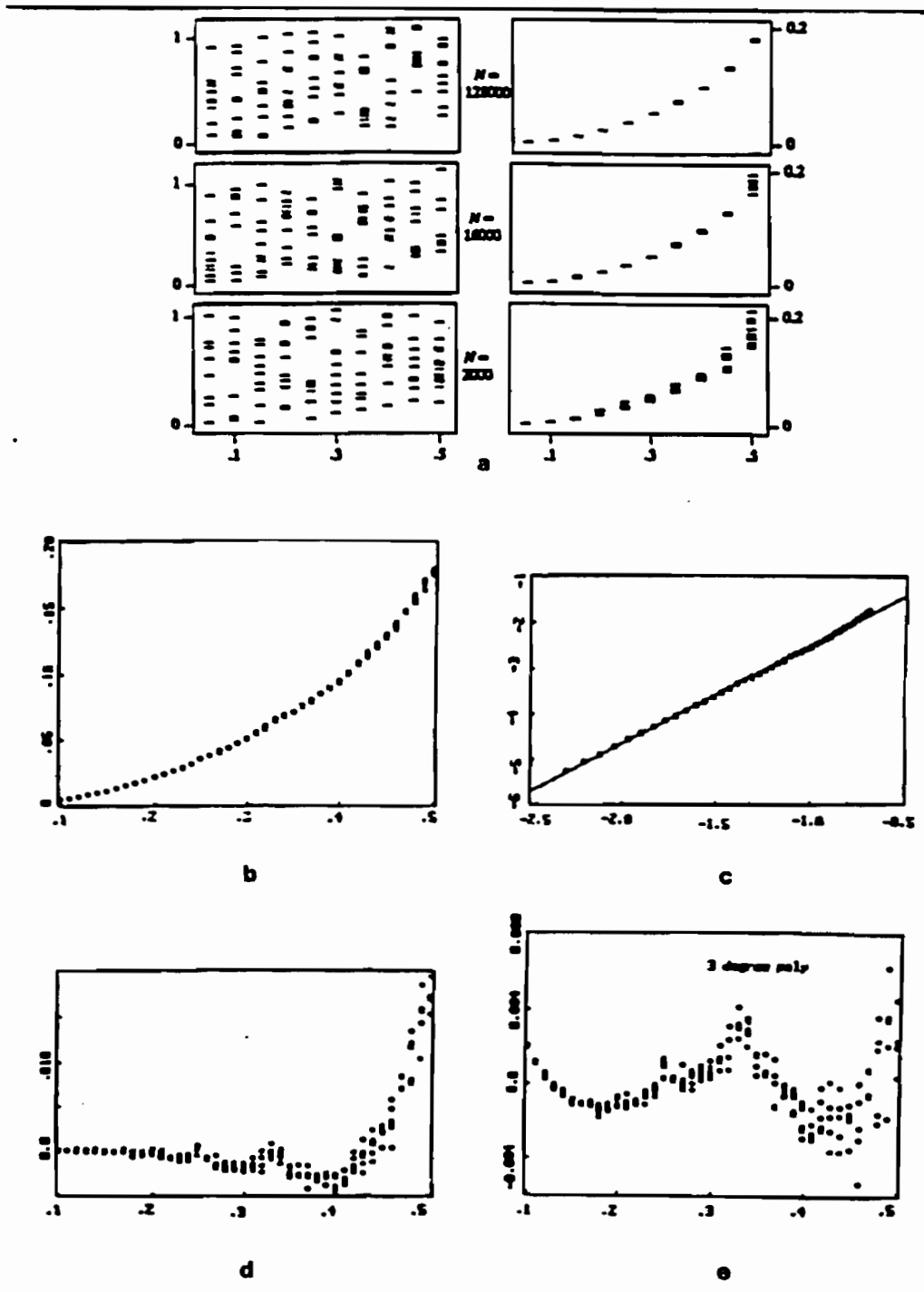


Exhibit 2-13: Partial Empty Space

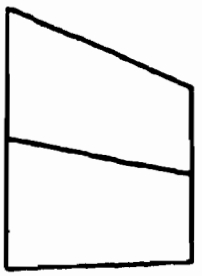


Figure a: *Items with weight between 1/2 and 1/3 have been packed. The items are stacked two-to-a-bin.*

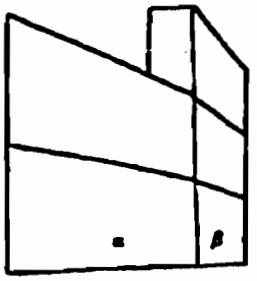


Figure b: *Items with weight between 1/3 and 1/4 have been added to the packing. Most of these items are stacked three-to-a-bin in region β. Some items backfill onto region α. The vertical lines mark the edges of regions α and β.*

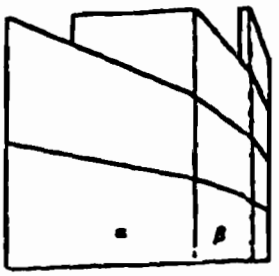


Figure c: *Items with weight betwen 1/4 and 1/5 have been added. Most of these items are stacked four-to-a-bin, but some backfill onto regions α and β.*

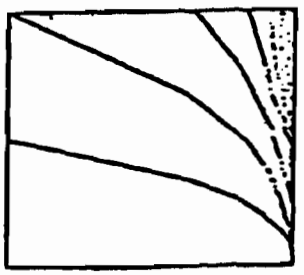


Figure d: *All the items have been packed.*

Exhibit 2-14: FFD Packings

Exhibit 2-14 presents "snapshots" of an FFD packing of weights drawn from [0, 0.5]. Each item is represented by a very narrow white vertical bar with its top marked in black. There are so many items that the black tops appear to form a continuous line. The first (largest) item goes into the first bin, and so must the second, since the weights are near 0.5. The third and fourth items must go into the second bin. This packing of items 2-to-a-bin continues until all items with weight greater than 1/3 have been packed (Figure a). Call the bins packed up to this point *2-bins* (not 2-item bins, since they will eventually contain more items). Once the items are of size less than 1/3, they may be packed 3-to-a-bin (call these the *3-bins*). Some of these items, however, are small enough to "backfill" the 2-bins. This process of backfilling and packing 3-to-a-bin continues until items with weight greater than 1/4 have been packed (Figure b). Continuing, items either backfill in the 2-bins or the 3-bins, or are packed 4-to-a-bin in new bins. This pattern continues until all items are packed. Observation of this structure in FFD packings was central to our proof of constant empty space in [3]. More importantly, "movies" of the packings, obtained from simple algorithm animation techniques, were directly responsible for suggesting the proof technique.

2.5.2. u above 0.5

This subsection examines FFD packings for $u > 0.5$. Empty space is measured here rather than partial empty space, since the last bin does not dominate the measurements. We first study the critical region, where outliers in empty space appear and mean empty space increases rapidly as u nears 1.

Recall from Exhibit 2-10 that the left side of the critical region appears to shift to the right at a rate at most logarithmic in n. It is not clear how to characterize the region by experimental methods; is likely, for example, that quantities such as the "edge of the region" and the "fraction of outliers" would be artifacts more of sample size and data analysis tools than of underlying phenomena. Instead of measuring properties of the critical region, this section examines properties of the weight list that are correlated with bad packings.

Exhibit 2-15 presents, for $n = 2000$ and $.84 \leq u \leq 1$, the distribution of empty space for 25 trials at each sample point. A panel corresponds to at single sample point; in each, the measurements of empty space for the 25 trials are plotted against their ranks. In the top left and center panels, corresponding to $u = 0.84$ and 0.86 (outside the critical region), empty space is fairly uniformly distributed between [0.8, 2] and [1, 2], respectively. At $u = 0.88$ the distribution smooth except for the last 3 points, which are sharply higher. At $u = .90$ a break is

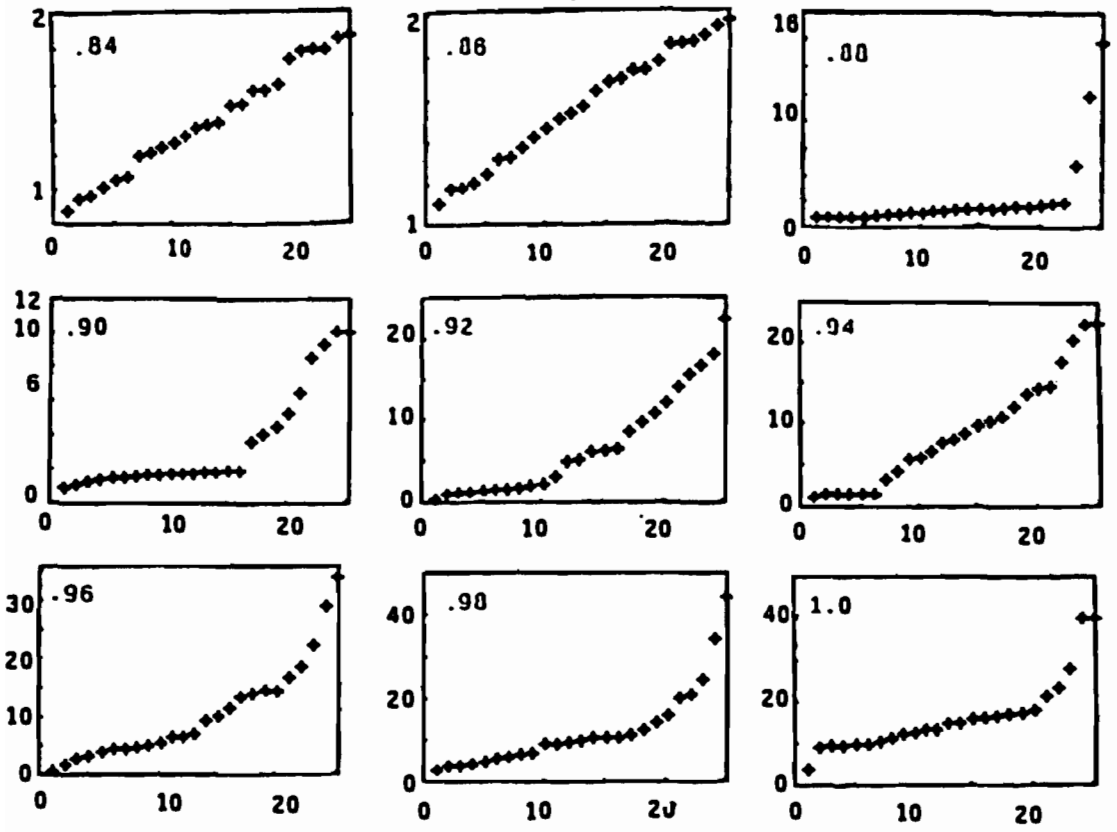


Exhibit 2-15: Distribution of Empty Space

seen at the 16th-largest point. At higher values of u an abrupt break is no longer seen. Theoretical characterization of the distribution of empty space for fixed n and u is an open problem. Although the sample size is small, it appears that the distribution changes significantly over the critical region.

A new measure gives more insight into behavior in the critical region. Call items with weight greater than 0.5 the *big items* in an input list. The expected number of big items is $n(u-0.5)/u$; at $n = 2000$, $u = 0.8$, for example, $2000(0.3/0.8) \approx 750$ of the items in the weight list are expected to be big items.

Graph 2-16-a presents the results of 1000 trials at $n = 2000$, with u generated uniformly *at random* from the interval [0.82, 0.88]. The y-coordinate of each point corresponds to empty space for that trial; the x-coordinate gives the difference δ between the *number of big items* in

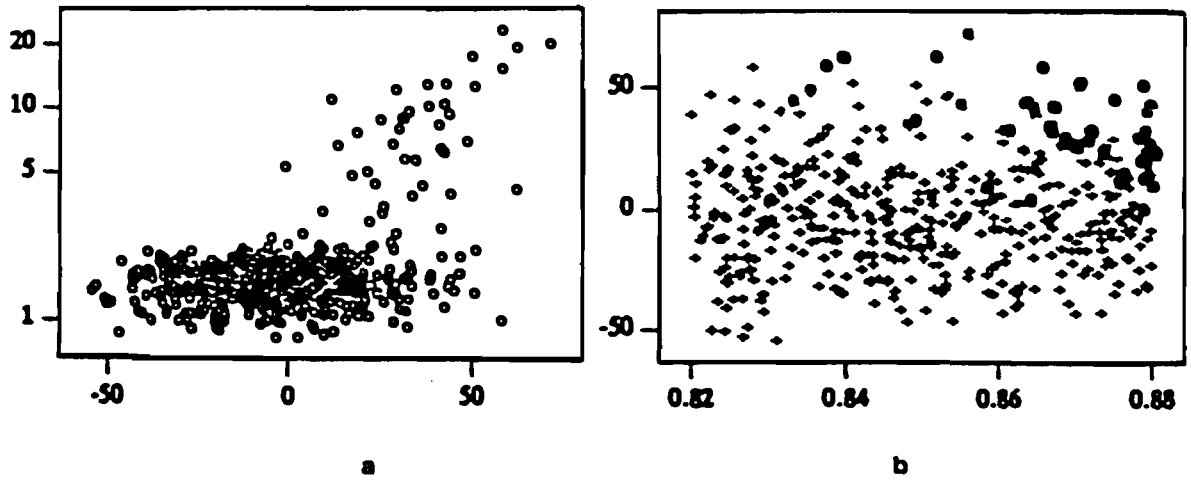
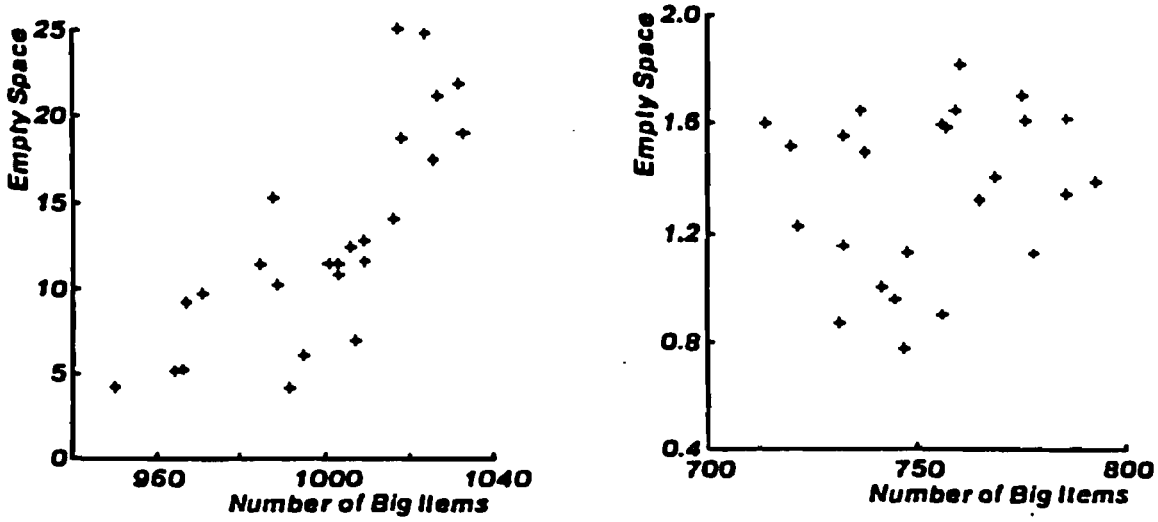


Exhibit 2-16: Bad Packings and δ

the list and *its expected value* for the corresponding u. For example, a point with x-coordinate 20 might corresponds to a trial with 2000 items generated uniformly from the range (0, .825]. The expected number of big items at this sample point is $2000(.825-.5)/.825 \approx 788$, but the list actually generated at that trial had 808 big items, so we have $\delta = 808-788 = 20$. In Graph 2-16-b, δ is plotted against u for each of the 1000 trials. The relatively bad packings – the 40 trials having empty space between 3 and 50 – are highlighted in the graph. (All other packings had fewer than 3 bins of empty space.)

Lists with positive δ (the topheavy weight lists) tend to give bad packings. From Graph 2-16-a it appears that very topheavy lists tend to give very bad packings. On the other hand, not all topheavy lists give bad packings; from Graph 2-16-b it appears that FFD is more sensitive to topheavy lists at high u, since bad packings are generally concentrated in the upper right corner of the graph.

Exhibit 2-17 supports this last observation. Graph 2-17-a plots empty space against the number of big items for 25 trials at the sample point $n = 2000$, $u = 1$. The expected number of big items at this sample point is 1000. Empty space increases with the number of big items in the weight list; that is, topheavy lists tend to give bad packings. In contrast, Graph 2-17-b depicts empty space versus the number of big items at $n = 2000$, $u = 0.8$, which is well below the critical region. Outside the critical region, empty space does not appear to grow with the number of big items.



a: $u = 1$ b: $u = 0.8$

Exhibit 2-17: Empty Space vs. Big Items

These and similar graphs suggest that as u nears 1 FFD becomes increasingly sensitive to the number of big items in the list. If the conjecture were true, it would explain why the critical region appears to shift to the right at high n. Let μ represent the expected number of big items in a list. By the central limit theorem, as n grows the probability of generating a list with more than $d\mu$ big items *decreases* exponentially as n grows. On the other hand, at fixed n the probability of generating a topheavy list *increases* linearly in u, since the range of weights increases. At high n and small u, therefore, the probability of getting a bad list is too low for outliers to be seen in 25 trials. Since the probability increases in u the boundary of the critical region would appear to shift to the right. While this observation gives a first cut at characterizing packings in the critical region, it is clearly not complete. Why, for example, do some topheavy lists cause bad packings and others do not? Further simulation might reveal more.

Finally, consider the packings in the well-behaved area with u below the critical region but above 0.5. The analytical bound of $O(n^{1/2})$ for empty space when $u = 1$ can be extended to any u (see [14]). In addition, it can be shown (see [3]) that $E[\Delta FFD(L_{u,n}) = \Omega(n^{1/3})$ for $0.5 < u < 1$. Finding the functions of u implicit in these order-of-magnitude bounds is an open problem.

Recall Exhibit 2-11, which depicts growth in u for three values of n; it appears from the bottom left graph that mean empty space grows linearly with u in this range. Linear regression fits to the simulation measurements in this region indicate that empty space is approximately described by the function $E[\Delta FFD(L_{u,n})] = 4.9n^{1/2}(u-0.5)$. As with regression fits when u is below 0.5, a cyclic component is once again seen in the residuals.

2.6. Best Fit Decreasing

The Best Fit Decreasing algorithm, like FFD and unlike BF and FF, sorts the item list before packing. BFD and FFD are compared in this section. Although the expected performance of FFD has been characterized to some extent for all values of u, theoretical analysis of BFD is much less complete.

Simulations comparing BFD and FFD on identical weight lists at various sample points suggest that the two algorithms give almost identical performance for this input model. For example, in 25 trials each at the sample points given by $u = 1$, $n = 125, 250, \ldots 8000$, comprising 175 measurements, empty space in corresponding packings differed in only one trial (at $n = 8000$), when FFD used 2 more bins than BFD. Note that bin counts differ if and only if empty space differs, since the weight lists are identical at each trial. This close correspondence holds for other values of u as well: experiments at $u = 0.8$ (and the same n values) produced one trial where bin counts differed by 1, and trials at $u = 0.5$ produced no differences. Similar experiments with n fixed at 8000 and $u = 0.2, 0.4, 0.6, 0.8$ produced no differences in bin count.

It is not necessarily the case that BFD and FFD give identical *packings*. Whether the packings are identical can be resolved to some extent by measuring partial empty space, which corresponds to empty space in all but the last bin. If the packings are identical, then partial empty space will be equal for the two. The converse is not necessarily true: it is possible that the packings be different but partial empty space is the same. Preliminary experiments at $u=1$ and varying n suggest that, on average, partial empty space differs between the two rules less than 1/3 of the time. Limited experiments at $n = 800$ give differences 84% of the time at $u=0.8$ and 65% of the time at $u=0.4$.

Consider the structure of BFD packings as compared to the pictures of FFD packings in Exhibit 2-14. On a perfect list (with items evenly distributed between 0 and u), BFD and FFD must produce identical packings because the first bin into which an item fits is also the best.

Suppose a random list with $u = 0.5$ is to be packed. Certainly the items with weight in the range (1/3, 1/2] would be packed identically by BFD and FFD. In the next packing stage it is possible that an item with weight from (1/4, 1/3] has its "best" fit in a 3-bin, but its "first" fit in a 2-bin, but this would only result in two items being swapped in the two bins, which would not affect partial empty space since the weight sum and the bin count are identical. By this argument it is surprising that partial empty space is observed to differ so often for u less than 1. Further experiments with more detailed measures could give more information about packing properties of the two rules.

2.7. Future Work

The results in this chapter have extended current understanding of the expected-case behavior of the four bin packing heuristics. Not only do the measurements allow a precise description of mean empty space as a function of n and u, but examination of properties of the packings gives deeper insight and new arguments to explain observed behavior.

An obvious next step is to develop theoretical characterizations of the heuristics, perhaps by formalizing some of the arguments contained here. It is unlikely that theoretical characterization as precise as these measurements will be obtained by current techniques.

The experimental work could be extended in a number of ways. First, it would be interesting to measure packings for nonuniform distributions on weights. Also, many other heuristics are worthy of consideration: Coffman, Garey, and Johnson [5], for example, survey results for over 20 packing rules. Little is known about the expected performance of most of these rules.

References

[1] J. Bentley, J. Faust.
Unpublished notes on simulations of FFD.
1980.

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

[3] J. L. Bentley, D. S. Johnson, F. T. Leighton, C. C. McGeoch, L. A. McGeoch.
Some unexpected expected-behavior results for bin packing.
In *Proceedings, 16th Symposium on Theory of Computation*. ACM, April, 1984.

[4] D. J. Brown.
A lower bound for on-line one-dimensional bin packing algorithms. .
Report No. R-864, Coordinated Science Laboratory, University of Illinois, Urbana IL, 1979.

[5] E. G. Coffman, Jr, M. R. Garey, D. S. Johnson.
Approximation Algorithms for Bin-Packing – An Updated Survey.
Available from the authors at Bell Laboratories, Murray Hill, NJ 07974.

[6] W. Fernandez de la Vega and G. S. Leuker.
Bin packing can be solved within 1 + e in linear time.
Combinatorica 1:312-320, 1981.

[7] S. Floyd and R. Karp.
FFD bin-packing for distributions on [0, 1/2].
In *Proceedings, 27th Symposium on Foundations of Computer Science*. IEEE, October, 1986.

[8] G. N. Frederickson.
Probabilistic analysis for simple one- and two-dimensional bin packing algorithms.
Information Processing Letters 11(4,5):156-161, December, 1980.

[9] D. S. Johnson.
Near-Optimal Bin Packing Algorithms.
PhD thesis, Department of Mathematics, Massachussetts Institute of Technology, Cambridge MA, June, 1973.

[10] N. Karmarkar.
Probabilistic analysis of some bin-packing algorithms.
In *Proceedings, 23rd Symposium on Foundations of Computer Science*, pages 107-111. IEEE Computer Society, 1982.

[11] N. Karmarkar and R. M. Karp.
An efficient approximation scheme for the one-dimensional bin packing problem.
In *Proceedings, 23rd Symposium on Foundations of Computer Science*, pages 312-320. IEEE Computer Society, 1982.

[12] D. E. Knuth.
The Art of Computer Programming: Volume 2, Seminumerical Algorithms.
Addison-Wesley Publishing Company, Reading, MA, 1973.

[13] G. S. Leuker.
Bin packing with items uniformly distributed over intervals [a,b].
In *Proceedings, 24th Symposium on Foundations of Computer Science*, pages 289-297. IEEE Computer Society, 1983.

[14] K. Maruyama, S. K. Chang, and D. T. Tang.
A general packing algorithm for multidimensional resource requirements.
International Journal of Computer and Information Sciences 6(2):131-149, 1977.

[15] H. L. Ong, M. J. Magazine, T. S. Wee.
Probabilistic analysis of bin packing heuristics.
Operations Research 32(5):983-998, September-October, 1984.

[16] P. W. Shor.
The average-case analysis of some on-line algorithms for bin packing.
In *Proceedings, 25th Symposium on Foundations of Computer Science*, pages 193-200. IEEE , October, 1984.

Chapter 3
Greedy Matching In One Dimension

3.1. Introduction

This chapter studies a restriction of the following problem: Given N points within the d-dimensional unit hypercube, what is the pairwise matching of the points that minimizes the sum of the Euclidean distances between matched pairs? The planar version of this problem arises, for example, in scheduling mechanical plotters: the input is a connected graph in the plane and the plotter must draw lines at all edges, minimizing the amount of time that is wasted while the plotter moves with pen up. Wasted pen movement can be eliminated if an Eulerian cycle exists in the graph. There is no Eulerian cycle if and only if the graph has an even number $N > 2$ of vertices of odd degree; in that case the minimum matching of those vertices can be added to the edge set to obtain the tour with minimum wasted movement.

The matching problem for points uniform on the unit square has been studied extensively. Edmonds [3] showed that a minimal matching in a general graph of N vertices can be found in $O(N^3)$ time, but this can be too expensive for plotting applications because N is often very large, say, in the thousands. Fast approximation algorithms are therefore of interest; Avis [1] reviews work in this area.

An obvious approximation algorithm for minimum matching is the "Greedy" one: keep removing the pair of vertices with minimum edge cost until the matching is complete. The straightforward implementation takes $O(N^3)$ time; Manacher and Zobrist [5] describe a version that from experimental results appears to run in $O(N)$ expected time when the points are distributed uniformly in the unit square.

Let $E_H(N)$ denote the *edge cost* of the matching produced by heuristic H; this is the sum of the weights of edges in the matching. (The subscript is dropped in the following when reference to the Greedy Heuristic is clear from the context.)

Reingold and Tarjan [6] showed that for any graph that obeys the triangle inequality, the worst-case ratio $E_{Greedy}(N)/E_{Optimal}(N)$ is bounded above by $(4/3)N^{\lg 1.5}$, or about $(4/3)N^{.585}$. Avis, Davis and Steele [2] showed that when the points are uniformly distributed within the unit d-cube (for d > 1), the cost of the Greedy matching asymptotically approaches $c_d N^{(d-1)/d}$, which is within a constant factor of the optimal matching.

We examine here the expected performance of the Greedy heuristic in one dimension: that is, the points are drawn independently from the uniform distribution on [0,1]. Certainly Greedy is a poor choice for this problem, since the optimal matching can be found quickly by pairing the leftmost point with the second leftmost, the third with the fourth, and so on, giving an expected matching cost of approximately 1/2. Although Greedy should not be implemented expressly for one-dimensional matching, its expected-case behavior is of interest. First, points in higher-dimensional problems might happen to lie on a straight line. It is useful to know how Greedy performs in this potentially frequent case. The second reason is purely theoretical: although Avis, Davis and Steele have characterized the expected edge cost in d-space for d > 1, the case d = 1 remains open. Finally, the expected running time of the Greedy algorithm described below, a modification of the Manacher and Zobrist algorithm for planar points, is an open problem: the one-dimensional case can give insight into behavior at higher dimensions.

3.2. The Study

Exhibit 3-1 presents a description of the Greedy algorithm used as the simulation model. The algorithm uses an array A containing the *N* points from the interval [0, 1]; the points are assumed to be presorted in increasing order. Greedy makes repeated passes through the point set, at each pass locating the smallest gap between adjacent points, accumulating edge cost, and removing the pair from the point set. This algorithm runs in time quadratic in *N*.

The simulation program is more efficient than the straightforward implementation. The program description is given in Exhibit 3-2; this is a modification of an implementation proposed by Manacher and Zobrist [5] for the two-dimensional matching problem. The simulation program also makes repeated passes through the point set, but each pass removes many points from the set rather than a single pair. Two points comprise a *nearest neighbor pair* if each is the nearest neighbor of the other; the distance between them is therefore a local minimum, and would eventually be removed by Greedy. Rather than removing the pair

```
Greedy(N)
Input: Array X of points, X[1] <= X[2] . . . X[N]
Output: edgecost

    while N > 0 do
        mingap = MaxReal;
        for i := 2 to N do                    Find smallest gap.
            gap = X[i] - X[i-1]
            if (gap < mingap) then
                mingap = gap
                index  = i
        edgecost = edgecost + mingap          Accumulate costs.

        for i := index+1 to N do              Remove matched points.
                X[i-2] = X[i]

        N = N - 2
```

Exhibit 3-1: Greedy Algorithm: Quadratic Implementation

with minimum gap at each pass, the program removes *all* pairs of points with locally minimum distances. Since removing points cannot produce smaller gaps, Program 3-2 correctly implements the Greedy heuristic.

Two measures of the Greedy algorithm are of interest. The *edge cost* corresponds to the sum of the lengths of the edges formed in the Greedy matching. Let $E(N)$ denote the expected edge cost for N points drawn uniformly and independently from the interval (0, 1]. The time required to compute the matching at each iteration of the while loop is proportional to the number of points encountered (assuming a preprocessing step to sort the points). The *computation cost* is therefore proportional to the sum, over all iterations of the while loop, of the number of points remaining at each iteration; let $C(N)$ denote expected computation cost.

The following section presents simulation results for the Greedy algorithm. The primary simulations were performed on a VAX-11/750[1] using 32 bit integers and 64 bit (55 bit mantissa) double precision reals. For program efficiency, integer computation was used throughout; that is, points were drawn from the integer range $[0, 2^{30}]$ and results were scaled to the real range [0,1] for output only.

The only parameter to the simulation is N, the number of input points. Sample points for the

[1] VAX is a trademark of Digital Equipment Corporation.

```
Greedy(N)
Input: Array X of points, X[1] <= X[2] . . .X[N]
Output: edgecost

        while N > 0 do
                m = MaxReal                          Initialize.
                r = X[2] - X[1]
                X[N+1] = MaxReal                     Set up sentinel.

                for i = 2 to N do
                        l = m
                        m = r
                        r = X[i+1] - X[i]            Find local minima.

                        if (l > m) and (r >= m)  then
                                X[i-1] = X[i] = NIL
                                edgecost = edgecost+m Accumulate
                                                      edge costs.

                j = 1                                Remove matched pairs.
                for i := 1 to N do
                        if X[i] is not NIL then
                                X[j] = X[i]; j = j + 1
                N = j - 1
end
```

Exhibit 3-2: The Simulation Program

study were taken at powers of two from $16 = 2^4$ to $262144 = 2^{18}$. In general, 25 trials were performed at each sample point. Given N, the simulation program generated N points by the cyclic feedback method described by Knuth [4] (Algorithm A, Section 3.2.2). The numbers were then quicksorted and presented to the matching routine. Some experiments were replicated (for $N \leq 4000$) in Basic on a TRS-80 Model III computer, using the system random number generator and 32-bit reals.

3.3. Experimental Results

Some upper bounds are known for the edge cost $E(N)$ of a Greedy Matching. Reingold and Tarjan's [6] worst-case bound holds in the one-dimensional case, giving $E_{Greedy}(N)/E_{Optimal}(N) \leq 4/3N^{0.585}$ (recall that $E_{Optimal}(N)$ is approximately 1/2 for linear matching). Moreover, it is easy to show that $E_{Greedy}(N) = O(\log N)$ when the points are placed on a line (see [2]): when there are N points in the unit interval, the two nearest points must be at most $1/(N-1)$ apart. Removing these two gives $k = N-2$ points and the smallest edge distance is at most $1/(k-1)$. Therefore

$$E_{Greedy}(N) \le \sum_{k=0}^{\lfloor N/2 \rfloor} \frac{1}{(N-1-2k)} = O(\log N).$$

It is not known whether this upper bound is tight. By analogy with the results of Avis, Davis and Steele [2] for higher dimensions, it is natural to conjecture that the ratio of edge cost for Greedy is within a constant factor of Optimal. This problem was described by Mike Steele, who had been trying to prove the constant bound. The initial goal of the simulation study was to find an empirical bound on the constant and so to direct the theorem-proving efforts.

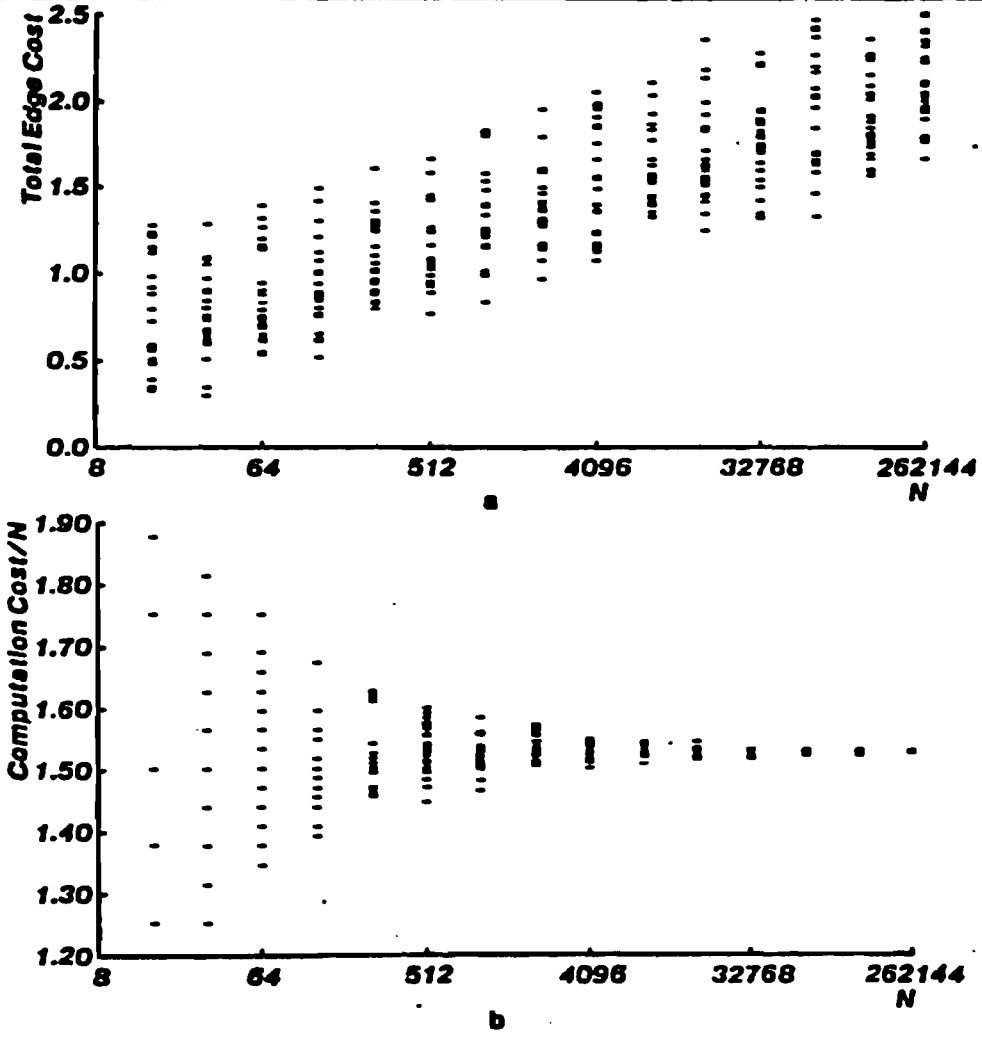


Exhibit 3-3: Performance of Greedy

Graph 3-3-a shows why the bound was so difficult to prove: $E(N)$ increases with N rather than remaining constant. The graph depicts the results of 25 trials each for N set at powers of 2 from 2^1 to 2^{15}. Since the x-scale is logarithmic and edge cost appears to grow linearly, logarithmic growth in N is indicated. A linear least-square fit on this scale gives

$$E(N) \approx 0.099 \log_2 N + 0.28.$$

This implies that unlike higher-dimensional cases, the ratio of the Greedy matching to the optimal matching is not bounded by a constant. Note that the coefficient is very small and that $E(N)$ therefore grows very slowly in N. Experiments over a smaller range of N values may not have permitted this observation.

The computation cost of the efficient implementation is also of interest. Computation cost is proportional to the total number of points examined in all passes through the point set. This number, divided by N, is displayed as a function of N in Graph 3-3-b; the results suggest that asymptotically $E(N) \approx 1.52N$.

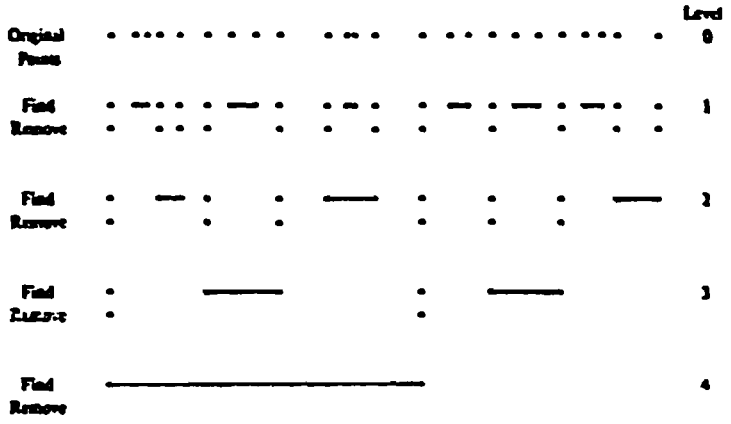


Exhibit 3-4: The Matching Algorithm

More detailed measurement of the matchings give more insight. Exhibit 3-4 shows the behavior of the shortcut algorithm on a small point set. Let a *level* correspond to one pass through the point sets, equivalent to one iteration of the outer loop in Program 3-2. At level 0, all the points are present. At each subsequent level, nearest neighbor pairs are removed and the *level edge cost* – the sum of the distances between paired points – is accumulated. This continues until some level where no points remain. Of course, different trials at the same sample point may not produce the same number of levels.

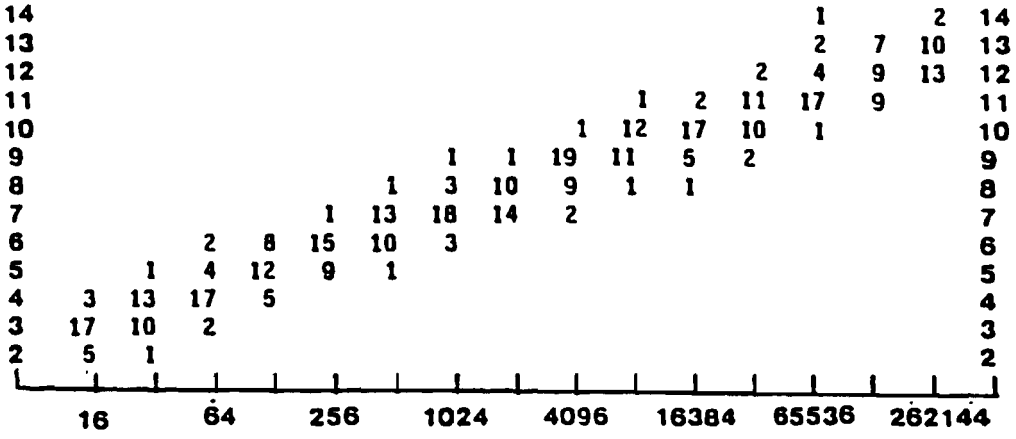


Exhibit 3-5: Number of Levels

Exhibit 3-5 shows the number of levels encountered as in 25 trials at each N. Each table entry gives the number of trials for which the corresponding N value (columns) reached the corresponding number of levels (rows). Zero entries are left blank. At $N = 16$, for example, 5 trials reached 2 levels, 17 trials reached 3 levels, and 3 trials reached 4 levels. Since N doubles each time (essentially producing a logarithmic scale on the N values) and the counts appear to increase linearly, this table suggests that the mean number of levels grows logarithmically in N.

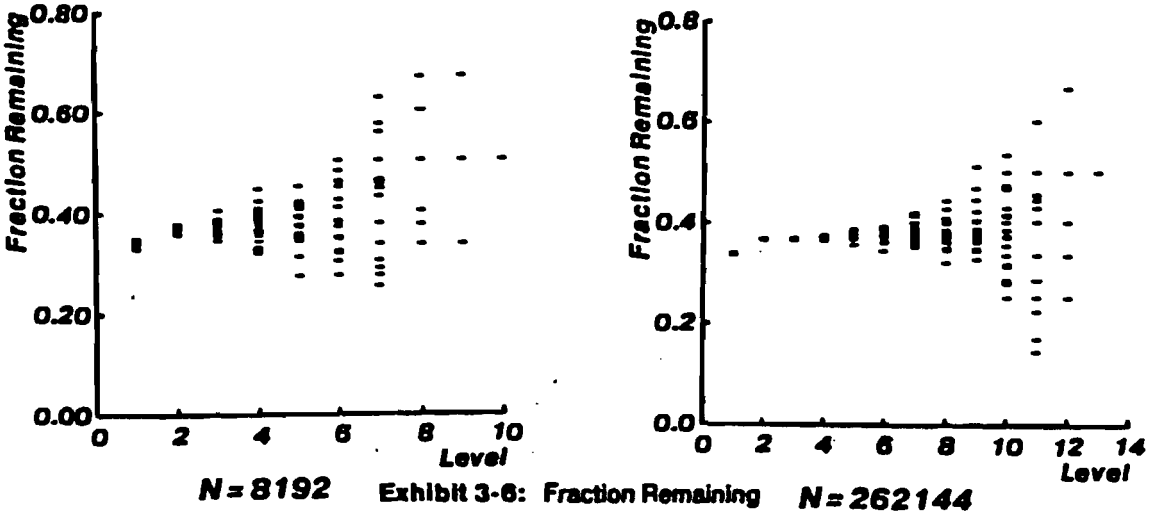


Exhibit 3-6: Fraction Remaining

This conjecture of logarithmic growth in the number of levels is supported by Exhibit 3-6, which presents the *fraction of points remaining* at each level for two values of N. The fraction remaining at level i is equal to the number of points at level i divided by the number of points at level i-1. For example, if there are 1200 points at level 0 and 900 are removed (leaving 300 points at level 1), then 300/1200 = 0.25 is the fraction remaining at level 1. Since approximately a constant fraction of points are removed at each level, only a logarithmic number of levels is typically reached.

Very nearly 1/3 of the original point set remains at Level 1. After Level 1, the mean fraction remaining is slightly higher (near 0.36), and is nearly constant throughout higher levels (although the variance increases). Note that at the last levels the fraction remaining must correspond to some "small rational" such as 2/4 or 4/6, since there are very few points left. The mean fraction remaining at a given level does not appear to vary with N. This observation suggests an argument for linear computation cost: at most of the levels a constant fraction f (observed to be near 0.36) of the points from the previous level remain to be processed. Since the cost of the algorithm at each level is linear in the number of points at the level, the recurrence for computation cost has the form

$$C(N) = C(fN) + O(N)$$

which has solution $O(N)$. Formalizing this argument would require finding an upper bound on f for most levels and either bounding the variance at the last levels or showing that they do not dominate the total computation cost.

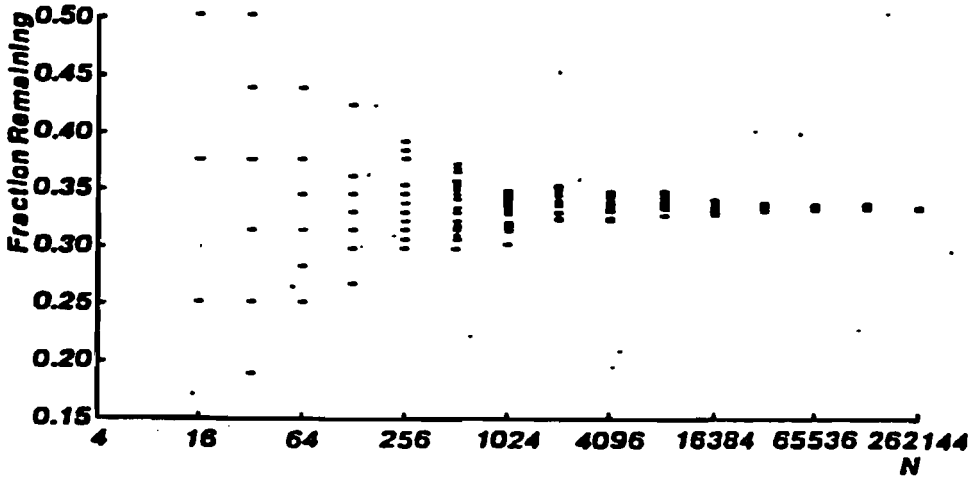


Exhibit 3-7: Fraction Remaining at Level 1

The fraction remaining at Level 1 is an interesting special case: given a set of points uniformly distributed on the unit interval, how many form nearest neighbor pairs? Exhibit 3-7 suggests that asymptotically two-thirds of the points form nearest neighbor pairs at Level 1, since the fraction remaining converges to 1/3. Steele [7] proves this observation. Intuitively, two points are nearest neighbors if the gap between them is a local minimum. For any triple of consecutive gaps, the probability that the middle one is the smallest is 1/3. Two points are removed every time this happens, so we expect to remove 2/3 of the points.

The points are no longer uniformly distributed after Level 1, so the above argument does not hold at later levels (Exhibit 3-6 indicates that the fraction remaining is near 0.36 at higher levels rather than 1/3). Theoretical characterization of the properties that determine nearest neighbor pairs at higher levels is an open problem. Note that a gap at Level 1 is either its original size or has been formed by the removal of nearest neighbor pairs, in which case it equals the sum of an odd number of original gaps. Since gaps between uniformly distributed points have a distribution similar to exponential, gaps at later levels are distributed as sums of (random numbers of) exponential variates.

Examination of the levels also gives an argument for logarithmic growth of edge cost (recall that total edge cost is the sum over all levels of the edge cost at each level). Exhibit 3-8 presents the edge cost per level for two values of N. At Level 1, edge cost is near 0.11; afterwards, mean edge cost remains near the constant 0.14 in the middle levels and increases at the last few levels. The mean edge cost per level does not appear to vary with N, although variance clearly depends on the number of points at a given level.

These observations suggest an argument for logarithmic growth of total edge cost. Suppose that level edge cost is near some constant e at all but the last few levels. Total edge cost must therefore be e times the number of levels. By the earlier argument relating fraction remaining to the number of levels, there are about $\log_{1/f} N$ levels, so E_N must grow as $e\log_{1/f} N$.

From the simulation results it appears that $e \approx 0.14$ and $f \approx 0.36$ (because $1/f \approx 2.7$), giving a conjecture that $E_N \approx 0.14 \log_{2.7} N$. A least-squares regression fit using the model $E_N = c_1 \log_{2.7} N + c_2$ for total edge cost produces $c_1 = 0.1427$ (which is very near the conjectured value) and $c_2 = 0.28$.

The simulation results presented here give new measurements, conjectures, and arguments for the performance of Greedy matching in one dimension. Formalization of the arguments

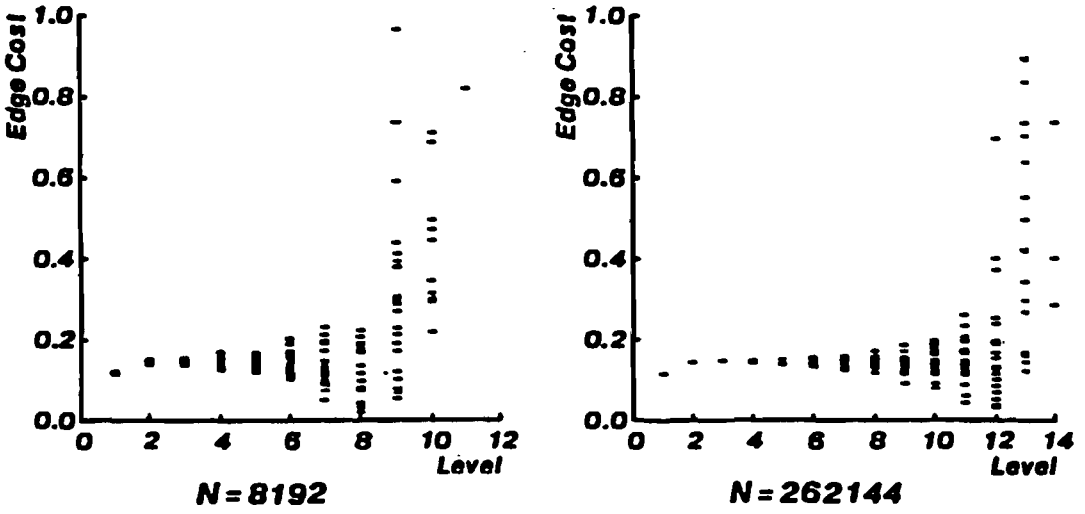


Exhibit 3-8: Edge Cost by Level

contained here requires deeper understanding of the distributional properties of point sets under nearest-neighbor removal.

References

[1] D. Avis.
A survey of heuristics for the weighted matching problem.
Networks 13(3):475-493, September, 1983.

[2] D. Avis, B. Davis, and J. M. Steele.
Probabilistic analysis of a greedy heuristic for Euclidean matching.
1988.
(To appear in Journal of Applied Probability).

[3] J. Edmonds.
Paths, trees, and flowers.
Canadian Journal of Mathematics 17:449-467, 1965.

[4] D. E. Knuth.
The Art of Computer Programming: Volume 2, Seminumerical Algorithms.
Addison-Wesley Publishing Company, Reading, MA, 1973.

[5] G. K. Manacher and A. L. Zobrist.
Probabilistic methods with heaps for fast-average-case greedy algorithms.
Advances in Computing Research: Computational Geometry.
F. P. Preparata, Ed., JAI Press, Greenwich CT, 1983, pages 261-278.

[6] E. M. Reingold and R. E. Tarjan.
On a greedy heuristic for complete matching.
SIAM Journal of Computing 10:676-681, 1981.

[7] M. J. Steele.
Personal communication.
1985.

Chapter 4
Comparisons in Quicksort

4.1. Introduction

Quicksort is among the most efficient of comparison-based sorting methods. Proposed by Hoare [2, 3] in 1960, the algorithm was thoroughly analyzed by Sedgewick [8, 6], who gave precise performance bounds for a number of implementation strategies. Knuth [5] also provides a detailed discussion.

To sort an array X of N elements, Quicksort partitions X around a *partition element*, s, so that elements with value less than s are to its left in the array and larger elements are to its right; after partitioning, s is in its correct position. The algorithm then recurs on the two subarrays on either side of s. Performance depends to a great extent upon the choice of partition element at each stage of the recursion. If, for example, the least element is selected each time, then Quicksort can require $\Omega(N^2)$ total comparisons during partitioning. Best performance is achieved if the list is divided in half at each stage. A number of efficient strategies for selecting a partition element with rank near the median have been examined.

A strategy that works well is to choose s at random at each stage. A generalization (suggested by Hoare) takes a sample of size T from the sublist and uses the median of the sample as an estimate of the true median (random selection corresponds to $T = 1$). Singleton [9] and Sedgewick [8] recommended *median-of-3* Quicksort (that is, $T = 3$) over a large class of selection strategies. Sedgewick showed that the percentage improvement in sorting cost is small for larger T and argued that the cost of finding the median of larger samples would quickly overtake any improvement in sorting cost. His argument was not made precise because the comparisons used in median selection were not an explicit part of his analytical model; instead, this cost was included as part of the overhead for each recursive call.

This chapter extends Sedgewick's analysis by explicitly including the cost of median

selection and by considering the trade-offs between partitioning and median-selection costs. Intuitively, larger sample sizes require more median-selection comparisons but produce better partitions, giving fewer partition comparisons. Section 4.3 presents simulation results that compare fixed-T strategies; although comparison cost is of primary interest, others measures are also considered. Section 4.4 examines strategies that allow T to vary as a function of sublist size at each recursive stage. Section 4.5 corrects a small error in Sedgewick's analysis of median-of-3 Quicksort.

4.2. Simulation Issues

It is not necessary to generate and sort random lists of numbers to obtain the desired measurements. A simple "shortcut" simulation program mimics the performance of Quicksort on random lists with much less computation time. Subsection 4.2.1 describes the model of Quicksort to be simulated, and Subsection 4.2.2 discusses implementation details of the simulation program.

4.2.1. The Model

Exhibit 4-1 gives the general structure of the simulation model. It is actually a combination of two sorting algorithms: "basic" Quicksort can be improved by not recurring on sublists smaller than some cutoff M. To sort the elements in the small sublists, a single final pass of Insertion Sort, which has low overhead, is used.

Assume that the N elements to be sorted consist of the integers 1 through N, initially arranged in some random permutation; this simplifies the discussion as well as the simulation since the rank of an element is identical to its value. Note that this implies that equal-valued elements do not occur in the input list; otherwise, this assumption does not affect the analysis of Quicksort because performance depends only on the ranks and not on the values of the elements. (See Sedgewick [7] for a discussion of Quicksort with equal elements.)

Quicksort is one of the few algorithms for which various implementation strategies have been analyzed exactly rather than in asymptotic order-of-magnitude terms. To do this, a set of measures are identified that correspond to the number of times various pieces of code are executed. The analysis remains independent of specific implementation because the measures correspond to the *number* of times various steps are performed rather than to their running times. The standard model incorporates the measures listed below; values at each

```
Procedure Quicksort              To sort array X[1..N] with cutoff M.
        Qsort(1, N)
        InsertionSort(1, N)

Procedure Qsort(lo, hi)
    If (hi-lo ≥ M)
        Sample T elements from X[lo,hi]
        Select the sample median, call it s
        Partition the array around s, and set j
                to the index of s in X
        Recur on the left and right subarrays:
                Qsort(j+1, hi)
                Qsort(lo, j-1)

Procedure InsertionSort(lo, hi)
     Set i = lo+1
     Loop until i = hi:
        If X[i] is smaller than X[i-1] then
                Sift down X[i-1] to X[1], shifting elements
                        and placing a in its correct sorted
                        position.
        Increment i
    Endloop
```

Exhibit 4-1: Quicksort

recursive stage correspond to Sedgewick's very efficient implementation of Quicksort [8, 6]. For a detailed discussion of the simulation model, the measures, and their analysis, see [8], [6], or [5] (Section 5.2.1).

- A: the expected number of times subroutine Qsort is called. At each recursive stage A is incremented by 1.

- B: the expected total number of *exchanges* performed during partitioning. At each recursive stage, this corresponds to the number of elements that must be moved in order to partition the subarray around the partition element s. If the array is of size $n =$ hi - lo + 1 and T is the sample size, then at each stage B the expected number of exchanges is given by

$$(n-1)(n-s)/\binom{n}{T}.$$

- C: the expected total number of *comparisons* of array elements to s during partitioning (but not during median-selection). In Sedgewick's efficient implementation C is incremented by $n-1$ at each recursive stage. This implementation requires a high-valued sentinel at the high end of array X. During the recursion previously-selected partition elements serve as sentinels for the subarrays.

- D: the number of *insertions* performed during Insertion Sort. In Program 4.2.1 this

corresponds to the number of times the conditional is true in the Insertion Sort procedure. D represents the sum, over the subarrays of size less than M, of the number of insertions in each. For each small subarray of size $n \leq M$ the number of insertions has expected value $n - H_n$.

- E: expected the number of *moves during insertion* (equivalently, the total distance items are shifted) performed by Insertion Sort. For each small subarray of size $n \leq M$, this has expected value $n(n-1)/4$. Like D, E represents the sum of this quantity over all small subarrays.

Sedgewick found closed forms (within an $O(N)$ term) for B and C for general fixed-T strategies when $M = 1$ (that is, no Insertion Sort is performed). He also derived exact formulas for A through E (within an $O(n^{-4})$ term) for $T = 3$ and arbitrary M. This study extends the standard model by explicitly counting the cost of median selection. In Sedgewick's model the sample size is fixed, so median selection is counted as part of the overhead of a recursive call, found by multiplying measure A by an appropriate constant. His analysis also assumes that $M \geq 2T$, which is not necessarily the case here.

Section 4.3 examines fixed-T strategies with varying M; Section 4.4 considers strategies where T is allowed to vary with the sublist size. Since the sample size is allowed to vary as a function of the subarray size at each stage, a general median-selection algorithm that takes the sample size as input is required. Let $t(n)$ be a function that returns an odd-valued integer in the range $[1, n]$: then $T = t(n)$ is the size of the sample taken from the n elements at a given recursive stage.

In Sedgewick's very fast implementation the sample elements are evenly-spaced over the subarray. General median-selection algorithms assume a *contiguous* set of elements, rather than an evenly-spaced sample of a larger set: how shall median-selection be embedded into Quicksort?

One strategy is to copy the sample elements to another array for median selection. Another is to perform median-selection *in place*, either by forming the sample from T contiguous elements in the middle of the subarray or by implementing a median-selection routine that accommodates noncontiguous elements (perhaps by using indirection in array addresses). This strategy avoids the overhead of copying elements between arrays. Another potential benefit of the in-place strategy is that the sample is correctly partitioned during median selection; a clever implementation of the (Quicksort) partitioning step might exploit this fact by not re-examining the sample.

Detailed evaluation of implementation possibilities would depend upon many factors specific to the environment; this (interesting) problem is more appropriate to a study that would measure program performance on a specific machine. The policy adopted for this simulation study was to keep it simple, to make as few assumptions as possible about implementation details, and to allow for the possibility of in-place median selection.

Exhibit 4-2 presents Select, a linear expected-time algorithm for selecting the median of array elements X[lo, hi], where T = hi-lo+1. The general algorithm for selecting the k^{th} largest of T items was first described by Hoare [2] (who called it Find) and was analyzed by Knuth (see [5], Problem 5.2.2-32). At termination of Select the median element lies in X[m].

Select is similar in structure to Quicksort except that only one side of the partition is considered in each iteration. In keeping with the theme of simplicity, a Median-of-1 (random sampling) strategy is adopted in the Sampling step. The model adopted for the partitioning step requires $n+1$ comparisons for each iteration on a subarray of size n, rather than the $n-1$ adopted for measure C (see [5] for a discussion of this partitioning model). This version requires no sentinel, however, which would complicate the model under the assumption of in-place median selection.

```
Procedure Select(lo, hi)                          T = hi - lo + 1
        m = ⌊(lo + hi)/2⌋  m = the index of the median
        loop
            i = lo; j = hi + 1
            Sample a random element s from X[lo, hi]
            Partition the array around s, and set j
                        to the index of s in X.
            Test: if (j < m) then lo = j + 1
                        else if (j > m) then hi = j - 1
                        else break;
        endloop
```

Exhibit 4-2: Median Selection

Let $f_{T,k}$ represent the number of *selection comparisons* required to find the k^{th} largest of T elements, and let $g_{T,k}$ represent the number of *selection exchanges*. Then the following recursions, with base cases $f_{1,k} = 0$ and $g_{1,k} = 0$, describe the computation time of Select (see [5]).

$$f_{T,k} = T+1 + \frac{1}{T}\sum_{s=1}^{k-1} f_{T-s,k-s} + \frac{1}{T}\sum_{s=k+1}^{T} f_{s-1,k}$$

$$g_{T,k} = \frac{1}{T}\sum_{s=1}^{k-1} \frac{(T-s)(s-1)}{(T-1)} g_{T-s,k-s} + \frac{1}{T}\sum_{s=k+1}^{T} \frac{(T-s)(s-1)}{(T-1)} g_{s-1,k}$$

Knuth [5] (Problem 5.2.2-32), solved $f_{T,k}$, showing that

$$f_{T,k} = 2[(T+1)H_T - (k+1)H_k - (T-k+2)H_{T-k+1} + T + 5/3] \qquad (1)$$

It appears that a closed form for $g_{T,k}$ has not been published. Let F represent the total number of comparisons performed during median selection.

When T is fixed and $M \leq T$, it is possible that a subarray is too small: how does one sample 5 elements from 3? Sedgewick's implementation samples *with replacement* in such a case, so that some of the elements are duplicated. Because this assumption is not compatible with that of sampling and selecting from contiguous elements the subarray, the elements are sampled without replacement. When T is greater than the subarray size n, T is set to n or $n-1$, whichever is odd. This implies that the true median is *always* found when n is odd and smaller than T, which may not be the case under Sedgewick's model. If T is less than the cutoff M, then this model is identical to Sedgewick's.

4.2.2. The Simulation Program

An obvious simulation strategy is to generate random lists of numbers and to sort them while recording the measures of interest. This would require $\Omega(N)$ steps to generate each list plus $\Omega(N \log N)$ steps to sort. A simple observation allows more efficient simulation; the "shortcut" implementation describe here is similar to a shortcut Bentley describes for a median-selection algorithm in [1].

Recall the assumption that an input list of size N consist of a random permutation of the integers 1 through N. When Qsort(lo, hi) is called, the subarray X[lo, hi] must therefore contain a random permutation of the integers lo through hi. Whatever the median-selection strategy, the partition element chosen must be from [lo, hi]. If the partition element is in X[k] after partitioning, then Quicksort recurs on the subarrays X[lo, k-1] and X[k + 1, hi].

Rather than *selecting* the partition element from the subarray at each recursive stage, the

shortcut simulation program *generates* a partition element from a hypothetical subarray according to an appropriate probability distribution. At each stage, the *size of the subarray* (equal to hi - lo + 1) and the *value of the partition element* (randomly generated by a procedure to be described shortly) are the only two quantities needed to accumulate the measures described earlier. For example, A, the number of stages reached, is simply incremented at each stage of the simulation program. If the partition element has rank s in a subarray of size n, then B, the number of exchanges, is incremented by

$$(s-1)(n-s)\Big/\binom{n}{T}.$$

The number of partition comparisons at this stage is $n-1$, and the number of selection comparisons was given earlier as formula (1) (with $k=(T+1)/2$, the median of T). Finally, the cost of Insertion Sort determined by the small subarrays. For a subarray of size $n < M$, measures D and E are incremented by $n-H_n$ and $n(n-1)/4$, respectively.

Note that the *expected* values of B, D, E, and F are accumulated at each level, rather than values of corresponding random variables. As a result, the variance for these four quantities is much smaller than would be displayed by Quicksort. Since only means are examined here, this "bug" becomes a feature: the small variance in experimental results means that few trials are needed. This is an application of a variance reduction technique discussed more fully in Section 7.3.

To obtain estimates for the expected values of A through F, then, it is sufficient to generate a partition element at each stage according to a probability distribution that is determined by the size of the subarray n and the sample size $T=t(n)$. The generation of such a partition element is easily accomplished in $O(t(n))$ time at each stage: simply generate a random sample of size T from $[1, n]$ and return the sample median. Two methods are employed in the simulation program, the choice depending on whether T is near n. When $T \ll n$, the simulation program simply generates integers uniformly T distinct ones appear; a hash table of size $2T$ is used to check for duplicates. The table uses an open addressing collision scheme, with the invariant that table entries are always in sorted order. Once generated, the T integers are shifted to the low end of the table and the median element, which occupies table position $m=(T+1)/2$, is returned.

When T is near n this method is inefficient because of the large number of duplicates generated before T distinct elements are found. Knuth [4] (Section 3.4.2) gives an algorithm for generating a sample of T integers from $1..n$ in *ascending order* by considering each integer

in turn and "accepting" it with appropriate probability. This algorithm is modified in the simulation to stop when the m^{th} integer is accepted.

This process requires about $T/2$ random number calls; as implemented, it is more efficient than the first method when $T \leq .03n$. Note that because the elements are maintained in sorted order (making it easier to find the median), both of these methods produce partition elements more efficiently than if median-selection had actually been performed on a random sample from the sublist. The second method is even more efficient because it only considers approximately $T/2$ elements.

```
Procedure Shortcut(n)
        if (n < M) then
                D += n - H_n                    Accumulate Insertion Sort measures.
                E += n(n - 1)/4
        else
                T = t(n)                        Determine sample size.
                m = (T+1)/2
                s = Generate-Partition-Element(n, T)

                A += 1                          Accumulate measurements.
                B += (n-s)(s-1) / Choose(n, T)
                C += n - 1
                F = 2((n + 1)H_n  - (m+1)H_m - (T-m+2)H_{T-m+1}+T+5/3

                Shortcut(s-1)
                Shortcut(n-s-1)

Procedure Driver
        Input N, M
        Set A through F to zero
        Shortcut(N)
        Report A through F
```

Exhibit 4-3: The Simulation Algorithm

The simulation program is sketched in Exhibit 4-3; the formulas for accumulating A through F reflect the model described in the previous subsection. Its average running time is given by the following recursion, where p_s represents the probability that s becomes the partition element when $t(n)$ elements are sampled. $Time_N = O(1)$ when $N \leq M$.

$$Time_N = O(t(N)) + 2\sum_{s=1}^{N} p_s Time_{s-1}. \qquad N > M$$

$$p_s = \binom{N-s}{(t(N)-1)/2}\binom{s-1}{(t(N)-1)/2}\Big/\binom{N}{t(N)}.$$

Finding a closed form for this recurrence for arbitrary $t(n)$ is difficult. When $t(n)$ is a constant function, the program runs in time linear in N.

The experiments described in the following sections were performed on a VAX 11/780 running under Unix[1]. The random number generator was the cyclic feedback method described by Knuth [4] (Algorithm A, Section 3.2.2, 2nd Edition). In most of the following, N is set at powers of two from 2^5 to 2^{15}, M is set at 0, 10, and 20, and T (in Section 4.3) is set at 1, 3, and 5.

To check some experimental results, the exact values of measures A through E were computed according to Sedgewick's formulas [8, 6] for Median-of-3 Quicksort. When a small error was found in one of the formulas (see Section 4.5), a dynamic program was implemented to check the formulas as well as the simulation program. For example, to compute C exactly by dynamic programming, array elements $C[n]$ are set to 0 for $n < M$, since no Quicksort comparisons are performed at n below the cutoff. The following sum is then computed for $n \geq M$.

$$C[n] = \sum_{s=1}^{n} \frac{(n-s)(s-1)}{\binom{n}{3}}((n-1) + C[s-1] + C[n-s])$$

In each term of the summation the quotient represents the probability that s is chosen as the median of three elements selected randomly from $[1, n]$. The terms in parentheses represent the cost associated with chosing s, which is given by the cost at this level $(n-1)$ plus the expected cost of partitioning around s. To further check the random number generator in the simulation program, this dynamic programming approach was extended to $T = 1, 3, 5$, and 7 for quantity A. In all tests of the simulation program, observed means for the measures were within 1.5% of the true means produced by the dynamic program.

Instead of performing simulations, why not just use the dynamic programs to produce exact

---

[1] VAX is a trademark of Digital Electric Corporation. Unix is a trademark of AT&T Bell Laboratories.

values for the measures? The main drawback of the dynamic program is inefficiency: since the probability of s being chosen must be recomputed for each $s \leq n$, the running time of the program is linear in each n and therefore quadratic in the highest n computed. Because the running time of the simulation program is linear in N for (fixed T) and the variance in simulation results was small (requiring few trials per sample point), experiments provided a much more efficient way to gather results for large problem sizes and many sample points.

4.3. Fixed-T Strategies

This section presents simulation results for median-of-T Quicksort for T fixed at 1, 3, and 5. A sample point is determined by N, M, and T. Most of the simulations were performed at sample points corresponding to $N = 2^8, 2^{10}, \ldots 2^{15}$, $M = 1, 10, 20$, and $T = 1, 3, 5$; measures A through F are considered. An obvious problem in describing simulation results arises: how to represent a function of three variables using two-dimensional graphs? A number of approaches to this problem are considered in Chapter 7; for this study, the following conventions are adopted. Measurements for the three M settings appear in separate panels. In each panel the x-coordinate of a point is determined by $\log_2 N + T/10$, and the y-coordinate of each point corresponds to the specified measure (giving the mean over 20 trials at this sample point). For example, in the left panel of Graph 4-4-a sample point $M = 1$, $N = 2^8 = 256$, $T = 1$ is represented by the leftmost cross in the panel. The second-leftmost point corresponds to the sample point $M = 1$, $N = 2^8$, $T = 3$, and is plotted with x-coordinate 8.3. This method of "coding" the x-coordinate of each point allows easy comparison of the measures at in terms of the simulation parameters.

Let C' represent the average *total* number of comparisons required in fixed-T Quicksort; that is, $C' = C + F$. Intuitively, C should decrease with T, because larger samples give better partitions, reducing the number of partition comparisons. On the other hand, F should increase with T since larger samples require more median-selection comparisons. The best choice of T – that is, the choice that minimizes total – is one that finds the right balance between these two measures.

Graph 4-4-a depicts C'/N for the sample points given above. Not surprisingly, large M (right panel) gives fewer comparisons at every sample point since the cost of Insertion Sort is ignored. The reduction in Quicksort comparisons at high M must of course be balanced against the increase in Insertion Sort time. This would be an important task in determining the best choice of M for a specific implementation.

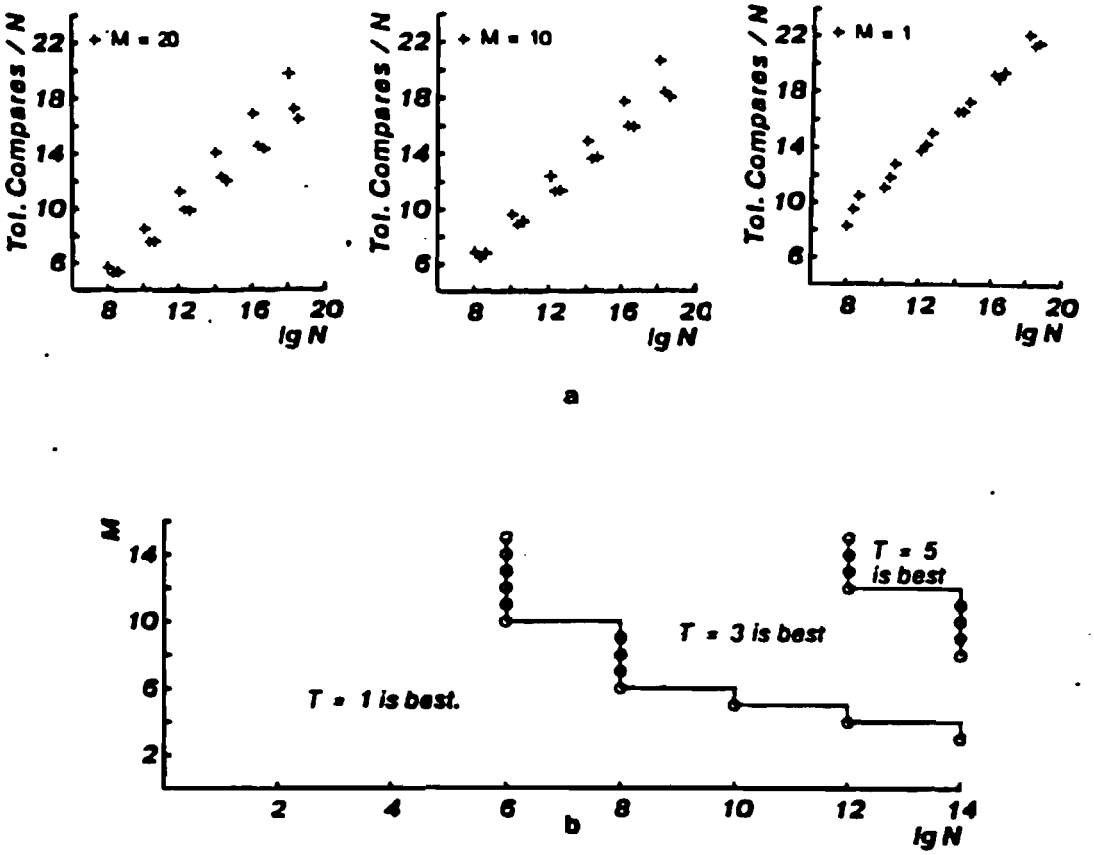


Exhibit 4-4: Total Comparisons

The left panel in Graph 4-4-a corresponds to $M = 1$; that is, to Quicksorting the entire array and not performing Insertion Sort at all. In this panel, the leftmost in each triple of crosses has least value when $N \leq 2^{14}$, and the middle cross has least value when $N > 2^{14}$. This indicates that the Median-of-1 strategy gives fewest total comparisons when N is small, but Median-of-3 is best when N is large (within the range of the experiments). If a cutoff of size $M = 10$ is used (middle panel), then Median-of-3 gives fewest total comparisons for N smaller than 2^{16}, and Median-of-5 gives fewest total comparisons at higher N. If $M = 20$ (right panel), then Median-of-5 is best for all but the lowest value of N sampled. Note that the separation in cost between Median-of-1 and the other two strategies becomes more pronounced at large M.

Graph 4-4-b presents the results of further simulations to determine the best choice of T for various combinations of N and M. Each region of the graph corresponds to the T setting that

gives the smallest mean value of C' for N at even powers of 2 and M below 16, 20 trials each. For example, when either M or N is low (bottom left region), Median-of-1 Quicksort gives fewest total comparisons. Note that the x-scale corresponds to quadrupling N each time whereas the ordinate corresponds to unit increases in M.

Consider the distribution of subarray sizes that appear during the recursion: a single array of size $n=N$ appears, then two arrays of size approximately $n=N/2$, then four arrays of size approximately $n=N/4$, and so on. As N grows, larger subarrays appear in the distribution but small subarrays become more numerous. Suppose that the choice of T to minimize total comparisons at a given recursive stage is an increasing unbounded function of the subarray size. Intuitively, with large subarrays at the beginning of the recursion a large sample size is the best choice, and smaller samples are more appropriate for small subarrays at lower recursive levels.

If the best choice of T grows fast enough in n, then as N (the problem size) increases, the large subarrays would eventually overcome the small subarrays in "voting" for the best choice of T. When M is greater than 1 the small subarrays are ignored, so the influence of the large subarrays is seen earlier in N. An implication of this argument is that the choice of T to minimize total comparisons (over the set of fixed T strategies) does not have a constant upper bound, but rather increases with N (and the rate of increase is determined by M). This contradicts Sedgewick's argument that Median-of-3 is probably the best choice among fixed-T strategies when $M = 1$. On the other hand, Graph 4-4-b suggests that Median-of-3 (or Median-of-1) is indeed the best choice over a large range of practical input sizes. Another implication is that a strategy which varies the sample size at each level according to the subarray size would give fewer total comparisons than any fixed-T strategy. Evidence that this is the case is presented in Section 4.4.

Consider how C' is divided between C and F. Sedgewick showed that for any fixed T, the number of comparisons during partitioning is given by

$$C=\frac{1}{H_{T-1}-H_{(T+1)/2}}(N-1)H_N+O(N)=O(\mathrm{Mog}\,N)$$

when $M = 1$. Graph 4-5-a presents $C/(\mathrm{Mog}\,N)$. Not surprisingly, C decreases as M grows (since the cost of Insertion Sorting small subarrays is ignored). Although the logarithmic x-scale in each panel makes the curves appear to grow more steeply than they actually do, the asymptotic constant is not easily seen at these values of N. As predicted, the triples of crosses show that C decreases in T: larger sample sizes tend to give better partitions, decreasing the total number of partition comparisons.

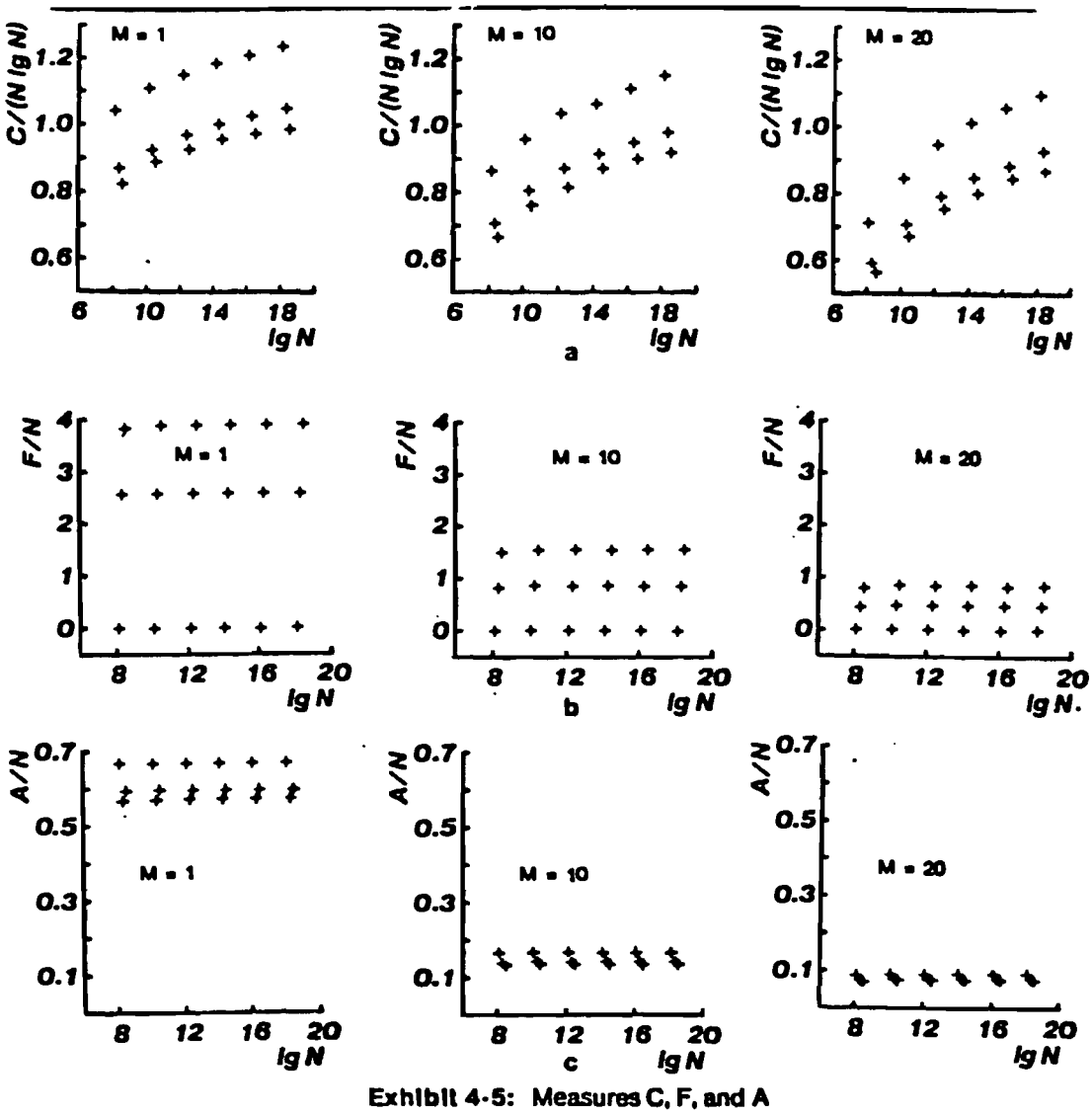


Exhibit 4-5: Measures C, F, and A

Graph 4-5-a shows that for any combination of value of M and N, the $T = 1$ strategy (the leftmost symbol in each triple) gives a significantly higher value of C than $T = 3$ or $T = 5$. Although Median-of-5 gives the lowest value for C everywhere, the improvement over Median-of-3 is never very great. This agrees with Sedgewick's observation that while Median-of-3 gives a substantial improvement over Median-of-1 for this measure, the percentage improvement at higher values of T is small.

For fixed T the total number of median-selection comparisons, F, must be proportional to the number of times the median selection routine is performed. Therefore, F must be proportional to A, the number of recursive stages seen during Quicksort. Specifically, for $m = (T+1)/2$, $F = f_{T,m}(T) \cdot A$, where $f_{T,m}(T)$ is given by formula (1) on page 72. Since the number of recursive stages reached is linear in N, F is also linear in N. Graph 4-5-b presents F/N. When $T = 1$ no median-selection is performed, so the leftmost cross in each triple is always equal to 0. As predicted, F tends to increase in T. In both 4-5-a and 4-5-b the difference between $T=1$ and $T=3$ is more pronounced than the difference between $T=3$ and $T=5$, although one measure increases in T and the other decreases in T.

Graph 4-5-c depicts A/N. The graph indicates that when $M = 1$, A is not monotonic in T. That is, median-of-3 Quicksort gives *fewer* stages, on average, than median-of-5 Quicksort. The explanation for this behavior is to some extent an artifact of the simulation model. Consider the case $n = 5$. Median-of-5 selection would find the *exact* median of the elements and recur on two subfiles, each of size 2. Remaining recursive calls would be on subarrays of size 1 and 0, which are below the cutoff M and do not contribute to A, so $A = 3$. In contrast, for $T = 3$ the sample might not produce the true median: with probability 6/10, either 2 or 4 is chosen. In such a case, Quicksort would recur on a subarray of length 1 and a subarray of length 3, and afterwards on two subarrays of size 1 (which contribute zero cost), for a total cost of 2. Under this strategy $A = (6/10) \cdot 2 + (4/10) \cdot 3 = 2.4$.

For arrays of length 5, then, Median-of-3 produces fewer stages than Median-of-5. This inequality propagates in the computations of A at higher N to give the nonmonotonic behavior observed in Graph 4-5-c. This relationship between Median-of-3 and Median-of-5 would disappear if $M = 2$, but an analogous relationship would then hold for $T=7$ and $T=9$. This observation can be generalized to find a similar pair of T values for any M.

By this cost metric it is not always a good idea to find the exact median of the sublist: if the cost of recurring on a sublist of size k is equivalent to that for a sublist of size $k+1$, then it is better to break a list of size $2k+1$ into a $k+1$-sized piece and a $k-1$-sized piece rather than into two pieces of size k.

4.4. Choosing T to Minimize Comparisons

In this section T is allowed to vary as a function of n, the sublist size at each level; the goal is to determine the optimum sample size for each n. "Optimum" means "the sample size that minimizes the total expected number of comparisons." Recall that $n-1$ comparisons are needed to partition a subarray of size n. Also, the number of comparisons required to find the median of $T = f(n)$ elements, for T an odd integer and $m=(T+1)/2$, is

$$f_{T,m} = 2[(T+1)H_T - (m+1)H_m + (T-m+2)H_{T-m+1} + T + 5/3].$$

Let C represent the *minimum* expected total number of comparisons; that is, C is the number of comparisons required (during partitioning and median selection) when the optimum sample size is chosen at each recursive stage. Letting $h = \lfloor T/2 \rfloor$ for notational convenience, we have

$$C(N) = N-1 + \min_T \left(f_{T,m} + 2 \sum_{s=h+1}^{N-h} p_s C_{s-1} \right), \qquad N > M,$$

where $$p_s = \binom{N-s}{h}\binom{s-1}{h} \Big/ \binom{N}{T}$$

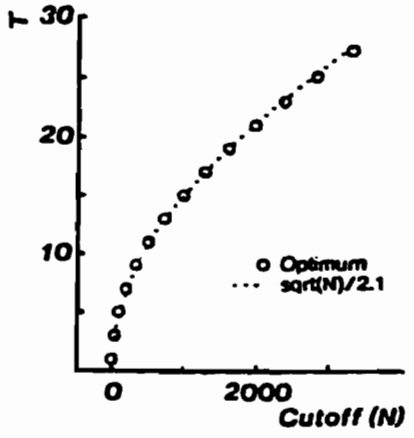
The $N-1$ term represents the cost of partitioning at each level. The summation index s ranges over all possible values for the partition element (as the median of T elements, s cannot be less than h or greater than $N-h$). p_s represents the probability that s was chosen as the median of T elements from N. $C(N) = O(1)$ when $N \le M$.

Let $t(n)$ represent the sample-size function that realizes the minimum total cost. A simple dynamic program can be used to determine $t(n)$ as well as C for small n. When $M = 1$, the boundary conditions are $C(1) = 0$ and $t(1) = 0$; for increasing n, the program searches for $t(n)$ to minimize the above function. Since $t(n)$ is nondecreasing and at most linear in n, it only necessary to check $t(n-1)$ against $t(n-1)+2$ for each n.

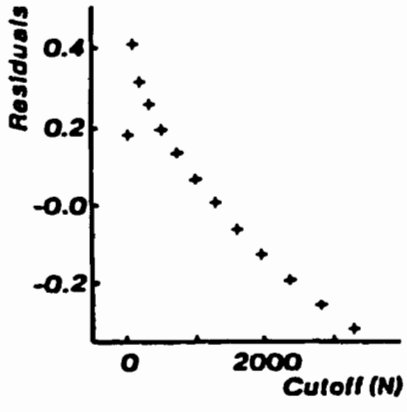
Table 4-6-a gives the lower boundary value of n corresponding to each $t(n)$, for $n \le 3500$. The second and third rows indicate, for example, that 3 is the optimum choice of T for subarrays of size between 35 and 92. Graph 4-6-b presents the table entries in graphical form; the dotted line corresponds to the function $N^{1/2}/2.1$. The residuals, representing the difference between the optimal values and this function, are presented in Graph 4-6-c. Similar computations for $M = 10$ differ from Table 4-6-a in that $t(3)$ has lower bound 30 rather than 35; otherwise the table entries are identical.

Least N	t(·)
1	1
35	3
93	5
197	7
337	9
515	11
730	13
984	15
1274	17
1603	19
1968	21
2372	23
2813	25
3291	27

a



b



c

Exhibit 4-6: Cutoffs for t(n)

Exhibit 4-7-a compares C to C' (corresponding to the fixed-T strategies), for the four sample points $N = 256, 1024$ and $M = 1, 10$. In each triple, the circles gives the ratio C'/C for $T = 1, 3, 5$, respectively. As predicted in the previous section, a strategy that modifies the sample size according to subarray size gives fewer total comparisons than any of the fixed-T strategies at these four sample points.

While these results are encouraging, it is difficult to determine t(n) for higher n. When $N = 2500$, for example, the dynamic program must compute the combination "2500 choose 24": although careful programming could push the computation higher, machine precision becomes a significant factor. Solving the problem analytically also seems to be difficult, requiring a solution to the recursion for C with arbitrary function $t(n)$ to find the $t(n)$ that minimizes C.

On the other hand, the results for small n can guide the search for good functions $t(n)$. Graph 4-6-c depicts the difference between t_n and the fit $Y = \sqrt{n}/2.1$; the small magnitude of error suggests that square-root form for $t(n)$ might do a good job, although the steady decrease in differences indicate that a more slowly-growing function might be needed at higher N. Preliminary simulations using the "odd floor" of $\alpha N^{1/2} + \beta$ (that is, the largest odd

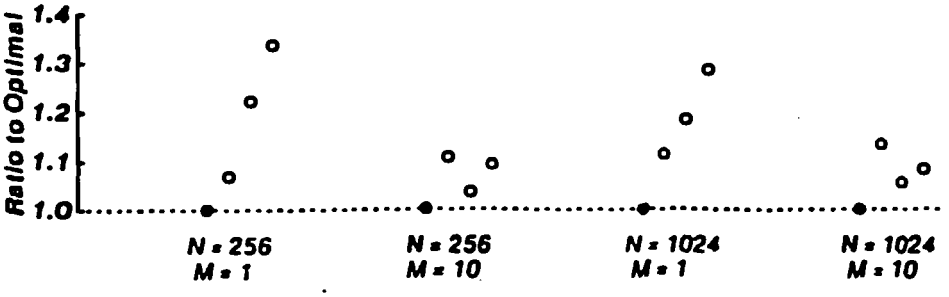


Exhibit 4-7: Optimum and Fixed-T Strategies

integer less than or equal to the function value) indicate that C and F are fairly insensitive to small differences in the cutoffs. For N at 2^{12} and 2^{16}, mean comparisons are minimized for square-root forms when $\alpha = 0.5$ and $\beta = 0$.

4.5. Insertion Sort

Sedgewick's [8] achievement was not only to present a very efficient implementation of Quicksort, but to demonstrate that it was more efficient than many alternatives by deriving exact formulas for quantities A through E. His final version employs such techniques as removal of tail-recursion, loop unrolling, careful ordering of conditionals, and fine-tuning of parameters. It uses a Median-of-3 selection strategy with a cutoff for Insertion Sort at $M = 9$.

One of my first tasks in building the simulation program was to check the experimental results against Sedgewick's formulas. This led to the discovery of an error in Sedgewick's analysis of quantity D, the number of insertions performed during the Insertion Sort phase. For median-of-3 Quicksort, D has the recursive form

$$D_N = 2\sum_{s=1}^{N} p_s D_{s-1} \qquad N > M,$$

where $$p_s = \frac{(N-s)(s-1)}{\binom{N}{3}},$$

$$D_N = N - H_N, \qquad N \le M$$

Sedgewick, in his thesis, gave the solution to this recurrence as

$$D_N = (N+1) - \frac{4(N+1)}{7(M+2)}(3H_{M+1} - 1).$$

This solution is incorrect: the last term should be $(3H_{M+1} + 1)$. Although this is clearly a minor algebraic error, the resulting computation of D is off by a factor of N and gives an erroneous calculation of the optimum value of M. A later paper by Sedgewick [6] gives the correct solution but fails to carry the correction through to the calculation of M. For the record, the correct derivation is given here.

Sedgewick showed that recurrences of the form

$$Y_N = y_N + 2\sum_{s=1}^{N}(N-s)(s-1)\binom{N}{3}^{-1} Y_{s-1}$$

can be broken into three simpler recurrences. For D_N we have $y_N = 0$, and it is (only) necessary to solve

$$(N+1)U_{N+1} = N \cdot U_N.$$

$$(N+1)T_{N+1} = (N+2)T_N + U_N. \quad \text{and}$$

$$(N+1)D_{N+1} = (N-5)D_N + T_N.$$

These have as base cases,

$$T_{M+1} = (M+2)D_{M+2} - (M-4)D_{M+1}.$$

$$T_{M+2} = (M+3)D_{M+3} - (M-3)D_{M+2}. \quad \text{and}$$

$$U_{M+1} = (M+2)T_{M+2} - (M+3)T_{M+1}.$$

Tedious but straightforward calculations produce the following solutions for D_{M+1}, D_{M+2} and D_{M+3}.

$$D_{M+1} = M + 5/3 - 2H_{M+1}$$

$$D_{M+2} = \frac{3M^2+14M+10}{3(M+2)} - 2H_{M+1}$$

$$D_{M+3} = \frac{3m^3+26M^2+61M+60}{3(M+2)(M+3)} - \frac{2M^2+10M+24}{(M+3)(M+2)}H_{M+1}$$

Therefore,

$$T_{M+1} = 7M + 10 - 12H_{M+1}$$

$$T_{M+2} = \frac{(M+3)(7M+10-12H_{M+1})}{(M+2)}$$

$$U_{M+1} = 0$$

and the three recurrences have the following solutions:

$$U_N = 0$$

$$T_N = \frac{(N+1)(7M+10-12H_{M+1})}{(M+2)}$$

$$(N+1)D_{N+1} = (N-5)D_N + \frac{(N+1)(7M+10-12H_{M+1})}{(M+2)}$$

Solving D_N from this point is relatively straightforward. Multiplying by the "summing factor" $N(N-1)\_(N-4)/6!$ and rewriting for the base term D_{M+1} gives

$$\binom{N+1}{6}D_{N+1} = \binom{M+1}{6}D_{M+1} + \frac{7M+10-12H_{M+1}}{(M+2)}\sum_{k=M+1}^{N}\binom{k+1}{6}.$$

or finally,

$$D_N = (N+1) - \frac{4(N+1)}{7(M+2)}(3H_{M+1}+1).$$

Sedgewick's efficient MIX implementation of Quicksort has average running time

$$(53/2)A + 11B + 4C + 3D + 8E + 9S + 7N.$$

Solving with the correct formula for D and regrouping terms gives

$$-\frac{47}{2} + \frac{372}{35}(N+1)H_{N+1}$$
$$+(N+1)\times\left[\frac{529}{49} + \frac{18}{M+2} - \frac{372}{35}H_{M+2} - \frac{36}{7(M+2)}H_{M+1} + \frac{48}{35}M + \frac{27(5M+3)}{7(2M+3)(2M+1)}\right],$$

which differs from Sedgewick's formula in the first and fourth terms (which he has as $-111/2$ and $-450(N+1)/(7(M+2))$, respectively). Exhibit 4-8 shows $F(M)$, the function defined by the terms inside the large brackets, as M varies. This graph shows that the average running time of Sedgewick's MIX implementation is minimized when $M = 7$, not 9 as reported.

M	F(M)
1	-1.832653
2	-5.028571
3	-6.799184
4	-7.819406
5	-8.378633
6	-8.626538
7	-8.650244
8	-8.504810
9	-8.227213
10	-7.843478
11	-7.372606
12	-6.828899
13	-6.223390
14	-5.564770
15	-4.860008
16	-4.114780
17	-3.333769
18	-2.520887
19	-1.679433
20	-0.812216

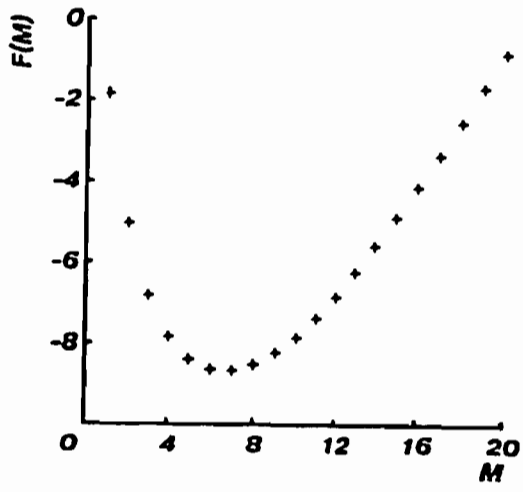


Exhibit 4-8: Minimizing F(M)

4.6. Conclusions

This chapter presents a version of Quicksort that allows sample size for median selection to vary with sublist size, with evidence that it outperforms fixed-T schemes. The tradeoffs between partition comparisons and median-selection comparisons are also examined for fixed-T strategies. A number of open problems remain. It appears to be difficult to find a closed form for C, and even harder to derive the $t(n)$ that minimizes the number of comparisons. Although comparison cost was the primary measure in the simulations, the behavior of other measures is also of interest. The number of exchanges performed during partitioning and median-selection is a measure of interest. The number of exchanges performed by the median-selection algorithm appears to be an open problem, ripe for experimental study.

An obvious next step is to study the actual running time of a "square root" strategy Quicksort. It is likely that the taking of square roots would dominate the computation time, in which case either a table-lookup scheme or a fast integer approximation of the square root function could be used.

References

[1] J. L. Bentley.
Progamming Pearls: Selection.
Communications of the ACM 28(11), November, 1985.

[2] C. A. R. Hoare.
Partition (Algorithm 63), Quicksort (Algorithm 64), and Find (Algorithm 65).
Communications of the ACM 4(7):321-322, July, 1961.

[3] C. A. R. Hoare.
Quicksort.
Computer Journal 5(4):10-15, April, 1962.

[4] D. E. Knuth.
The Art of Computer Programming: Volume 2, Seminumerical Algorithms.
Addison-Wesley Publishing Company, Reading, MA, 1973.

[5] D. E. Knuth.
The Art of Computer Programming: Volume 3, Sorting and Searching.
Addison-Wesley Publishing Company, Reading, MA, 1973.

[6] R. Sedgewick.
Analysis of Quicksort programs.
Acta Informatica 7(4):327-355, 1977.

[7] R. Sedgewick.
Analysis of Quicksort with equal keys.
SIAM Journal of Computing 6(2):240-267, June, 77.

[8] R. Sedgewick.
Quicksort.
PhD thesis, Stanford, 1975.

[9] R. C. Singleton.
An efficient algorithm for sorting with minimal storage (Algorithm 347).
Communications of the ACM 12(3):185-186, March, 1969.

Chapter 5
Self-Organizing Search

A self-organizing sequential search rule maintains a *search list* of N items so that frequently accessed items are near the front. Since access frequencies are assumed not to be known in advance, the rule is allowed to modify the ordering of the search list according to the *request sequence* of previous accesses. The class of *Move-Ahead-k* rules is studied here: when an item is requested, it is moved forward k positions in the search list (or to the first position if already less than k from the front), for $1 \le k \le N-1$. The rules Move-Ahead-1 and Move-Ahead-(N-1) are better known as Transpose and Move-to-Front, respectively.

For convenience, let the items be named 1 through N. A common theoretical model assumes a sequence of T *requests* for items in the search list: the request sequence is formed by drawing item names randomly and independently according to the probability distribution $P_N = \{p_1, p_2, \ldots p_N\}$. That is, the probability that "3" is the next item requested is given by p_3. Assume without loss of generality that $p_i \ge p_{i+1}$.

The *request cost* is equivalent to the distance of a requested item from the front of the search list: the first item has request cost 1, and so on. The simulations described here measure expected request cost for various rules assuming a fixed probability distribution on the request sequence. The cost of reordering the list after each request, which is bounded above by the request cost, is not measured here. Sequential search rules have been studied for almost two decades; previous work is briefly surveyed in the following section. Section 5.2 discusses simulation details, Section 5.3 presents experimental results for expected search cost, and Section 5.4 considers properties of the search list permutations for various rules.

5.1. Previous Work

Most previous analyses have considered sequential search as a Markov process: each search list permutation corresponds to a state and the state-transition probabilities are derived from request probabilities. For a given rule and probability distribution on requests, the *asymptotic expected search cost* is the sum over all search-list permutations of the product of expected search cost of a permutation and its steady-state probability. The expected search cost of a permutation is the sum, over all items, of the product of request probability for each item and its position in the permutation.

For a given distribution P on request probabilities let $M(k,N,T)$ denote the expected search cost of the Move-Ahead-k rule for a search list of size N after T requests have been made. Let $M(k,N)$ denote the expected cost as $T \rightarrow \infty$.

Exhibit 5-1 presents previous results for the expected costs of Transpose ($k = 1$) and Move-to-Front ($k = N - 1$). The rules are compared to the Optimal Static (OS) rule which maintains the items in the search list by decreasing request probability. This rule knows the request probabilities in advance and never reorders the search list. All results in Exhibit 5-1 hold for arbitrary probability distribution $P = \{p_1, p_2, \ldots p_N\}$ and all assume that the search list is initially in random order. The first, for the Optimal Static rule, is easily derived. The second appears in [4], [6], [9], [10], [11], and [12]. Formula (3) is due to Bitner ([3], [4]). Formula (4) is from Chung, Hajela, and Seymour [7], and formulas (5) and (6) are found in [12].

Exhibit 5-1 can be summarized as follows. The asymptotic expected search cost of Move-to-Front is never more than $\pi/2$ times that of Optimal Static (by formula (4)) and the expected cost for Transpose is never more than that for Move-to-Front (by (5)). Note that computing the expected asymptotic cost for Transpose requires summing over the $N!$ permutations of the search list (6). The rate of convergence to asymptotic performance is given for Move-to-Front by the summation term of formula (3). Bitner [3, 4] showed that even though Transpose has better asymptotic cost than Move-to-Front, the latter converges more quickly.

The rules have also been analyzed for specific probability distributions on the requests, especially Zipf's Distribution (also known as Zipf's Law), which is defined by $p_i = 1/(iH_N)$, where

$$OS(N) = \sum_{i=1}^{N} ip_i. \quad (1)$$

$$M(N-1,N) = 1/2 + \sum_{i,j=1}^{N} \frac{p_i p_j}{p_i + p_j}. \quad (2)$$

$$M(N-1,N,t) = M(N-1,N) + \sum_{1 \le i < j \le N} \frac{(p_i - p_j)^2}{2\cdot(p_i + p_j)} \cdot (1-p_i-p_j)^t. \quad (3)$$

$$M(N-1,N) \le \pi/2 \cdot OS(N). \quad (4)$$

$$M(1,N) \le M(N-1,N). \quad (5)$$

$$M(1,N) = Pr(I_N)^{-1} \sum_{\pi} \prod_{i=1}^{N} p_i^{1-\pi(i)} \sum_{j=1}^{N} p_j \cdot \pi(j). \quad \text{where} \quad (6)$$

π = an ordering of the search list,
$\pi(i)$ = the position of i in π,
$Pr(I_N)$ = probability of the optimal ordering occurring initially

Exhibit 5-1: Previous Results

$$H_N = \sum_{j=1}^{N} 1/j.$$

H_N is known as the N^{th} harmonic number and grows approximately as the logarithm of N. A family of distributions related to Zipf's will also be studied; define the distribution Z^{λ} by

$$p_i = \frac{1}{c \cdot i^{\lambda}},$$

where $c = H_N^{\lambda} = \sum_{j=1}^{N} 1/j^{\lambda}$.

Setting $\lambda = 1$ gives Zipf's Distribution, and $\lambda = 0$ gives the uniform distribution. When the request probabilities correspond to Zipf's, the asymptotic average search cost for the Optimal Static rule is N/H_N, and the average cost for Move-to-Front is about 1.386 times this (see [10]). Gonnet, Munro and Suwanda [8] give closed forms for $M(N-1,N)$ when $\lambda \le 2$.

Some worst-case bounds also exist. Since the worst-case cost per request is trivially N, the *amortized* cost (the average cost over a worst-case sequence of requests) is used. Bentley

and McGeoch [2] showed that for any request sequence the amortized cost of Move-to-Front is at most twice that for Optimal Static. In contrast, the ratio of search cost for Tranpose to that for Optimal Static can be arbitrarily high. Sleator and Tarjan [13] showed that under a slightly different cost model, Move-to-Front has amortized cost a most twice that for *any* rule, static or dynamic.

No expected-case bounds are known for general Move-ahead-k rules. Bitner [4], Gonnet, Munro and Suwanda [8], and Rivest [12] have conjectured that for any two rules in this class, the one with lower index will approach its asymptote more quickly *and* the other will have lower asymptotic cost. (Bitner demonstrated this for the special cases of Transpose and Move-to-Front.)

Move-Ahead-k rules have been studied experimentally with Zipf's Distribution describing the request sequence. Rivest [12] presented simulation results at the sample points $N = 7$, $T = 5000$, $k = 1, \ldots, 7$ to support the above conjecture. Tenenbaum [14] measured average search cost for k ranging from 1 to 7, N from 3 to 230, and T at 12,000. Although he uses a slightly different model in accumulating costs, his tables of average search costs suggest the best choice of k for each N within this range.

5.2. Measures of Search Rules

A practical experimental approach is to generate a sequence of requests and to record the *request cost* of searching for requested items. This is the measure used in previous simulation studies of sequential search rules (see [5], [1], [12], and [14]). An alternative measure is described in this section.

For a fixed permutation, the cost of searching for the next request is a random variable depending upon the current search list permutation and the request probabilities. The *expectation* of this random variable is equivalent to the expected search cost of the permutation: call this expectation the *permutation cost* of permutation π. Permutation cost as well as request cost is an unbiased estimator of expected search cost at time T. In addition, permutation cost is guaranteed to have smaller variance than request cost; see Section 7.3 for further discussion of this idea. Many simulation studies may be improved by replacing random variables (e.g. the request cost for permutation π) by their expectations (the permutation cost).

For the simulations described in this chapter, permutation cost at time T is measured rather than request cost. Otherwise the simulation programs are straightforward implementations of the search rules: at every request, the routine for each rule records the permutation cost and then reorders its search list according to the requested item. The search lists are all initialized to the same (randomly chosen) permutation.

The request cost at time T can be computed in time proportional to the position of the requested item. As noted in Section 5.1, this cost has asymptotic expectation $1.386N/H_N$ for Move-to-Front when requests are generated by Zipf's Distribution.

The permutation cost at time T can also be computed in time proportional to the position of the requested item by keeping a list of summary information with the search list. The second list records cumulative permutation costs counting from the *rear* of the search list. That is, if $s(i)$ denotes the name of the i^{th} item in the search list and $p_{s(i)}$ its probability of being requested, then the j^{th} entry in the summary list contains the sum

$$\sum_{i=j}^{N} s(i) \cdot p_{s(i)}.$$

When a requested item is found at position i in the search list and the appropriate search list permutation is performed, only the information in the first i positions of the summary list need be changed, requiring time proportional to the cost of searching the list. The permutation cost for the entire search list is found in the first position of the summary list.

This use of a secondary array was not discovered in time for the simulation study; in the simulation programs, the cost of each permutation was computed by summing over the search list (requiring linear time). The running time of the simulation routine for each rule was therefore increased (for Zipf's Distribution) by a factor of between $H_N/1.386$ (for Move-to-Front) and H_N (for Optimal Static, a lower bound on costs for Move-ahead-k rules). For this approach to be practical, the variance in permutation cost must be at least this much less than the variance in request cost. Experimental evidence suggests that this is bound was easily met in the simulation.

Permutation cost must be summarized in some way: when $T = 1{,}000$, say, it is difficult to manipulate or display the 50,000 numbers that would be generated over 50 trials Exhibit 5-2 depicts two possible summarization schemes. For a hypothetical search rule at fixed N, the *plusses* in each graph represent the search cost at time T, for T ranging from 1 to 25. Previous simulation studies have taken running averages of search costs, represented by the

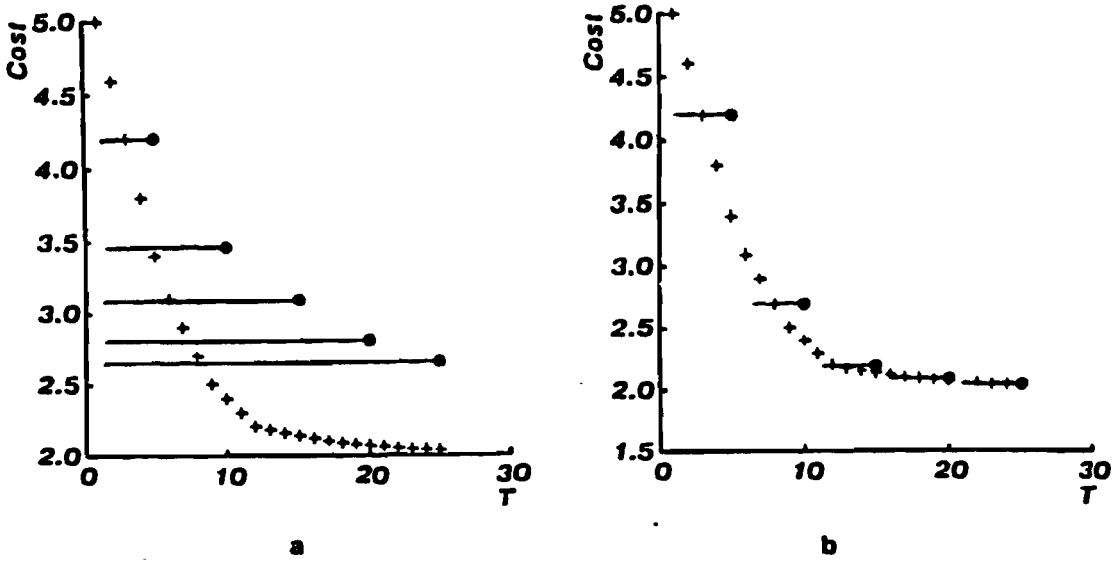


Exhibit 5-2: Two Summarization Methods

circles in Graph 5-2-a, where a new running average is reported every fifth request. This approach is not entirely satisfactory in capturing true search costs: since the search rules are characterized by high initial cost, the running averages consistently overestimate average search cost at time T.

Graph 5-2-b displays *grouped averages* (also called *batched means*). In this graph, each circle represents the mean of the previous 5 requests only, rather than all previous requests. These means give a more accurate measure of average search cost over time because they are less influenced by initial costs. In the following section batched means are used rather than running averages to summarize search costs. The parameter G denotes the group size for a particular experiment (in Graph 5-2-b for example, $G = 5$). Means are taken over all trials for each group. At the sample point $N = 10$, $T \leq 100$, $G = 5$, for example, there are 20 groups per trial, each containing 5 measurements; if 25 trials are taken, then each group average represents the mean of 125 measurements.

The experiments are paired in the sense that in each trial the same request sequence is submitted to all rules. For each trial, the search lists are initialized to a random order, identical for each rule. Random request sequences are generated by the method of aliasing (see Section 7.7). The simulation program requires $O(N)$ setup time (to initialize the search lists and the random variate generator), constant time to generate a request, $O(N)$ time per

search rule to compute permutation cost per request, and $O(k)$ time per search rule to reorder the search list.

A sample point is determined by k, N, T, λ and G. Because of high variance in the data (even though variance was reduced by the new measure), 50 to 100 trials were taken at various sample points. For efficiency and manageability of the results, k was only set to odd values $1, 3, \ldots N-1$. In most of the following experiments N was set at 6, 8, 10 and the parameter λ (determining request probabilities) at 0, 0.5, 1, 1.5, 2. The largest T value used was 2000 and the largest group size was 200.

The following section presents experimental results for the mean permutation costs for Move-ahead-k rules under request sequences generated by Z^{λ}. Section 5.4 considers other properties such as variance and distribution of permutation costs.

5.3. Experimental Results

This section presents experimental results for the mean permutation cost of Move-ahead-k rules. As the previous section notes, permutation cost at time T is an estimator of the expected search cost at time T. The following subsection presents results for request probabilities generated according to Zipf's Distribution. Subsection 5.3.2 considers search costs for varying λ. For notational convenience, the Move-ahead-k rule is denoted by Mk. At times the Move-to-Front rule is denoted by MF (rather than M(N-1)).

5.3.1. Zipf's Distribution

Exhibit 5-3 displays the mean permutation cost for 100 trials each at $N = 6, 8, 10$, $T \leq 20$, and $G = 1$. The curves correspond to the Move-ahead-k rules with odd index; for example $k = 1, 3, 5, 7$ for $N = 8$. In each panel the extreme rules M1 and MF are denoted by solid lines and intermediate rules are marked by dotted or broken lines. The curves are labeled according to their final ordering at the right side of each panel.

Recall the conjecture that for any two of these rules, the one with higher index will converge more quickly and the other will have lower asymptotic cost. Exhibit 5-3 supports this conjecture for Zipf's Distribution. At $N = 6$ (top panel), for example, the M1 has lowest cost after the 11^{th} request, but has highest cost at earlier T. The M3 rule has least cost when $5 \leq T < 11$ and is second-lowest at higher T. Finally, the Move-to-Front rule (M5) has least

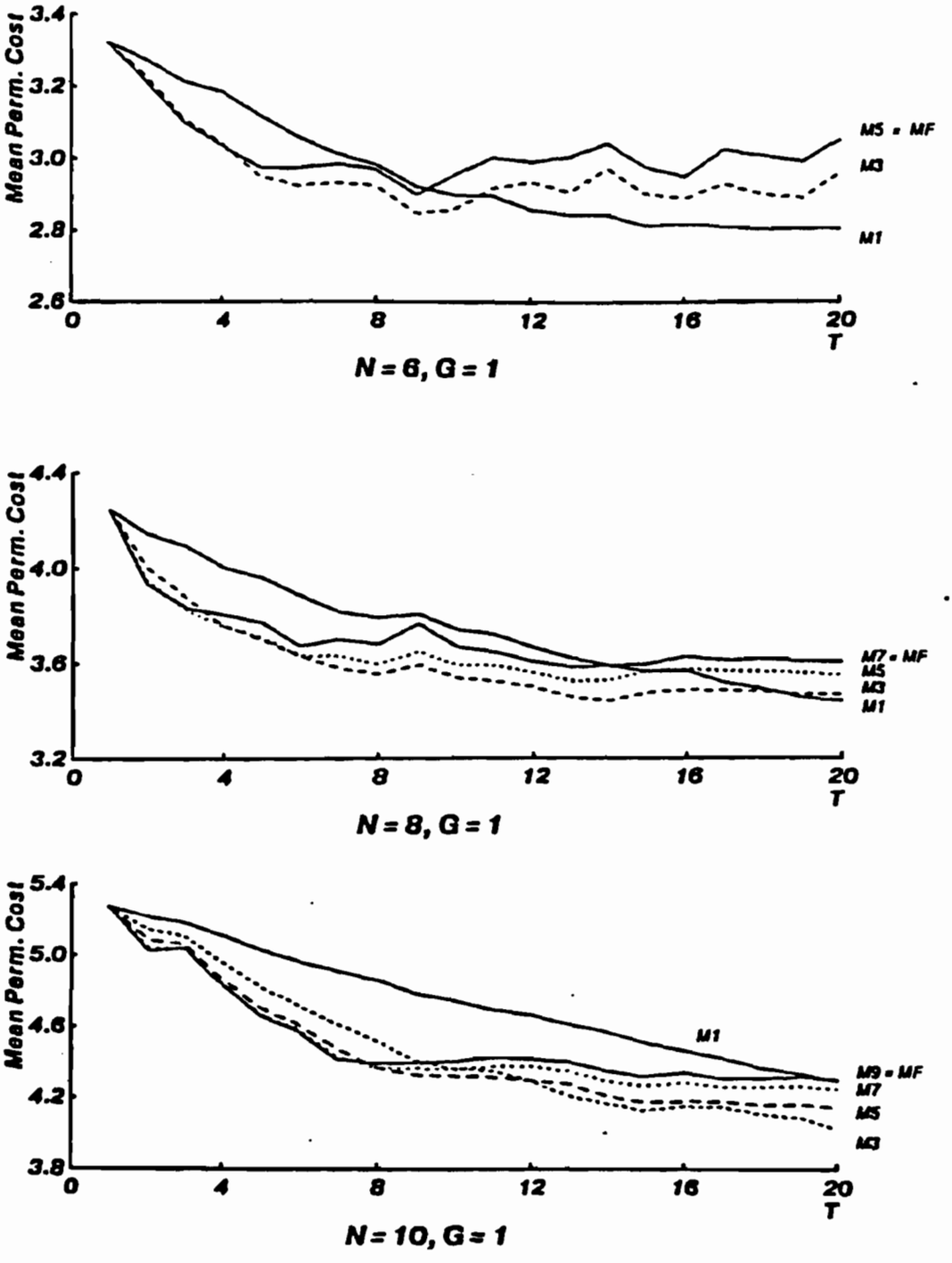


Exhibit 5-3: Mean Permutation Cost

cost when $T < 5$, but eventually has highest cost of the three. Similar behavior is displayed for $N = 8$: each rule in turn has least cost as T grows, with cutoff points at $T = 3, 6.19$ (marked by arrows). Eventually the rules arrange themselves into their conjectured asymptotic ordering. In general, rules with highest index dominate at the first few requests, then rules with lower index dominate in sequence until M1 dominates continuously at high T.

At $N = 10$, the asymptotic ordering of the search rules has evidently not been reached by the 20^{th} request: the M1 rule, although declining steadily, does not yet have lowest cost. The other rules have reached in their conjectured asymptotic order. It appears, then, that the number of requests required before M1 dominates increases as N grows.

. Bitner [5] has shown that for Zipf's Law, M1 will dominate MF after $\Omega(N^2)$ requests: how many requests are required before M1 dominates *any* rule? Assuming that the conjecture about relative convergence rates is true, this is equivalent to asking how many requests are required before M1 dominates M2. Although M2 was not measured in these experiments, Exhibit 5-3 gives a partial answer. For $N = 6, 8, 10$, M1 has cost lower than M3 for the first time at $T = 11, 19. (\approx 45)$, respectively. This observation and measurements at other N values suggest that the cutoff point grows approximately as the cube of N. The cutoff point for M2 and M1 must grow at least this quickly in N (again, assuming that the conjecture holds).

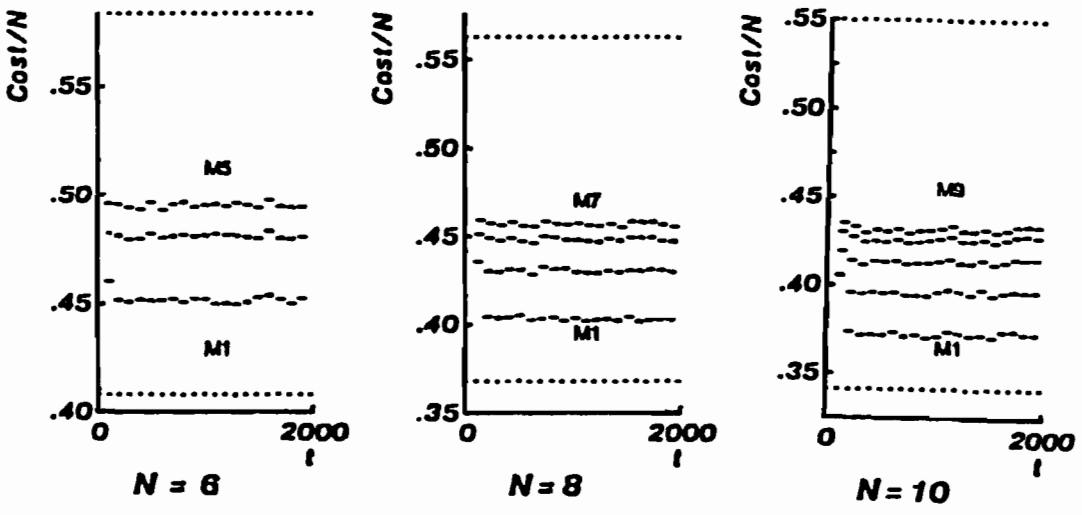


Exhibit 5-4: "Asymptotic" Behavior

Exhibit 5-4 presents "asymptotic" average search costs when requests are described by Zipf's Law. The data points in these graphs represent mean permutation cost for 100 trials each at the sample points $N = 6, 8, 10$, $T \leq 2000$, and $G = 100$: the rightmost data point in each curve, for example, corresponds to mean permutation cost for requests 1901 through 2000 (and 100 trials).

The conjectured asymptotic ranking of search rules is supported in these graphs, since rules with high index have higher cost than rules with low index when T is this large. Only the M1 and the MF rules are therefore marked; the intermediate rules appear in proper sequence between these two. In each graph the cost of the Optimal Static Ordering (a lower bound on Move-Ahead-k rules) is presented as a line at the bottom. The cost of the *random permutation* rule, which randomly reorders the search rule at each request and has cost $(N+1)/2$, appears as a line at the top.

The graphs are scaled for comparison by giving Cost / N, which corresponds to the fraction of the list searched at each request rather than the absolute number of comparisons. On this scale, the random permutation rule has expected cost $(N+1)/2N$ and the Optimal Static rule has cost $1/H_N$.

Not surprisingly, Move-Ahead-k rules have worst cost than Optimal Static but better cost than random orderings. As N grows, the range between the two bounds increases as $(N+1)/2N - 1/H_N$.[1] Since asymptotic expected search cost for Move-to-Front is bounded by approximately 1.386 times the cost of the Optimal Static Ordering (see Section 5.1), the gap between the Move-Ahead-k rules and the random ordering must increase while the gap between the rules and the Optimal Static Ordering remains bounded by a constant.

Exhibit 5-5 gives an idea of the relative asymptotic performance for the rules. In addition to experimental results, each table gives asymptotic bounds for the Optimal Static and the Move-to-Front rule, which can be computed from the formulas in Section 5.1. The column labels (Optimal) and (MF) correspond to these computed values. Table 5-5-a presents mean permutation cost for the last 100 of 200 requests (corresponding to the rightmost data point for each curve in Exhibit 5-4). Table 5-5-b gives these values divided by N (corresponding to fraction of list searched) so that comparisons across N may be made. Table 5-5-c presents ratios of permutation cost to Optimal Static at each sample point.

---

[1] H_N grows as $\ln N + \gamma + 1/(2N) - 1/(12N^2) + 1/(120N^4) + \varepsilon$, where $0 < \varepsilon < 1/(252N^6)$ and γ is Euler's constant $= 0.5772156649$.

N	(Optimal)	M1	M3	M5	M7	M9	(MF)
6	2.449	2.62	2.77	2.88			2.966
8	2.943	3.21	3.40	3.55	3.62		3.646
10	3.414	3.67	4.08	4.23	4.32	4.33	4.295

a. Expected Search Cost

N	(Optimal)	M1	M3	M5	M7	M9	(MF)
6	0.41	0.44	0.46	0.48			0.49
8	0.37	0.40	0.43	0.44	0.45		0.46
10	0.34	0.37	0.41	0.42	0.43	0.43	0.43

b. Expected Search Cost / N

N	(Optimal)	M1	M3	M5	M7	M9	(MF)
6	1.00	1.07	1.13	1.18			1.21
8	1.00	1.09	1.16	1.21	1.23		1.24
10	1.00	1.07	1.20	1.24	1.27	1.27	1.26

c. Expected Search Cost / Optimal

Exhibit 5-5: Asymptotic Search Costs

In Table 5-5-a, the column labeled (MF) gives the asymptotic expected cost for Move-to-Front, for which observed values are given by the rightmost Mk rule in each row. The differences between asymptotic expected cost for Move-to-Front and mean permutation cost for the corresponding Move-ahead-k rule are .063, .026, −.045 for $N = 6, 8, 10$, respectively. These differences are suggestive of the magnitude of error involved in trying to use measurements at finite T to assess asymptotic behavior (as $T \rightarrow \infty$).

At fixed N, it appears that the expected permutation cost increases sublinearly in k; the difference between costs for M9 and M7, for example, is much smaller than the difference between M3 and M1. It is possible that the differences between rules evens out as T grows; on the other hand, measurements at smaller T do not suggest significantly greater disparity. This conjecture of decreasing increments in search cost as k increases is supported by consideration of the behavior of the search rules. Suppose a search list of 10 items is initialized in random order. The Move-Ahead-9 rule is equivalent to Move-to-Front. In general, the Move-Ahead-7 rule performs a Move-to-Front operation unless the requested item is in the ninth or tenth position in the list; as T grows these exceptions become rare, so

the search costs for the rules are similar. In contrast, the Move-Ahead-1 and Move-Ahead-2 rules give *different* behavior unless the requested item is in the first or second position.

5.3.2. Varying Lambda

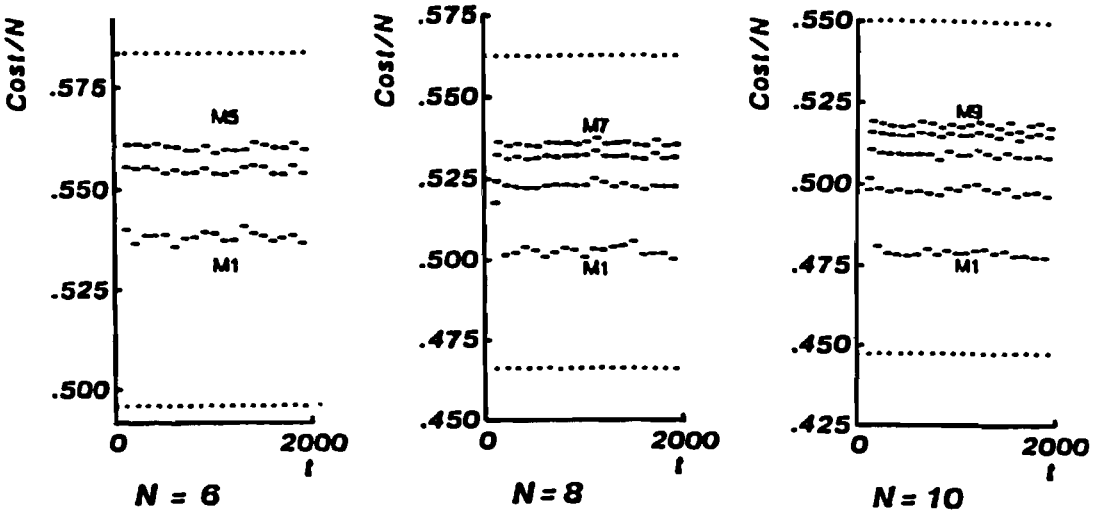
This subsection considers permutations costs for Move-Ahead-k rules as λ varies. Recall that $\lambda = 0$ corresponds to generating requests from the uniform distribution on the integers [1..N]. Zipf's Distribution is generated when $\lambda = 1$. A higher value of λ corresponds to a steeper density function for request probabilities. Limited preliminary experiments indicate that all the rules tend to converge quickly at higher λ.

Exhibit 5-6 presents the mean fraction of the list searched for the sample points $N = 6,8,10$, $T \leq 2000$, and $G = 100$, with request probabilities corresponding to $\lambda = 0.5$ and 1.5; compare these graphs to corresponding results for $\lambda = 1$ in Exhibit 5-4. Higher λ tends to gives lower average search cost for all the rules, in absolute terms (indicated by the change in scale) as well as in relation to the random permutation rule. In addition, the spread among Move-Ahead-k rules tends to decrease as λ increases.

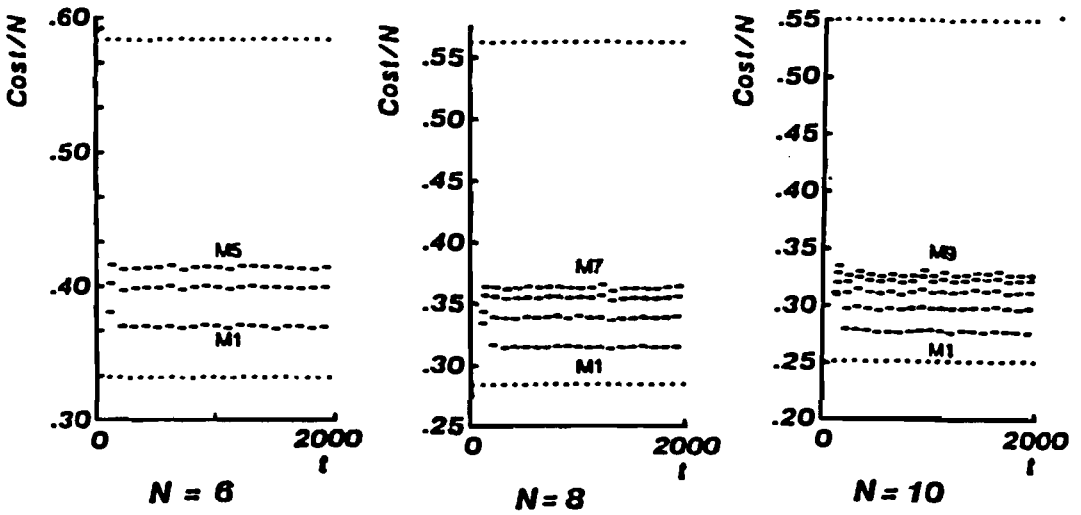
Exhibit 5-7 compares "asymptotic" search costs among the rules; each point represents the mean of the last 100 of 2000 requests for 100 trials at each sample point. Graph 5-7-a presents permutation costs for λ set at 0, 0.5, 1, 1.5. Within each group, costs for each N are ordered by increasing k. When $\lambda = 0$ the requests are uniformly distributed, so permutation costs are identical for the rules, equivalent to $(N+1)/2$. Graph 5-7-b compares the Transpose (M1) rule and the Move-to-Front rule (M5, M7, M9, respectively) for the three N settings; within each group the points are plotted as a function of increasing λ.

5.4. Properties of Search List Permutations

The theoretical measure "expected search cost" is defined in terms of a probability distribution on the search list permutations; the probability of a given permutation of the search list appearing at time T (or as T goes to infinity) is combined with the cost of that permutation to determine the expected search cost. To give a more detailed view of the Move-Ahead-k rules, this section examines properties of permutation costs and permutation frequencies. Intuitively, a good rule ensures that permutations with low cost appear with high frequency.



a: Lambda = 0.5



b: Lambda = 1.5

Exhibit 5-6: Mean Permutation Cost

An appropriate experimental approach might be to sample search list permutations and estimate their distribution as a function of N and T. Unfortunately, the space of search list permutations is of size $N!$, which presents a number of obvious difficulties. For one thing, the

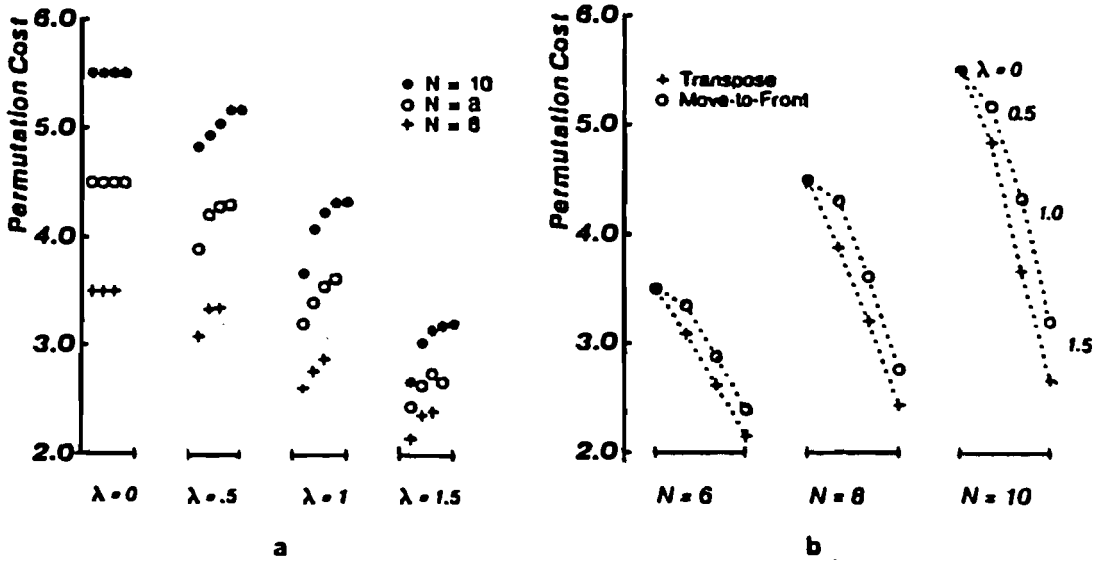


Exhibit 5-7: Varying Lambda and *N*

simulation program would require an exponential amount of space to store the frequency counts for each permutation. Also, many trials must be run to obtain a useful sample when the parent population is so large. The permutation cost was therefore adopted for the simulation.

On the other hand, the permutation cost serves as a "signature" for the permutation at time *T*; it should be possible transform frequency distributions on permutation costs into frequency distributions for permutations. How are permutation costs distributed among the permutations of search lists?

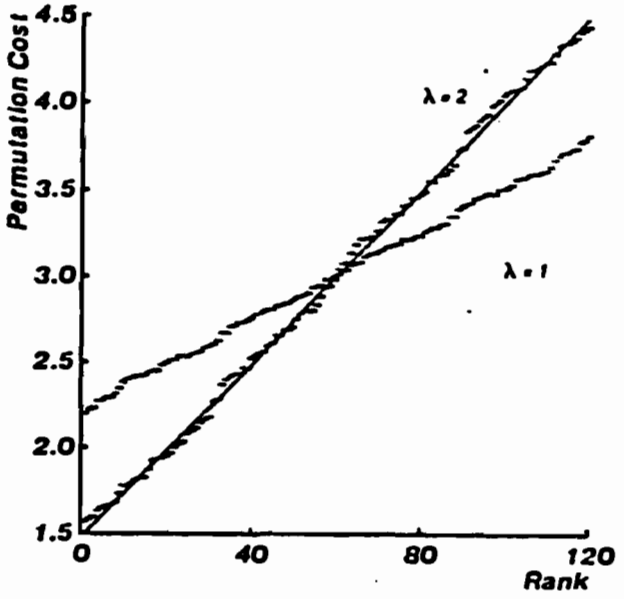
Clearly permutation costs are symmetric: the permutation with least cost is in reverse order of the permutation with highest cost, the permutation with second-lowest cost is in reverse-order from that with second-highest cost, and so forth. It might be the case, however, that most permutations have moderate cost and a few have extremely high- and low costs. For a particular permutation π, where $\pi(i)$ represents the position of item i in the permutation and p_i its probability of being requested, permutation cost is given by

$$C(\pi) = \sum_{i=1}^{N} \pi(i) \cdot p_i$$

Table 5-8-a presents permutation costs assuming $N = 4$ and Zipf's Distribution for the request probabilities. The first column gives the index of the permutation in a lexicographical

Lex	Perm.	Cost
0	1234	1.92
1	1243	1.96
2	1324	2.00
4	1423	2.08
3	1342	2.12
6	2134	2.16
7	2143	2.20
12	3124	2.32
13	3142	2.44
18	4123	2.44
8	2314	2.48
19	4132	2.52
10	2413	2.56
14	3214	2.56
20	4213	2.68
16	3412	2.80
22	4312	2.84
9	2341	2.84
11	2431	2.88
15	3241	2.92
21	4231	3.00
17	3421	3.04
23	4321	3.08

a



b

Exhibit 5-8: Permutation Costs

ordering: 1234, 1243. 1324, . . . 4312, 4321. The second column gives the permutation and the third column the permutation cost.

Graph 5-8-b presents permutation costs for the case $N=5$ and for the distributions corresponding to $\lambda=1$ (Zipf's Law) and $\lambda=2$ (Lotka's Law) with a linear regression line superimposed on the latter. The permutation costs are plotted against their rank. Permutation cost for Zipf's Law has range [2.19. 3.18] and permutation cost for Lotka's Law has range [1.56, 4.44]. The probability distribution giving the most extreme range in permutation cost has $p_1=1, p_{i>1}=0$: permutation cost in this case is equivalent to the position of item 1 and ranges from 1 to N.

Similar graphs indicate that ranked permutation costs for the family of Z^{λ} distributions are evenly distributed over their range and are well represented by straight lines. For the case $\lambda=0$ (corresponding to a uniform distribution on requests), permutation costs would give a horizontal line at $1/N \cdot \sum_{i=1}^{N} i = (N+1)/2$; for $N=5$ this value is 3. The slopes of linear

regression fits to the ranked costs for $\lambda = 0, 1,$ and 2 are 0, 0.013, and 0.0252 respectively: these and similar results suggest that ranked permutation costs have slopes that increase proportionally to λ.

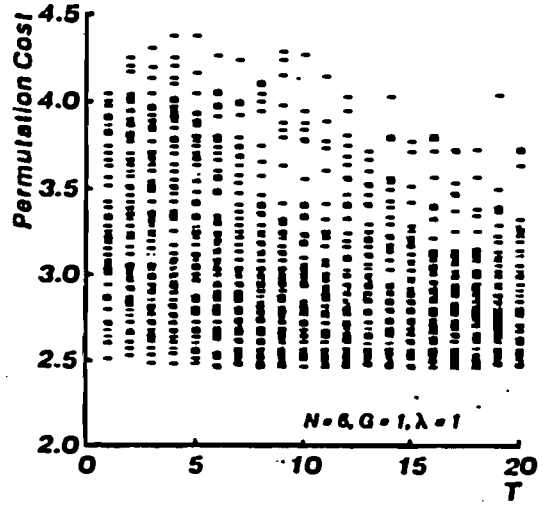


Exhibit 5-9: Permutation Distributions

Since ranked permutation costs form an almost straight line, permutation cost is linearly related to permutation index (when the permutations are ranked by increasing cost). A graph presenting the observed distribution of permutation costs would therefore have shape nearly identical to that of graph representing the distribution of permutations by rank.

Exhibit 5-9 presents the distribution of permutation costs for the Transpose rule for 100 trials at the sample point $N=6$, $T \leq 20$, $G=1$, $\lambda=1$. The smooth bottom line in the points suggest that the optimal ordering, with cost $6/H_6 \approx 2.449$, is regularly achieved. On the other hand, the pessimal ordering, with cost 4.55, never appears. In general, permutations appearing most often are concentrated at the low-cost end of their range, and the concentration gets tighter as T grows.

Although this graph gives a good idea of the location of permutation costs, the distribution of costs are not clearly seen. Exhibit 5-10 presents stem-and-leaf charts showing the distribution of permutation costs for 100 trials at the request for $N=6$ and $k=1, 3, 5$, $\lambda=1$. The top three charts give permutation costs at the fifth request, and the bottom three give permutation costs at the twentieth request. The two leftmost charts, for the M1 rule, present

M1	M3	M5
43 6	44 4	44 8
42	43	43 66
41 55	42 44	42
40 3	41	41
39 112	40	40 5
38 5578	39 1	39
37 3689	38 59	38
36 225799	37 38	37 3
35 3	36	36 02359
34 0167	35 0	35
33 11239	34 68	34 1
32 02223346	33 69	33 089
31 001116	32 334	32 112334778
30 2334779	31 11112334668	31 01111124667
29 55689999	30 1333556789	30 3335599
28 034446	29 00337999999	29 39999
27 000011112256789	28 01344	28 000002344456669
26 23556777778	27 001112222667777788899	27 0011112227778
25 0334668	26 11112777889	26 01133777789
24 7	25 034466677	25 23346677999
	24 556678	24 5778

N = 6, T = 5

M1	M3	M5
	42 8	
	41 1	41
	40 3	40 15
	39 6	39 9
	38 55	38 44999
37 012	37 14	37 11349
38 2	36 5	36 57
35	35 59	35
34	34 01466	34 0166
33 112	33 11556888	33 14455568
32 11244457	32 00457	32 144558
31 038	31 11	31 116
30 1114	30 111344	30 1114444446
29 033377789	29 000005	29 0377
28 00033347	28 00011344466778889	28 0000111344467788899
27 0000555568677888899	27 002555799	27 015579
26 0122367779	26 03333356799	26 00011233588
25 689999999	25 002246666666679999	25 006799999
	24 68	24 6788

N = 6, T = 20

Exhibit 5-10: Distribution of Permutation Costs

exactly those data points appearing in Graph 5-9-a at $T=5$ and $T=20$. The other two charts give corresponding measurements for the M3 and M5 rules.

In each chart the stem corresponds to the first two digits of permutation cost, and the entries in the leaves to the last digit. For example, the bottom line of the bottom right chart corresponds to permutations with costs 2.46, 2.47, 2.48, 2.48. In the top row of charts the optimal permutation, corresponding to the data point 2.45, appears once in 100 trials under the M5 rule, twice under the M3 rule, and zero times under the M1 rule. (See Section 7.8 for a discussion of how to read stem-and-leaf charts.)

The lengths of the rows suggest the relative frequency of permutation costs appearing over 100 trials. In the top charts, the search rules have only processed five requests and therefore have little information about request frequencies. This observation is reflected in the large spread of points in the top charts. M1 has a somewhat smoother distribution of costs than the other two rules: both tend to M3 and M5 have stragglers and gaps at their high ends.

The bottom charts depict permutation costs after the 25th request. The M1 rule has greatly reduced its range of permutation costs, concentrating them towards the low end. It is interesting to note that although Transpose gives generally lower permutation costs, the optimal permutation and other low-cost permutations (with costs 2.46, 2.47, 2.47) are never seen in 100 trials. The M5 and M3 rules, although they have reduced the range of permutations from those seen at the fifth request, still tend to straggle towards the high end, giving higher mean cost overall.

The graphs and tables in this section give preliminary insight into the relationship between permutation costs and permutation frequencies. Theoretical characterization of permutation distributions for general Move-Ahead-k rules remains an open problem.

References

[1] M. E. Bellow.
Performance of Self-Organizing Sequential Search Heuristics under Stochastic Reference Models.
PhD thesis, Department of Statistics, Carnegie-Mellon University, Pittsburgh, PA, November, 1983.

[2] J. L. Bentley and C. C. McGeoch.
Amortized analysis of self-organizing sequential search heuristics.
Communications of theACM 28(4):404-411, April, 1985.

[3] J. R. Bitner.
Heuristics that Dynamically Alter Data Structures to Reduce Their Access Time.
PhD thesis, University of Illinois, July, 1976.

[4] J. R. Bitner.
Heuristics that dynamically organize data structures.
SIAM Journal of Computing 8(1):82-110, February, 1979.

[5] J. R. Bitner.
Heuristics that dynamically organize data structures.
SIAM Journal of Computing 8(1):82-110, February, 1979.

[6] P. J. Burville and J. F. C. Kingman.
On a model for storage and search.
Journal of Applied Probability 10:697-701, 1973.

[7] F. R. K. Chung, D. J. Hajela, and P. D. Seymour.
Self-organizing sequential search and Hilbert's inequalities.
In *Proceedings 17th STOC*, pages 217-223. ACM, May, 1985.

[8] G. H. Gonnet, J. I. Munro, and H. Suwanda.
Exegesis of self-organizing linear search.
SIAM Journal of Computing 10:613-637, 1982.

[9] W. J. Hendricks.
The stationary distribution of an interesting Markov Chain.
Journal of Applied Probability 9:231-233, 1972.

[10] D. E. Knuth.
The Art of Computer Programming: Volume 2, Seminumerical Algorithms.
Addison-Wesley Publishing Company, Reading, MA, 1973.

[11] J. McCabe.
On serial files with relocatable records.
Operations Research 12:609-618, 1965.

[12] R. Rivest.
On self-organizing sequential search heuristics.
Communications of the ACM 19(2):63-67, February, 1976.

[13] D. D. Sleator and R. E. Tarjan.
Amortized efficiency of list update and paging rules.
Communications of the ACM 28(2), February, 1985.

[14] A. Tenenbaum.
Simulations of dynamic sequential search algorithms.
CACM 21(9):790-791, September, 1978.

Part III

Experiments and Algorithm Analysis

A quite ordinary fact, principle or technique from one branch of science may be novel and fruitful when applied in the other branch.

– W. I. B. Beveridge
The Art of Scientific Investigation

Principles and techniques for a well-developed simulation study are addressed in this section. The contributions of the thesis are surveyed and suggestions for future work are presented.

Chapter 6
Experiments and Algorithms

This chapter discusses the applicability of experimental research to algorithm analysis and presents principles for performing experimental studies.

6.1. Why Do Experiments?

One of the goals of this research is to demonstrate that experimental analysis can make significant contributions to the understanding of combinatorial algorithms. The previous four chapters describe experimental studies of a variety of algorithm problems. Do the studies contribute new understanding in the algorithm domains? Were they necessary to obtaining the understanding? I claim the answer is yes. The case studies deal with well-known domains. Quicksort is one of the most extensively analyzed algorithms of all time. Heuristics for bin packing have generated considerable previous research, both experimental and analytical. Sequential search rules been studied for over twenty years. Matching is a well-known problem on graphs. Despite the extensive previous attention these problems have received, new facts were discovered. To my knowledge, the following observations from the case studies represent new results in the problem domains; all were direct products of experimental research.

- Bin Packing
    - The observation that empty space in First Fit and Best Fit is asymptotically optimal when $u = 1$. The derivation of subsequent theorems.
    - The observation of nonmonotonicity in FF and BF, and the conjecture that empty space is linear in n for some values of u.
    - Measurement of the location of minima and maxima in the nonmonotonic curves. Observation that the local minimum shifts in n.
    - Measurements of k-item bins and gaps and detailed arguments for linear growth of empty space at some values of u.
    - Proof that the expected number of 1-item and 2-item bins must be at least linear in n when u is greater than 2/3 and less than 1, for any packing rule.
    - Observation that Best Fit gives better packings than First Fit at all sample points.

- Discovery that empty space in First Fit Decreasing packings is $O(1)$ when $u \leq 0.5$. A subsequent proof of this fact and better understanding of the structure of FFD packings.
- Observation that partial empty space grows nearly as the cube of u when $u \leq 0.5$. Discovery of a cyclic component in partial empty space.
- The discovery of a "critical region" in First Fit Decreasing where very bad packings appear, and a partial characterization of lists that cause bad packings.
- Characterization of near-linear growth in u for empty space FFD packings when u is between 0.5 and the critical region. Observation of cyclic behavior of empty space in this region.
- Comparisons of empty space and partial empty space in BFD and FFD packings of uniform weight lists. Observation that the rules give identical empty space very often.

• Greedy Matching

- Observation of logarithmic edge cost of Greedy matchings in one dimension.
- Observation of linear computation cost of the shortcut algorithm for Greedy matching.
- Observation that the number of levels reached by the shortcut algorithm grows logarithmically in N.
- Observation that 1/3 of the points are removed at the first level, prompting the subsequent (trivial) proof of this fact.
- Observation that slightly fewer than 1/3 of the remaining points are removed at higher levels, and that the fraction removed is constant in N although variance increases with the level number.
- Observation that the mean edge cost per level is constant in N.
- An argument for logarithmic growth in expected edge cost.
- An argument to support the logarithmic number of levels reached by the shortcut algorithm.
- An argument for linear computation cost of the shortcut algorithm.

• Quicksort

- Measurements of fixed-T strategies for an extension of the previous analytical model that explicitly counts the cost of median selection.
- Discussion of tradeoffs between partition comparisons and selection comparisons.
- Presentation of the M and N ranges where each fixed-T rule dominates (in terms of total comparisons).
- Observation that in some metrics it is not necessarily a good idea to obtain the exact median, and an argument to generalize this observation.
- Measurements of a version of Quicksort that allows the sample size to vary as a function of sublist size.
- Derivation of optimal choices of T that minimize the total number of comparisons. Observation that optimal T grows approximately as the square root of n, the sublist size.
- Approximation of the optimal strategy by a square-root strategy, and discussion of the "best" square-root rule in the range of experiments.

- Discovery of an error in earlier analysis and the derivation of the correct formula. A new computation of the optimum cutoff value in Sedgewick's MIX implementation of Median-of-3 Quicksort.

- Sequential Search

  - Discussion of a new measure to estimate average search cost for sequential search rules.
  - Measurements of search costs for a spectrum of Move-ahead-k rules when request sequences are distributed according to Zipf's Law.
  - Characterization of the area of dominance for each rule as the length of the request sequence increases.
  - Approximate measurements of asymptotic performance for each rule.
  - A comparison of asymptotic costs as the "sharpness" of the request distribution varies.
  - Characterization of ranked permutation costs for the Z^{λ} family of request distributions.
  - Comparison of permutation frequencies and permutation costs for a set of rules.

Clearly, experimental results need not be limited to benchmark-style comparisons or tables of measurements at a few sample points. Experimental analysis can lead to new observations, new conjectures, arguments to explain observed behavior, new theorems, and new insights into underlying mechanisms.

6.2. Applications and Limitations of Experimental Analysis

This section considers the type of algorithmic problems that might be appropriate to the application of experimental techniques.

The algorithmic problems considered in the case studies all involve expected-case analysis. Experimental work is naturally applicable here because it is generally a straightforward task to generate input instances according to a well-defined probability distribution. Although this research was restricted to consideration of expected-case behavior, application to other analysis domains are possible. For example, experimental results are useful when the input is gathered from an existing system. It can be very difficult to obtain an adequate mathematical description of realistic input to the system or to develop an efficient generation scheme; gathering examples of "typical" input may be the only approach available. A related approach is to compare a promising algorithm against existing algorithms by testing on a standard set of input instances. New heuristics for the Traveling Salesman Problem, for example, are often evaluated on a set of problem instances which includes U. S. state capitals

and major German cities (see [4] for more standard problems). Even when a model of input is available and algorithmic behavior can be analyzed, experiments can give precise measurements of resources used.

These uses of experimental research have many properties of standard benchmark-style simulation, which differs somewhat from the approach taken here. Within the context of expected-case analysis of algorithms, the uses of simulation are many:

Experiments can be used to compare algorithms. Simulation results can identify the "best" algorithm within a class for the given sample points. This information can be used to characterize dominance among the algorithms, or to identify input properties that determine best performance. Many examples of this use of simulation can be found in previous work as well as in the case studies.

Experimental results can direct theorem-proving efforts. Experiments can be used to support or refute conjectures developed by partial theoretical characterization. Experiments are especially valuable when they contradict prior intuition. In the Bin Packing study it was widely conjectured (see [3] or [6], for example) that since online algorithms for bin packing (including First Fit and Best Fit) have no opportunity to rearrange their input, they cannot be asymptotically optimal; even in an expected-case model they would have an asymptotic bin ratio strictly greater than 1. Simulation results suggested that this intuition was wrong since empty space was observed to be sublinear in n, implying that the bin ratio must approach 1. Also surprising was the observation of nonmonotonicity in empty space as u varies; this phenomenon has not yet been characterized theoretically.

The Greedy Matching study gives another example: in all higher dimensions, the Greedy heuristic produces matchings that are within a constant factor of optimal. It is natural to conjecture that this will be the case in one dimension, and Steele had tried to prove the constant-factor bound. Experiments demonstrated, however, that the bound does not hold, since the cost of the Greedy matching grows logarithmically in N and the Optimal matching is known to have constant cost.

Early in the Quicksort study, simulation results were compared to Sedgewick's formulas for Median-of-3 Quicksort. Observation that the measurements matched every formula but one led to the discovery of an error in the theoretical formula and to a recomputation of the optimum value for M in Sedgewick's fast implementation of Quicksort.

Experiments allow greater precision of analysis. Experimental results are generally expressed with more precision than current theoretical approaches can attain - for example, an experimental result is more naturally given as $3.45N$ rather than $O(N)$. Experimental results can therefore suggest directions for tightening current theoretical bounds. Simulations of First Fit Decreasing led to the conjecture that empty space is constant in N. The theoretical bound in [1] gives a constant of at least 10^{10} and the proof only holds for very large lists. Floyd and Karp [2] have recently reduced the asymptotic bound to 10 under a slightly different averag-case model. Experiments results suggest, however, that empty space is rarely outside the range 0.7 ± 0.5. The measure *partial empty space* reveals even more precision; this measure converges in n and is never observed to vary by more than ± 0.005 at high n. Partial empty space is more precise than empty space by a factor of 100; it is more precise by than the current theoretical bound by a factor of 10000.

Simulation can give results more efficiently than analysis. Experiments can be of use even when theoretical analysis already exists, especially if simulation is more computationally efficient than theoretical analysis. For example, in the Search study a formula exists for the asymptotic search cost of the Transpose rule. Computing this formula requires $N!$ time, however, and has only been done for small N and requests described by Zipf's Law. Chapter 5 gives simulation measurements for this rule (and others) for a range of distributions that includes Zipf's Law.

Experiments can generate new insight, new arguments, and even new theorems. Experimental results need not be limited to "mere measurement." In the case studies functional relationships were characterized and detailed arguments were developed to explain observations. In the Bin Packing and Matching studies some arguments were formalized to become theorems. Detailed views of algorithmic behavior and precise measurements can give deep insight into underlying structures. The potential for producing new insight gives strong motivation for using experimental tools in this domain.

As a simulation problem, the study of algorithms presents special difficulties as well as opportunities. Textbook examples of simulation problems generally come from studies of domains such as economic systems or performance of computer operating systems. Problems in algorithm analysis differ in a number of ways from more familiar domains:

- Algorithms are simpler to simulate. Unlike economic systems, they have simple, rigorous, mathematical descriptions. In expected-case studies, the input usually has a simple mathematical description as well.

- Some issues of traditional simulation research, such as developing and validating realistic models, become less important. In current practice algorithms are analyzed in terms of simple abstract machines and well-defined probability distributions. While an eventual goal of algorithm analysis is presumably to obtain theoretical results that accurately reflect computation on real machines, this goal is approached incrementally in order to establish a firm mathematical base.

- Algorithms have relatively few parameters. Except for the Search study, which involved parameters N, T, k, G, and λ, the case studies and previous work were generally restricted to consideration of one or two parameters. As a result the complexity of displaying and analyzing interactions between parameters is generally less than for classic simulation problems.

- Experiments are often less expensive. Algorithms are interesting because they are efficient. In the case studies simulation time per trial was generally reckoned in seconds, while simulations of complex systems can require hours, even days of computation time. There are of course exceptions to the above generalities. Simulations by Johnson and McGeoch [5] of a simulated annealing algorithm for the Traveling Salesman Problem took up to six hours per trial.

It might seem that the study of algorithms presents a much simpler simulation problem than standard domains. On the other hand, simulation results must be compared to theoretical characterizations of algorithms. Theorems have been preferred over experimental results because they represent certainty about bounds on algorithmic behavior and can sometimes be generalized to broad classes of algorithms and input distributions. Theorems also express understanding of the mechanisms underlying the algorithm. In contrast, experimental results consist of measurements at specific sample points with specific implementations. As with any experimental domain, generalization of experimental results without real understanding of the underlying process must contain some degree of uncertainty.

Two fundamental problems in applying simulation to algorithms are how to reduce uncertainty in simulation results and how to use the results to gain new insight into underlying mechanisms. While these problems cannot be entirely eliminated, much can be done to lessen their severity. The following section discusses principles for experimental research in this domain. Chapter 7 presents a number of tools that can be applied in order to realize these principles.

6.3. Principles

Four general principles for experimental research in the domain of algorithm analysis are presented in this section. The principles were developed from experience with the four case studies, three small experimental studies not presented in this thesis, and the survey of previous work presented in Section 1.2.

- *Match the simulation results to a well-defined analytical model.* Simulation research in algorithm analysis is usually prompted by unanswered questions from theoretical approaches. Experimental approaches should be viewed as a companion to theoretical approaches when studying a particular algorithm. It is therefore important to reduce as much as possible the distinction between simulation model and simulation program, and to obtain simulation results that can be expressed in analytical terms. Sections 7.1, 7.2, and 7.5 discuss techniques for establishing the accuracy of simulation results.

- *Search for a good "view" of the data.* A good view of experimental results is obtained when the variation at fixed sample points is small relative to growth as parameter settings vary. When a good view is obtained it is generally easier to obtain accurate measurements, to assess functional relationships, and to discover underlying structures. The view of the data can be improved by the use of appropriate data-analysis tools. In addition, a number of techniques may be applied to improve the results of the simulation before data analysis occurs. Sections 7.1, 7.3, and 7.4 discuss techniques for improving simulation results. Section 7.8 discusses analysis techniques that proved useful in the case studies.

- *Analyze the data, don't just measure it.* Analysis of experimental results should not stop at a tabular presentation of means for each sample point. Measurements can be transformed and combined and functions can be fitted. The object of data analysis in this context is to manipulate measurements to gain new insight into relationships between parameters and measures. Section 7.8 presents a number of data analysis tools.

- *Iterate theoretical and experimental approaches.* A fundamental concept in traditional experimental domains is that theory and experiment must be iterated. An important component of the case studies was the rich interaction between experimental and analytical approaches to analysis. I tried to preserve this evolutionary development in the presentation of the case studies. Not only did experimental results direct theorem-proving efforts, but theoretical insight often suggested more useful measures, better choices of sample points, and more efficient experimentation. Sections 7.1, 7.4, 7.5, 7.6, and 7.7 explore opportunities for improving the simulation study and for developing simulation programs that support an interactive, iterative approach to analysis.

These four principles can be approached at many levels. Techniques of algorithm analysis, for example, can suggest better measures, faster simulation programs, and ways to check the

accuracy of simulation results. Statistical methods for sampling and experimental design can be applied to gain more efficiency of experimentation and to eliminate redundant experiments. Program development tools are needed in building efficient simulation programs and supportive environments. Many practical hints from the domain of simulation can be applied. A variety of data analysis tools are useful.

The following chapter presents practical hints, statistical techniques, and data analysis tools for achieving the four goals listed above. The discussion of Chapter 7 results from experience with the cases studies; I believe they can be of use in many simulation studies of algorithms.

References

[1] J. L. Bentley, D. S. Johnson, F. T. Leighton, C. C. McGeoch, L. A. McGeoch.
Some unexpected expected-behavior results for bin packing.
In *Proceedings, 16th Symposium on Theory of Computation*. ACM, April, 1984.

[2] S. Floyd and R. Karp.
FFD bin-packing for distributions on [0, 1/2].
In *Proceedings, 27th Symposium on Foundations of Computer Science*. IEEE, October, 1986.

[3] D. S. Johnson.
Near-Optimal Bin Packing Algorithms.
PhD thesis, Department of Mathematics, Massachussetts Institute of Technology, Cambridge MA, June, 1973.

[4] S. Lin.
Computer solutions of the traveling salesman problem.
The Bell System Technical Journal 2245-2269, December, 1965.

[5] L. A. McGeoch.
Personal communication.
1986.

[6] H. L. Ong, M. J. Magazine, T. S. Wee.
Probabilistic analysis of bin packing heuristics.
Operations Research 32(5):983-998, September-October, 1984.

Chapter 7
Tools and Techniques

This chapter presents tools and techniques for enhancing experimental studies of algorithms. Algorithmic insight and program development techniques are applied to improve efficiency and accuracy of simulation programs. Statistical techniques such as sampling plans are considered. A number of guidelines and techniques from the field of simulation are presented. Finally, useful data analytic tools are surveyed.

Many of the topics addressed here are found in advanced texts on simulation, experimental statistics, or data analysis. The contribution of this chapter is to gather knowledge from diverse fields, to describe those techniques and guidelines that were particularly useful in the case studies, and to discuss their application in the domain of algorithm analysis.

Familiarity with elementary statistical analysis is assumed. For a survey of statistical concepts, see DeGroot [12] or Mosteller, Fienberg and Rourke [21]. Feller's [14] two-volume work is a standard source in probability theory. Box, Hunter, and Hunter [5], and Miller and Freund [20] discuss statistical issues that particularly apply in experimental research. For texts on simulation, see Adam and Dogramaci [1], Brately, Fox, and Schrage [9], Fishman [15], or Hammersley and Handscomb [17]. Many of the data analytic tools presented here are described in Tukey [24], Chambers et al [10], Cleveland [11], and Mosteller, Fienberg, and Rourke [21].

The principles described in the previous chapter can be approached at many stages during the simulation study. For example, the correspondence between measurements and models can be influenced by choice of measure, correctness of the implementation, choice of random number generator, and the placement of sample points. The following sections are organized by procedural issues that arise in the course of a simulation study.

7.1. Choice of Measure

Usually the measure adopted for the simulation is suggested by previous theoretical analysis. Simulation results should match theoretical measures as much as possible. Timing statistics of a program are therefore rarely useful when investigating the time complexity of an algorithm: noise due to implementation factors, machine loads, or compiler optimization can seriously degrade this measure. Accurate results can be obtained by embedding simple bookkeeping mechanisms into the simulation program to count the number of key operations.

Measures should be suggested by theoretical results, but not necessarily constrained by them. It may be the case that the analytical measure of interest is difficult to measure experimentally. This occurs, for example, when the algorithm is a heuristic for an NP-hard problem and performance is expressed relative to the optimal solution (which cannot generally be found).

If the measure suggested by the simulation model is not amenable to experimentation, it might be possible to identify an alternative measure that is theoretically interesting as well as experimentally practical. The Bin Packing study used the well-defined and easily-computed measure *empty space*, for example, rather than the *bin ratio*. In Search, the analytical model involved the steady state probabilities on the $N!$ search list permutations. Since this measure is impractical for simulation because of the size of the sample space, the alternative measure *permutation cost* was used in the case study.

Choice of measure is also constrained by available analysis tools. A too-detailed measure can produce huge amounts of data, possibly overwhelming statistical analysis tools, graphical display packages, or even machine storage capability. At one point in the Bin packing experiments I tried generalizing the measure *number of big items* (which counted the number of weight list items with size greater than 0.5) to the number of items with weights in each of the subranges $(0, 0.1]$, $(0.1, 0.2]$, ..., $(0.9, 1]$. This measure was discarded due to the ten-fold increase in data and the difficulty of characterizing empty space in terms of ten variables. Early experiments in the Search study reported permutation cost at every T for T as large as 5000. It quickly became clear that a summarization scheme was needed, so grouped averages were adopted.

Whatever the initial measure, be prepared to change it. At the beginning of a simulation study, consideration of theoretical results can suggest good measurements. An important

component of the iterative approach to experimental analysis, however, is to modify the measure as new insight is gained. In the case studies the measure generally became more detailed as experiments evolved. In Bin Packing the measure progressed from *empty space* to more detailed measurements such as *partial empty space*, *counts of k-item bins*, and *gaps in k-item bins*. In Matching the measures changed from total costs to consideration of costs at each level. Even if there is no hope of theoretically characterizing the detailed measurements, they are valuable for giving insight into underlying structures. Increase in detail was often accompanied by a reduction in the number of trials and sample points, due to limitations of technology and patience.

7.2. Ensuring Correct Results

Experimental results are only as strong as the fidelity of the simulation program to the simulation model. The relationship between measurements and the algorithmic model must be close and well-understood. A number of techniques for checking the accuracy of experimental results are available.

It is often possible to compare measurements against known theoretical results. This situation arises when the simulation involves an extension of a standard model, as was the case in the Quicksort study. In this study, checking simulation results against formulas led to the discovery of an error in previous theoretical work.

Comparison against known theoretical results can also be useful for clarifying details of the simulation model. For example, the simulation program for Quicksort requires a routine to generate the median of T integers selected randomly from $[1, N]$. For the analogous problem on the real interval $(0, 1]$, the median of T randomly-selected reals has a Beta distribution with parameters $(T+1)/2$ and T. Beta variates can be generated in constant time per variate, so I implemented a generator that produces Beta variates and scales them to the integers $[1, N]$. The simulation model is not *exactly* met by this implementation because Beta generation corresponds to sampling without replacement and the model to sampling with replacement. I reasoned that this difference would not significantly perturb the results since T is generally much smaller than N. I was incorrect: measurements under the Beta scheme differed significantly from the model, so this method was discarded.

Once a clear specification of the model is obtained, the next task is to make sure that the simulation program performs as specified. Standard program verification and validation

techniques are appropriate here and should be applied. Limitations of machine precision can be an important factor since theoretical models generally assume properties of reals.

In simulation problems the choice of pseudo-random number generator can be critical. There is a vast literature describing empirical and statistical tests of generators, as well as evaluations of well-known generators. Random number generators used in the case studies are described in Section 7.7. No matter how many statistical tests a particular generator passes, it may have subtle non-random properties exposed by the simulation problem. Early in the Search study, for example, the *cost per request* exhibited unusual periodicity in the number of requests. This periodicity was not dependent upon the random number generator in any obvious way because the cost function depended upon the search list ordering as well as on the requested item; that is, cyclic patterns in the requests should not necessarily give cyclic patterns in search costs for *all* search rules since *each* rule reorders its list differently. In any case, the cyclic behavior clearly indicated a violation of the assumption of independence in requests: replacing the linear-congruential generator by an alternative generator caused the periodic behavior to disappear.

The most import assurance of experimental integrity is *replication*, a standard component of research in the experimental sciences. At the very least, critical experiments should be replicated using an alternative random number generator. An even better practice is to develop a secondary simulation environment, varying such factors as implementation strategy, random number generator, and machine word size. If results are consistent across the two environments, then there is strong assurance that the results are independent of environmental factors. Section 7.8 presents tools for comparing results from separate implementations.

7.3. Variance Reduction Techniques

One of the principles for simulation research is to obtain a good view of the data – that is, to reduce variation in measurements at specific sample points with respect to growth as parameters vary. One way to obtain a better view is to reduce variance in the measurements at each sample point.

An obvious way to reduce variance is to take more trials per sample point; Section 7.6 discusses methods for improving the efficiency of simulation programs. In addition, many *variance reduction techniques* can be incorporated into the simulation programs. Variance

reduction techniques that were useful in the case studies are described in this section. For further discussion and descriptions of other techniques, see texts on simulation such as [9], [15], or [17].

In the following, assume that algorithms A and B are being studied at a fixed sample point: measures are denoted by X, Y, and Z. Since input instances are randomly generated, the measurements at a given sample point are random variables from some (unknown) distribution. Let X_i denote the value taken for measure X at the i^{th} trial.

Conditional Monte Carlo

In studying algorithm A, it may be the case that measure X is a function of other random variables in the simulation program. Variance in X can be reduced if the intermediate random variables are replace by their expectations. Suppose, for example, that there exists a measure Y for which $Z = E[X|Y]$ can be either analytically calculated or efficiently estimated. Instead of estimating $E[X]$ by averaging the X_i values, a better method is to take the means of the Z_i values. The second measure is an unbiased estimator of $E[X]$ and is guaranteed to have smaller variance than the first (see [9] Section 2.6, or [17] – the actual values for variance depend upon the specific problem). This technique is called *Conditional Monte Carlo*.

In the Quicksort study, the random variates associated with the number of *exchanges* and the number of *selection comparisons* performed at each recursive level were replaced by their expectations. As a result, the random variates representing the total number of exchanges B and selection comparisons F (which are summed over all recursive levels) had smaller variance than would occur in a straightforward implementation of Quicksort. Similarly, in the computation of Insertion Sort costs, the random variables associated with the number of *insertions* and *insertion moves* for sublists of size less than M were replaced by their expectations. Corresponding measurements D and E (representing sums over all sublists) had smaller variance than would be produced by an implementation of Insertion Sort.

In previous simulation studies of Sequential Search rules, the *request cost* (the cost of searching for requested items) was used as the simulation measure. Request cost is a random variable with value depending upon the item requested and upon the current permutation of the search list. For a fixed permutation, request cost is a random variable that has as its expectation the *permutation cost* – the average cost of searching for items in the

permutation. Replacing the measure *request cost* by *permutation cost* is another application of Conditional Monte Carlo.

In this case the decrease in variance was accompanied by an increase in the running time of the simulation program, since request cost could be computed in the time it takes to find an item whereas permutation cost took time proportional to N, the size of the search list. For one model of request probabilities (Zipf's Law), a lower bound on expected search cost per request is N/H_N. Computing permutation cost therefore increases running time by a factor of H_N per request: for this variance reduction technique to be successful, variance should be reduced by at least this much. Although improvements in variance cannot be derived because the probability distribution on permutations is not known, experimental evidence suggests that the technique is cost-effective.[1] Examples of successful tradeoffs between variance reduction and simulation time in many simulation domains are given by Hammersley and Handscomb [17]. In one remarkable example ([17] page 88), computation time was increased by a factor of 4 and variance decreased by a factor of 10^4.

Control Variates

Suppose that measure $E[X]$ is to be estimated for algorithm A. Suppose also that there is another measure Y that is positively correlated with X, and that $E[Y]$ is known; without loss of generality, let $E[Y] = 0$. Y is called a *control variate* for X. Y may be another measure of A (an internal control variate), it may correspond to some property of the input, or it may be a corresponding measure of algorithm B that is similar to but simpler than A. For every positive constant k,

$$E[X] = E[X - kY].$$
$$\mathrm{Var}[X - kY] = \mathrm{Var}[X] + k^2\mathrm{Var}[Y] - 2k\mathrm{Cov}[X,Y].$$

If the sum of the last two terms can be made negative, then $\mathrm{Var}[X - kY] < \mathrm{Var}[X]$, and a better estimator of $E[X]$ is gained. Bratley, Fox and Schrage [9] (Chapter 2.3) discuss difficulties of establishing that the variance behaves as desired and of determining the correct value for k in a given problem instance.

The intuition behind the use of control variates is that X can be "seen" more clearly if a positively-correlated source of variation can be subtracted. Control variates were used in this

---

[1] This point became moot when an algorithm for computing permutation cost in the time proportional to that of computing request cost was discovered too late to be incorporated into the simulation study – see Section 5.2.

informal sense throughout the case studies. In the Bin Packing study, the *bin count* was the measure suggested by previous theoretical analysis. The control variate *sum of the weights* was identified, and the measure *empty space* (which represents the difference between the bin count and the sum of the weights) was adopted for the simulations. In the study of First Fit Decreasing, *empty space in the last bin* became a control variate for empty space and the difference, *partial empty space*, was measured.

Paired Experiments

When the control variate comes from another algorithm, the use of control variates is called *paired experiments*. For example, suppose that algorithms A and B are to be compared in terms of measure X; in particular, the object is to estimate the expected difference $D_i = X_{A,i} - X_{B,i}$. The variance of D_i is given by

$$\begin{aligned} \mathrm{Var}[D_i] &= \mathrm{Var}[X_{A,i} - X_{B,i}] \\ &= \mathrm{Var}[X_{A,i}] + \mathrm{Var}[X_{B,i}] - \mathrm{Cov}[X_{A,i}, X_{B,i}]. \end{aligned}$$

Variance in D_i can be reduced if the covariance of $X_{A,i}$ and $X_{B,i}$ can be increased. If simulation experiments for A and B are run on independently generated sets of input, then $\mathrm{Cov}[X_i, Y_i] = 0$. In many situations, however, inputs that give high measurements for A tend to give high measurements for B. In paired experiments, different algorithms are given identical sets of inputs so that differences for corresponding trials may be computed rather than average differences for independently generated trials.

In the Search study, requests for items with low probability tends to produce high request costs for any reasonable rule. If algorithms A and B are given identical inputs at each trial it is likely that the covariance of search costs would be positive. Paired experiments were used in Search and in some of the Bin Packing experiments. In both cases, experimental evidence as well as intuitive arguments for positive correlation of measurements were available.

7.4. Placement of Sample Points

Placement of sample points can also improve the view of experimental results. Some rules of thumb are presented in this section.

Measure the largest problem size possible. A better view of the data is usually provided if

extremal parameter values are taken. An important special case is to study the largest practical problem size. Experience with the case studies demonstrated many times that measurements at larger problem sizes are generally worth the extra investment in time. Previous experimental studies of Bin Packing rules, for example, measured lists as large as 200 and 1000; simulations described in Chapter 2 studied lists of size 128,000. Many observations, such the nonmonotonic behavior in First Fit and Best Fit, could not have been seen at smaller input sizes.

Measuring large problems is also important when the theoretical model involves asymptotic analysis. Although measurements at finite input sizes can generally give only approximations to asymptotic behavior, larger input improves the accuracy of the approximation. The similarity between theoretical model and simulation results is therefore increased. In the Bin Packing, Matching, and Quicksort studies, the problem size doubled at each sample point rather than increasing by a fixed amount. This seems to be a powerful method for obtaining measurements at large N with less investment in computing time.

Sample many points within the range of parameter settings. The usual goal of algorithm analysis is to characterize some measure of algorithm performance as a function of input parameters. So that simulation results may be expressed in theoretical terms, characterization of function forms should also be the goal of simulation research. If a parameter is set only at its extremal values, assessment of the functional relationship between the parameter and measure is rarely possible. The number of intermediate sample points is of course limited by simulation cost and by the available data analysis tools. Some case study results suggest, however, that it can be worthwhile to take as many sample points as the environment will allow: the cyclicity of partial empty space in FFD packings with $u \leq 5$ would not have been discovered if, say, five values of u had been sampled in this range.

Apply stratified sampling. A recurring issue arising in the case studies was whether it is better to take discrete sample points or to take random sample points over the range of the parameter settings. In practical terms, which type of graph in Exhibit 7-1 gives more information: the left, with discrete sample points, or the right, with randomly-placed sample points?

Hammersley and Handscomb [17] (p. 554) remark that it is generally a good idea to eliminate sources of randomness in the simulation wherever possible. They and other authors of texts on simulation recommend *stratified sampling*: a better view of functionality

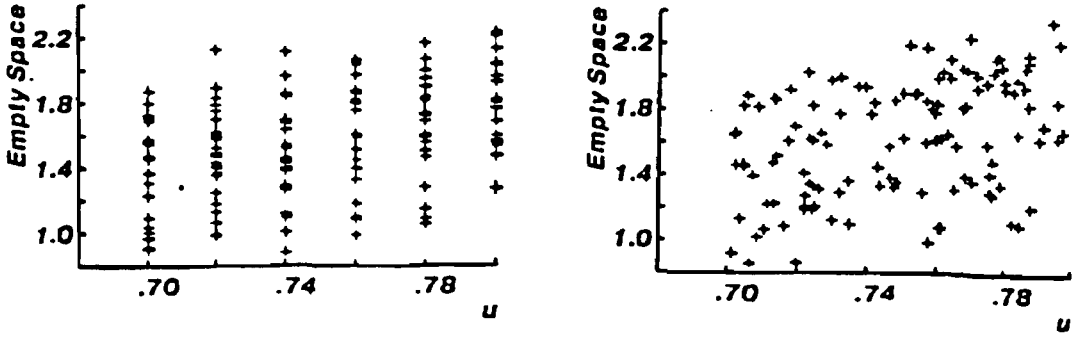


Exhibit 7-1: Placement of Sample Points

and a reduction in variance may be gained if input instances are generated so that certain input properties (specified by the parameters) occur with probability 1.

Stratification is not precisely the same as taking discrete sample points. In the Bin Packing study parameters u and n were set at discrete values. The parameter n was stratified, because every input at the sample point (n_0, u_0) was exactly of length n_0. The parameter u was not stratified, however, since the largest weight in each input instance was a random variable determined by n and u. Nevertheless, the arguments for stratification tend to support the taking of discrete sample points. As a simulation study progresses it may be useful to stratify inputs even further than described by the initial parameters: it might have been helpful in the study of FFD packings in the critical region, for example, to stratify the parameter b = *number of big items* and to generate weight lists with exactly b big items, rather than generating random lists according to n and u simply reporting the value of b each time. The relationship between topheavy lists (with large b) and bad packings might then be more clearly seen.

Design the experiments. The goal of experiment design is to determine the placement of sample points so that the most information may be gained with the least cost. Not surprisingly, the best time to design an experiment is *after* the study, when the problem is better understood. For this reason, an iterative approach to placement of sample points is important.

Of course the experimenter has to start somewhere. A *complete factorial design* is a useful starting point. This design is quite straightforward: for each input parameter, choose a few settings that span its range and establish sample points from the cross product of the settings. In the Quicksort study of fixed-T strategies the parameters are N, M, and T. Setting

$N = 10, 100, 1000$, $M = 1, 5$, and $T = 1, 3, 5, 7$ gives a 3 x 2 x 4 factorial design with 24 sample points.

An alternative to the complete factorial design is the "one-factor-at-a-time" approach: fix N and M and vary T, then fix N and T and vary M, and so forth. This approach is natural in the context of algorithm analysis – in the Bin Packing study, u was fixed to study growth in n and then n was fixed to study growth in u. Even so, the method is not generally accepted by statisticians today (see [5] or [8]), primarily because observations may be extended erroneously. Initial experiments in the Quicksort study were of the "one-at-a-time" variety: the parameter M was fixed at 1 and N and T were varied. Many of the early observations were later found not to hold in general.

7.5. Pilot Studies

It is useful to implement a *pilot study* before beginning extensive simulation. A pilot study is simply a small-scale preliminary version of the simulation program, where the simulation model is implemented in a straightforward manner with little attention to program efficiency. Inputs are generated with minimal programming effort, using the system random number generator and other system routines when possible. The object is to use the information gained from the pilot study to improve the power and quality of more extensive simulations. The information can be used in a number of ways.

First, the pilot implementation allows clarification of the simulation model and a method for checking the specification before much coding effort is invested. This is especially useful when partial analytical results already exist. As a preliminary step in the Quicksort study measurements from a pilot implementation were checked against known formulas for Median-of-3 Quicksort. The implementation failed: observed means for the five measures of interest were consistently higher than their predicted values. Closer examination revealed a number of subtle differences between the model and the implementation. For example, when sublists are smaller than the sample size, the analytical model assumes that the sample is drawn with replacement while the pilot program drew a smaller sample without replacement. Similar minor differences (which significantly affected the results) were discovered in the pilot study. Once the details of the model were clarified, the simulation measurements produced (to within 1.5%) the mean values predicted by theory.

Second, monitoring the pilot program can direct efforts for improving the running time of

later simulation programs. This simple idea was used, for example, in the implementation of the First Fit Decreasing rule. Some of the speedups were algorithmic: instead of generating a random lists of weights and sorting them by an $\Omega(N \log N)$ sorting algorithm, a variation on Binsort was used, exploiting the fact that the weights are uniformly distributed. Instead of finding the "First Fit" bin by linear scan through the bins, a heap was imposed over the bin set to find the correct bin in $O(N \log N)$ time. In addition to algorithmic improvements, a number of standard techniques for improving program efficiency (such as those described by Bentley [6]) were applied. Monitoring, for example, revealed that most of the computation time was spent in searching for the correct bin to contain each item. Careful recoding using standard techniques such as loop unrolling, code motion, and placing loop variables in registers, decreased the running time of the program considerably: the final implementation could generate and pack a list of 128000 items in just over 1 minute of machine time, a factor of four improvement over the initial implementation.

Third, the pilot studies are useful in planning of future experiments: preliminary results suggest, for example, how many trials will be necessary, how many sample points should be taken and where to space them, and what sort of distribution arises at each sample point. Such information saves a lot of trial and error in later simulations. The factorial design approach described in Section 7.4 was of considerable use in obtaining this information quickly.

Fourth, and perhaps most importantly, the pilot implementation can be saved so that final simulation programs, with different random number generators, finely tuned code, and shortcut implementations, may be compared to a straightforward version of the algorithmic model. The pilot study can therefore provide a secondary system for replication of experiments. This backup system was critical in establishing the accuracy of results in all four case studies.

A final reason for building a pilot program is that it may be sufficient for the algorithmic problem at hand: there may be no need to develop a highly-tuned implementation. Even if it turns out that the pilot implementation is sufficient, it is still a good idea to build a "backup" system to validate the simulation results. At the very least, an alternative number generator should be used.

7.6. Simulation Shortcuts

In one sense there is little need for an efficient simulation program: the experimental results will be the same no matter how long the experiments take. There are many reasons, however, for spending some effort in developing a fast simulation program. Most importantly, extensive use of an iterative approach depends upon the speed with which results are obtained: a researcher is less likely to pursue a conjecture if results require a few days rather than a few minutes. Faster simulation programs allow more trials per sample point in the same amount of time. Algorithmic improvements can allow larger problems sizes, giving a better view of the data and more insight into asymptotic performance.

Integer computation was used throughout the case studies. In the Bin Packing and Matching studies the algorithms were performed on integers ranging from $(0, 2^{20}-1]$ rather than on reals from the range (0. 1]. In general, integer arithmetic is faster and gives more accurate results than computation on reals. Also, since most generators of uniform random variates (including those used in the studies) produce integers, the cost of converting to reals was saved in the simulations.

In addition to program speedups, a powerful technique in simulation is to look for *simulation shortcuts*. The motivating principle is that a simulation of an algorithm is required, not necessarily an implementation. As a trivial example, suppose the average cost of searching for items in a random list of size N is of interest, where each item is equally likely to be requested. The naive simulation program repeatedly generates random lists and random sequences of requests and accumulates the costs of searching for requested items. The "shortcut" program generates search costs from a uniform distribution on the integer range [1,N]: although no random lists are built and no searches are performed, the results are the same. (The "super shortcut" program prints "(N + 1)/2" and stops.)

Often, partial understanding of the algorithm can be exploited to obtain shortcuts in the simulation. In the Quicksort study, the straightforward approach to simulating Quicksort would be to generate randomly-ordered lists of numbers and to Quicksort them, recording the measures of interest. In this case no random lists generated and no lists were sorted, yet the desired measurements were obtained. The shortcut was possible because tha expected values of the various measures *at each level of recursion* could be described analytically. The shortcut program exploited this partial understanding by calculating appropriate values at each recursive level rather than by simulating them. Since this simulation shortcut also happened to be a variance reduction technique, simulation efficiency was doubly improved.

The median-generation routines in the Quicksort study also incorporated shortcuts: for $T \ll N$, maintaining an ordered hash table to detect duplicates exploited the fact that the sample was drawn uniformly from the integer range and that only the median needed to be found quickly. For T near N the generation routine only considered about $T/2$ numbers before producing a median.

Early in the study of Search rules a simulation shortcut was developed for the Transpose rule by the addition of an auxiliary data structure. The straightforward implementation of Transpose maintains a list of N items; when a request is made, the item is found in the list by sequential search, the number of comparisons required to find the item is recorded, and the item is transposed with the one preceding it. The shortcut implementation maintains a second data structure (indexed by item names) that records the position of each item in the search list. When an item is requested, its position in the search list (and therefore the number of comparisons needed to find it) is found by lookup in the secondary structure rather than by sequential search. Locating the requested item and updating the two lists requires constant time per request rather than the time to search for the item. (When the measure in the simulation study was changed from request cost to permutation cost, this shortcut was no longer used.) Some information must usually be sacrificed in order to use a shortcut: in the Quicksort study, for example, no sorted list was produced as output. This was not a liability in the simulation study because it did not affect the measures of interest.

Hammersley and Handscomb [17] give many examples of simulation shortcuts in their discussion of Monte Carlo Techniques. Bentley [3] describes a simulation shortcut in his study of an algorithm for median selection. Beardwood, Halton, and Hammersley [2] make good use of a shortcut in their study of heuristics for TSP. Given a set of points generated uniformly within the unit square, the Strip heuristic divides the square into vertical strips of fixed width, connects the points within each strip, and then connects the strips. The expected tour length for the Strip heuristic is essentially the expected length within each strip multiplied by the number of strips. The shortest tour within a strip is found by connecting the points in order from top to bottom. Rather than developing a straightforward simulation of the Strip heuristic, they exploited understanding of average distances between points in a strip to calculate tour lengths without producing tours.

7.7. The Simulation Environment

A simulation study requires more than just a simulation program. Routines for random number generation are needed, data files must be managed, and programs (or packages) for data analysis must be available. Flexible and efficient simulation environments are needed to support iterative analysis. This section discusses issues of program and environment development.

One important rule in developing simulation programs is to avoid premature summarization of data. Very often in the case studies, examination of the distribution of data points at each sample point led to new insight. Simulation programs should produce measurements taken at *each trial* rather than average measurements for each sample point, so that the experimenter can see the raw data.

Another rule that proved useful in the case studies is to produce results that are readable by other programs. This principal is one component of the "Unix" style of program development; see Kernighan and Pike [18] for more discussion of this approach (Unix is a trademark of Bell Laboratories). The output of a simulation program, if it is to be easily analyzed and manipulated, should not be cluttered with column headings and annotations; this is especially true if the results are to be submitted to a data analysis package.

Statistical Analysis Packages

The available statistical analysis package influences the arrangement of experimental results in data files. Tools used in the case studies included the statistical analysis packages *S* (developed at Bell Laboratories) and *Minitab* (developed at Penn State University) as well as the graph-drawing packages *Plot* (developed by Ivor Durham for use in the Computer Science Department at CMU), and *Grap* (a preprocessor for the Troff typesetting system).

All of these packages are *column-oriented*: that is, commands are typically expressed in the following format.

- Plot the data in column 1 against corresponding values in column 2.
- Compute a multiple least-squares regression using the values in column 2 and 3 to predict values in column 5.
- Assign to column 6 the logarithms of values in column 3.

If a statistical package is available then it is useful to develop simulation programs that give results in a format compatible with the package. In the case studies results were generated so that each row of data corresponded to a single trial. The leftmost fields in each row gave sample point settings and the rightmost fields gave values for the measures of interest. For example, the following data are from an experiment in the Bin Packing study. The columns, from left to right, correspond to the *name of the packing rule*, *n*, *u*, *number of bins packed*, *empty space*, *empty space in the last bin*, and *number of big items*. Each line gives results for one trial: two sample points are recorded.

```
FFD     125  1.000000 72 5.358297 0.468258 69
FFD     125  1.000000 63 2.500774 0.336832 58
FFD     125  1.000000 68 4.348960 0.502865 68
FFD     125  1.000000 65 2.633522 0.488351 63
FFD     125  1.000000 61 1.705485 0.805012 57
FFD     250  1.000000 137 6.808572 0.999558 137
FFD     250  1.000000 129 2.943276 0.315850 121
FFD     250  1.000000 130 4.867452 0.997307 129
FFD     250  1.000000 131 4.696441 0.459429 129
FFD     250  1.000000 145 9.049346 0.488392 142
```

Statistical packages can be of great use in analysis of experimental results and can be critical to the development of an interactive style of analysis. On the other hand, much can be accomplished with less sophisticated tools. Portions of the analysis described in the case studies were performed on the Plot graphical-display package. Plot is not a statistical package: its primary capability is to read a 2-column file of data and to plot values in the first column against those in the second column. Plot commands deal with modifications of the graphical display. Although the functionality of Plot is limited in comparison to a statistical analysis package, much of the difference was made up by the *awk* filter, a standard Unix facility. In the awk language, each command comprises a *pattern* and an *action*: if the current input line matches the pattern, then the action is taken. The pattern-matching language allows comparison and algebraic manipulation of field entries. The action part is as powerful as most programming languages and supports associative arrays. Awk was regularly used for many of the functions available in statistical packages, and the output of the awk filter was passed to Plot for graphical display of the results. At times, the filter alone was sufficient for quick analysis of small data sets: awk performed tedious manipulation and summarization of the data and produced results that could be quickly recorded on graph paper.

Generating Random Inputs

A good source of uniform random variates is needed in all simulation problems. A huge literature exists describing and evaluating algorithms for generating random variates. See for example Knuth [19], Brately, Fox and Schrage [9], Fishman [15], or most texts on simulation. The primary generation algorithm for uniform variates used in the case studies is from Knuth [19] (3.3.2, Algorithm A, Second Edition). Exhibit 7-2 gives the algorithm implemented as a C macro. The 55-element array Rand was initialized by 55 calls to the BSD Unix 4.1 system random number generator, a linear congruential generator producing integers in the range $(0,2^{31}-1]$.

```
#define Maxrand (1 << 30)
int Rand[55];
int K,J;
.
#define RAND(X)         X = Rand[K] + Rand[J];                  \
                        if (X >= Maxrand) X -= Maxrand;         \
                        Rand[K] = X;                            \
                        if (K == 0) K = 54; else K--;           \
                        if (J == 0) J = 54; else J--;           \
```

Exhibit 7-2: Uniform Number Generator

The secondary generation method, used in the pilot studies and in backup implementations, was some form of linear congruential generator. The secondary studies varied among the problem domains: some were performed on a Radio Shack TRS-80 personal computer, which has a system linear congruential generator that generates reals from the unit interval. Other backups systems used the system generator for BSD Unix 4.1.

The Bin Packing and Matching studies required the generation of *order statistics* of uniform variates: that is, sorted lists of numbers drawn independently and uniformly from a specified range. A number of approaches might be used: for example, the variates could be generated and then sorted by Quicksort. Since the numbers are known to be uniformly distributed, Bucketsort might be more appropriate. Nijenhuis and Wilf [22] also give a clever algorithm for generating N integers in linear time and linear space. Bentley and Saxe [7] present two linear-time algorithms for generating the order statistics of uniform reals. Although efficient, the latter two were not considered for the case studies because they produce real numbers rather than integers (integer computations were used throughout). A small study of the running times for the generation routines revealed that a variation on Bucketsort was most efficient.

Many techniques exist for generating random variates from specific non-uniform distributions, as well as general techniques for arbitrary distributions: see [9], [15], or [19] for a good discussion. A well-known general method is *inversion*: if F is an invertible distribution function, then setting $X = F^{-1}(U)$ (for U a uniform variate) produces variates with distribution F. If the inverse of F is not easily computed, then $F^{-1}(U)$ may be approximated by a *tabular inversion* method, where the i^{th} entry in the table contains the pair $[F(x_i), x_i]$. A simple *rejection* approach generates points uniformly in the unit square and rejects any point that lies above the specified density curve; if the point is below the curve x-coordinate of the generated point is reported. In general, rejection methods, generate points in a region close to the probability curve so that the number of rejections is small. A third method, the method of *aliasing*, was adopted for generating of Z^{λ} variates in the Search study. Although the method requires a table of size $2N$ and setup time $O(N)$, (for N the range of possible values), it uses constant time per variate generated.

7.8. Analyzing Simulation Results

This section presents tools and guidelines for analyzing experimental results. Just as the study of algorithms presents special problems in development of simulations studies, this domain present special types of data-analysis problems. Some of the properties listed below are typical of simulation studies in general; some are features of algorithmic domains.

- Sample points are usually chosen to correspond to discrete, evenly-placed spots in the space of parameter settings.
- Since parameters are often stratified, plotting a measure against a given parameter results in *slices* of data points. At each slice the measurements correspond to a random sample from some (usually unknown) distribution.
- The relative efficiency of simulation in this domain (compared to traditional simulation problems) allows huge amounts of data to be generated.
- Measurements of algorithms usually (but not always) move smoothly with respect to parameter settings.
- A common goal is to characterize functional relationships between measures and parameters. Comparison of algorithms at fixed sample point is also of interest.
- Usually, little is known about the functional relationship. Even if theoretical bounds exist, they often describe asymptotic behavior and may not be appropriate for the domain of the experiments.

Many analytical tools were applied during the case studies that were not mentioned in Part II. The remainder of this section describes a number of tools that were particularly useful. First, general techniques for studying any data set are discussed. Sections 7.9.1 through 7.9.3 describe tools for specific analysis problems.

Readers familiar with traditional techniques of experimental statistics will realize that very few were mentioned in the case studies: there were no formal experimental designs, no analysis of variance tables, few instances of hypothesis testing, and limited applications of regression analysis. Instead, tools of *descriptive* statistics were used extensively. Statisticians specializing in *Exploratory Data Analysis* (EDA) distinguish between *confirmatory methods* – where statisticians apply powerful analytical tools that rely upon a mathematical model that closely describes the data in order to make inferences and to assess experimental errors – and *descriptive methods*, which are used to obtain a good view of the data and to produce summaries that are easily grasped. In the past, descriptive statistical methods have been limited to elementary tools such as histograms and box plots. EDA provides a more sophisticated set of tools for describing sets of numbers.

There are a number of reasons for the emphasis on descriptive statistics here. Tukey's seminal text, *Exploratory Data Analysis*, was published in 1977: the approach is relatively new and, it appears, not very well known to computer scientists. The techniques deserve better exposure to this audience. Also, at least for the case studies, the questions of interest were answered more naturally by EDA approaches than by inferential methods.

Experience with the case studies suggests that much insight can be gained by examining the data produced at each trial rather than averages for each sample point. Data sets at varying sample points reveal convergence rates and changes in variance as well as distributional properties. Graphical techniques for data analysis become very important in this data-rich domain, since graphs are invaluable for clearly and concisely presenting huge amounts of information. Consider, for example, the size and unreadability of the tables that would be required to represent the information about First Fit Decreasing packings contained in Exhibit 2-11. Graphs also allow functional relationships and distribution properties to be more easily seen.

Summarizing and Transforming Data

Although the raw data should be examined, it may be helpful to calculate *summary statistics*

to represent the data at each sample point. The sample mean is usually a good choice for representing the location of a data set, especially in studies of expected-case behavior. Statistics for describing dispersion, such as the standard deviation or the variance, are well known.

If the distribution at a sample point is skewed (trailing off at either high or low values), or if a bimodal distribution appears, then alternative summary statistics may more appropriate. One approach used often in the case studies was to record certain order statistics such as the median and extremes of the data values. The *quartiles* are also useful: the high quartile of a data set is the data value that is smaller than 25% of the set, and the lower quartile is greater than 25% of the data. Half of the data values therefore fall between the quartiles.

A *transformation* of a set of numbers is achieved by applying some function to each value in the set; common transformations include the logarithm and the square root. Transformations may be applied for a number of reasons. If the values represent units of time, for example, taking reciprocals of the measurements converts "infinite time" to "zero speed," which can be easier to deal with.

Tukey [24] identifies a number of "types" of data, including *counts and amounts*, *times*, *fractions*, *proportions of a whole*, *balances*, and *grades*. Different types require different analytical tools. Simulation results for algorithmic problems generally take the form of counts and amounts: measurements arising in the case studies, such as amount of empty space, number of k-bins, number of comparisons, number of pairs matched, and number of recursive stages reached, are all examples of this data type.

Counts and amounts have positive values with arbitrarily high upper bounds. If the ratio of the highest value to the smallest is large, then the high values will dominate the view of the data. Transforming to logarithms will "spread out" the low values so that they may be more easily seen. Tukey [24] (p. 57) remarks that counts and amounts generally profit from logarithmic transformation (he calls it reexpression) unless the ratio of the largest to the smallest value is near 1.

Another type of data arises in the study of residuals to regression fits: *balances* have positive and negative values and are generally grouped around zero. Balances usually require no transformation.

The following subsections survey specific data analysis tools that proved useful in the case studies, with emphasis on graphical and EDA tools. As before, familiarity with elementary statistical analysis is assumed. Many of the techniques described here may be found in Tukey [24] and Chambers et al [10]. Cleveland [11] gives an excellent discussion of issues of graphical style.

7.8.1. Looking at Distributions

At a fixed sample point the measurements represent a random sample drawn from some probability distribution. In algorithmic problems the form of this distribution is usually unknown. There are many reasons for looking at the distribution of measurements: How large is the range of the data? Are the points arranged symmetrically about their mean, or are they skewed? How are the measurements distributed at this sample point? Summary statistics can suggest the location, shape, and spread of the data set. In addition, a number of graphical tools are available.

The most familiar way to display a distribution is by a histogram. Recently, statisticians have argued against the use of histograms in data analysis – the essential drawback is that the visual message depends greatly upon the choice of graphical parameters such as the width and cutoff points for the bars, rather than on the data itself. Chambers et al [10] and Cleveland [11] discuss histograms and their weaknesses.

Exhibit 7-3 presents a number of alternatives to the histogram using data from the Search study. The graphs depict permutation costs for the Move-to-Front rule in 100 trials at the sample point $N=6$, $\lambda=1$, and $T=5$. A partial list of the results, showing the 5 highest and 11 lowest values observed in 100 trials, are presented at the left of the exhibit.

A compact and informative representation is given by a *one-dimensional scatter plot*. Graph 7-3-a reveals that observations range between approximately 2.5 and 4.5, and that about 90% of the data is below 3.5. Also, the distribution is densely concentrated towards the bottom of its range and sparse at the top. This type of graph was used extensively in the case studies for comparisons of distributions among sample points. Two potential disadvantage with one-dimensional scatter plots is that the density of the points may overwhelm the graph-drawing technology, producing a blob of ink rather than distinct marks, and that duplicate values are overwritten. Methods for avoiding such problems by techniques such as "jittering" (giving each point a random horizontal position within a small range) are described in [10] and [11].

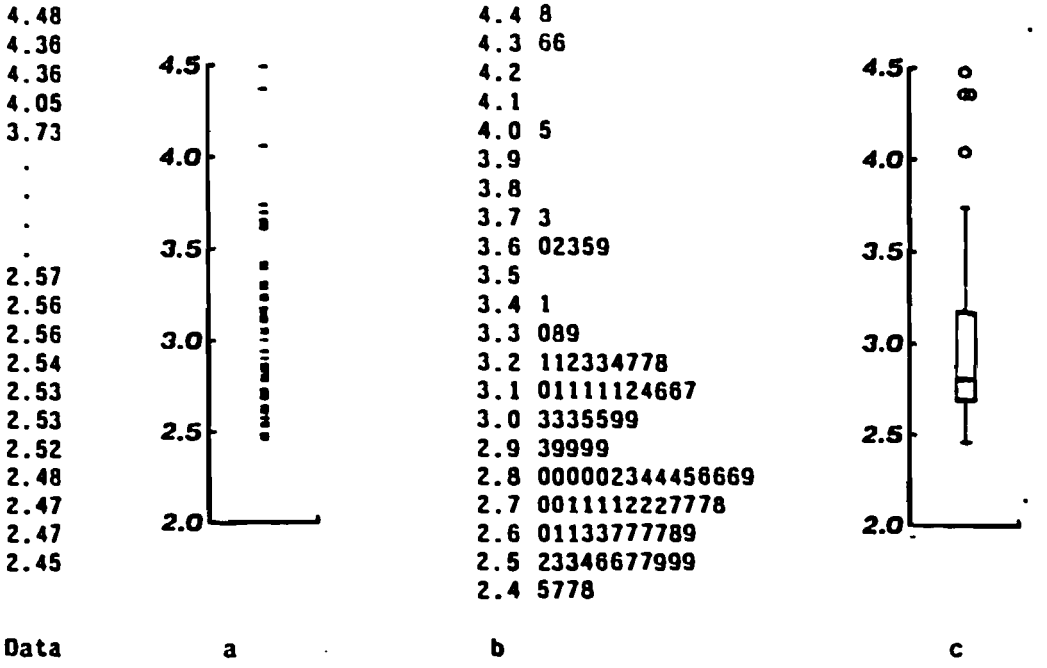


Exhibit 7-3: Displaying Distributions

The *stem and leaf chart* (Graph 7-3-b) is a combination graph and table. The high-order digits of the data values are written in the *stem* (the horizontal column), and low-order digits are recorded in the *leaves* (the row entries). This chart displays the trailing off at high values more clearly and gives a better view of the shape of the distribution. From this chart we see that most of the values are concentrated between 2.45 and 3.41, with ten stray values above. The median value is at 2.80, the extremes are 4.48 and 2.45, and the quartiles are 3.16 and 2.68. This data is clearly skewed towards the bottom. The distribution also appears to display some bimodality, with peaks around 3.1 and 2.8. The most common values occur between 2.80 and 2.89.

Stem-and-leaf charts provide an excellent way to record a data set and to display distributional properties. Also, the data is not obscured by limitations of graphical technology. The view obtained from these charts can often be improved by transformation of the data: transformations can be used to induce symmetry in the distribution and to scale results for better comparison among data sets. In addition, order statistics can be easily found, since the data points are presented in sorted order. One disadvantage of stem-and-leaf charts is that they can take up a great deal of space.

The *box plot* is probably as well known as the histogram. Unlike the first two graphs of Exhibit 7-3, box plots are graphical summaries of the data set and do not display all the measurements. In Graph 7-3-c, the ends of the box correspond to the quartiles of the data set and the horizontal bar to the median. A common problem arises in deciding which points to include in the vertical bars and which to mark as outliers. A standard rule of thumb (see [24] or [10]) is to compute the *interquartile range* II - the difference between the quartiles - and to plot values more than $1.5II$ away from the quartiles as outliers. In Exhibit 7-3, the interquartile range is $II = 3.16 - 2.68 = 0.48$, so points greater than $3.16 + 0.48(1.5) = 3.88$ are plotted as outliers. There are no low outliers. Note that duplicate values in the outliers are "stacked" - placed side by side - so that they may be seen.

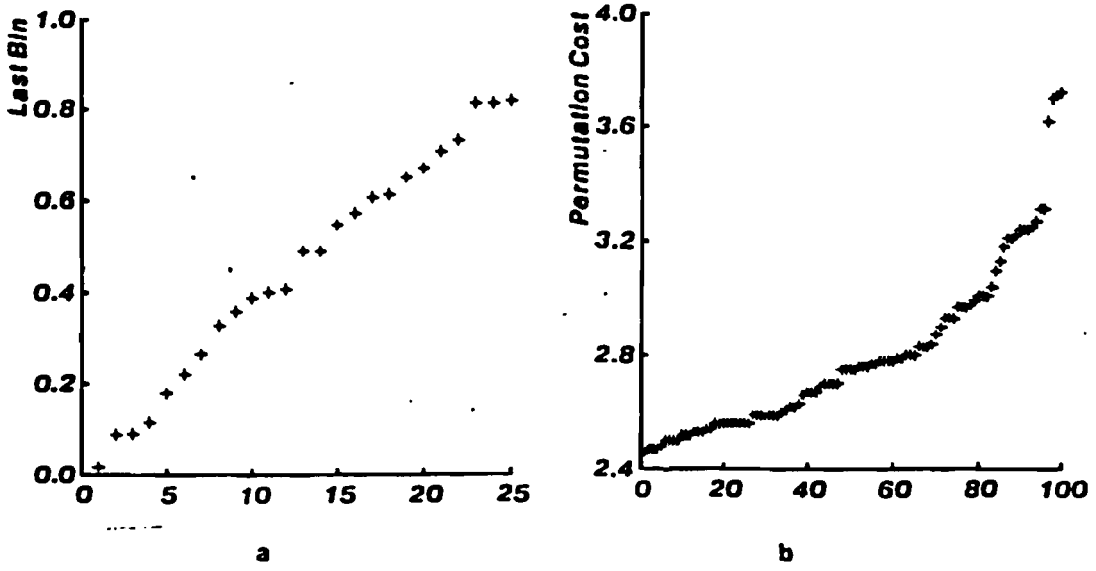


Exhibit 7-4: ECDF Plots

Another graphical technique for examining the distribution of a data set is to plot the values against their ranks, producing an *empirical cumulative distribution* plot (ECDF, also known as a quantile plot or a cumulative frequency diagram). Guidelines for interpreting ECDF plots are given by Chambers et al [10], and by Mosteller, Fienberg, and Rourke [21]. If the curve in an ECDF plot is generally straight, then the data are nearly uniformly distributed in their range: Graph 7-4-a presents an ECDF plot for empty space in the last bin in 25 FFD packings at the sample point $n = 128000$, $u = 0.5$.

Graph 7-4-b presents an ECDF plot for the data from Exhibit 7-3. The concentration of

values at the bottom of the distribution is indicated by slow growth in the left side of the graph; quick growth on the right side marks the outliers at the high end. In ECDF plots the slope of the curve corresponds to the density of the data points. Although it is more difficult to see peaks in the density function, ECDF plots give a good view of the symmetry and spread of a set of data. Also, statistics such as the means and quartiles are easily found from ECDF graphs.

7.8.2. Comparing Sets of Data

Comparison of data sets at fixed sample points is a common task in simulation studies of algorithms. Measurements for two algorithms at the same sample point can be compared to determine if one consistently outperforms the other. Data from adjacent sample points can be compared to determine if the measure changes as parameter settings vary. Data from extremal parameter settings can be compared to measurements at middle settings to determine if a horizontal straight line is produced. Also, measurements from secondary implementations can be compared to corresponding values from the primary simulation program to determine if the results are statistically equivalent.

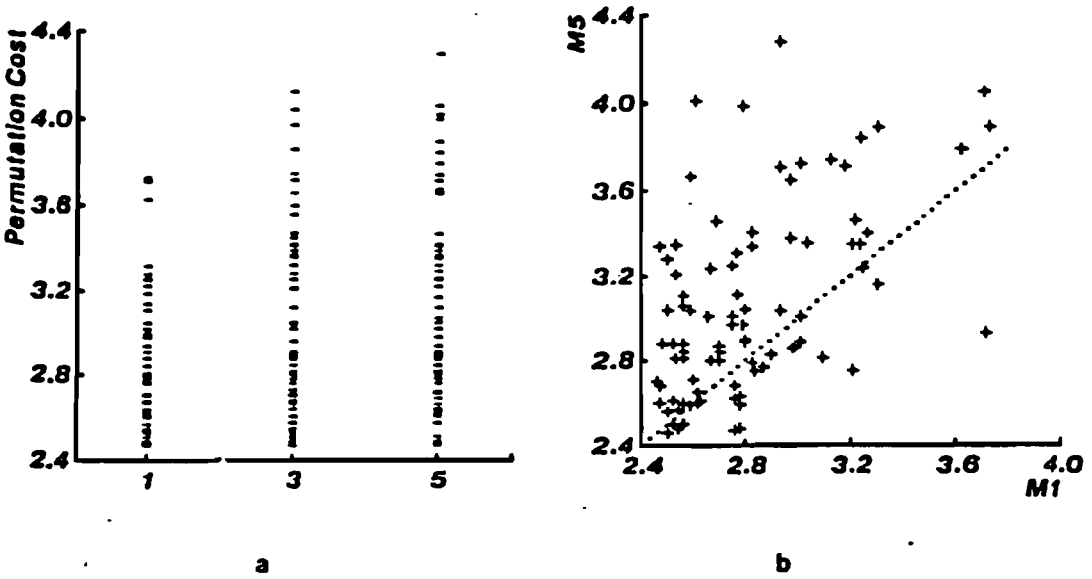


Exhibit 7-5: Comparing Data Sets

A simple graphical technique for comparing data sets is by juxtaposition of one-dimensional

scatter plots. For example, this method was used to study the critical region for the First Fit decreasing rule (Exhibit 2-11). Graph 7-5-a presents a comparison of the sequential search rules M1, M3, and M5, for 100 trials at the sample point $N=6$, $T=25$, $\lambda=1$, $G=1$. The graph suggests that M1 has much smaller variance than the other two rules as well as smaller mean. All three rules have similar low bounds, but upper bounds vary among the rules. The distribution for M3 appears to be skewed towards the bottom to a greater extent for M5. Juxtaposition of stem-and-leaf charts, similar to that in Exhibit 5-10, was also a frequently-used analysis tool.

Another technique for comparing data sets is to plot corresponding values against each other; this approach is especially useful for comparing results of paired experiments. In Graph 7-5-b the x-coordinate of each point corresponds to permutation cost for the M1 rule and the y-coordinate to permutation cost for the M5 rule for 100 paired trials at the sample point given above. The dotted line gives the identity function $y = x$. Most of the points are above the line, indicating that M5 generally has higher cost than M1 at this sample point. There appears to be a small but not overwhelming positive correlation between costs for the two rules.

Pairs of numbers can be also compared by reporting either their difference or their ratio in terms of some input parameter. Often in the case studies the clarity of the view was influenced by the choice of comparison method. Usually, differences gave a much better view than ratios, even when ratios (for example solution ratio in the Bin Packing study) were suggested by theoretical analysis.

Gnanadesikan and Gustafson [16] note that significantly different sizes in numerator and denominator can give a bimodal distribution in ratios (which usually causes difficulties in summarization and analysis). In visual displays the human eye is better at judging distances (corresponding to differences) than proportions (corresponding to ratios). Tukey [24] suggests that if ratios are necessary to the analysis, then it is probably better to take logarithms of the data and to study differences in the new metric (which correspond to ratios in the original scale), and then to translate conclusions back to the original scale.

In the Bin Packing study, empty space (which bounds the difference between heuristic bin count and optimal bin count) lead to much stronger results than previous studies which measured the bin ratio. The bin ratio, as it approaches some asymptotic constant, tends to change very slowly in n. This obscures the view of the data, which is best when growth over sample points is large compared to variation within sample points. In contrast, empty space

(for $u = 1$) grows as $n^{1/2}$, so change in terms of n is easily seen. Bentley and Faust [4] were the first to measure empty space instead of bin ratio for the Bin Packing problem.

7.8.3. Assessing Functional Relationships

The most common goal of a simulation study in the domain of algorithm analysis is to characterize the functional relationship between a measure and the parameters. This section presents techniques for studying functional forms.

Using Regression Analysis

Although standard techniques of regression analysis are well known, examples of previous work (from Section 1.2) demonstrate that this powerful analysis tool should be interpreted with care. There are at least two reasons that a regression model might be interpreted erroneously in algorithmic problems. First, the precise functional relationship between measures and parameters is not usually known beforehand. An approximate model that appears to describe the data must therefore be used. Second, even if the model is known, it may not be appropriate: theoretical results are usually expressed in terms of asymptotic order-of-magnitude bounds, whereas experimental measurements at finite problems sizes correspond to a curve approaching its asymptote.

In any reporting of regression results, it is not sufficient to simply state the model used and the coefficients obtained. Regression results should always be accompanied by a precise description of the variation between the data and the regression fit. Standard tools for checking model accuracy and for describing model deficiencies are found in texts on experimental statistics such as [5], [12], and [21]. Proper interpretation of regression results cannot be made without correct application and reporting of these results as well.

A useful tool for studying deficiencies of a regression fit is a graphical display of the residuals. Since residuals represent the difference between the data and the fit, properties of such a graph can indicate deficiencies in the model. Observing generally straight horizontal residuals does *not* necessarily mean that the correct model has been found, however. The study of First Fit gives an example: although the residuals from the fit $y = x^{0.7} + b$ display no marked curvature (Exhibit 2-1), Shor [23] later proved asymptotic bounds of $O(n^{2/3}(\log n)^{2/3})$ and $\Omega(n^{2/3})$.

Although regression provides a useful descriptive tool, care must be taken in extending the fit to behavior outside the range of the experiments. At times in the case studies, a poor fit gave more information than a good fit: if the residuals curve upward, for example, then the data is growing more slowly than the fit, suggesting that the fit might gives an upper bound on the data. (Of course there is always the possibility that the data increases at a much faster rate for problem sizes larger than those measured: see Eppinger [13])

The regression models discussed in Part II are all linear models (sometimes on logarithmic scales). At times during the analysis, attempts were made to apply more sophisticated models. In most cases it was very difficult to determine if one model was any better than another. I therefore used simple linear models in discussion of the case studies, which are more obviously seen as descriptive tools than as factual descriptions of functional relationships.

EDA Techniques

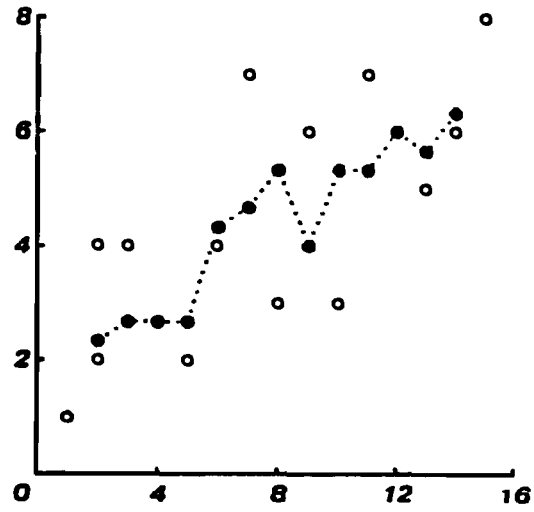
Examining residuals from a regression fit is analogous to the general approach for studying functions by EDA methods. First, look at the *smooth*: transform the data, fit models, or use other techniques to get an idea of the general relationship between the measure and parameters. Then look at the *rough*, the variation between individual data points and the general trend.

Texts on exploratory and graphical methods present many techniques for *smoothing* the data points. Smoothing gives a better view of the general relationship between two data sets. Exhibit 7-6 gives an example of smoothing for a made-up set of data. In this example, the y-coordinates are smoothed by taking means of every three values. Graph 7-6-b presents the original and the smoothed data.

Smoothing did not appear to be generally helpful in the case studies. First, measurements of algorithmic behavior tended to be fairly smooth anyway, so there were few opportunities to apply this technique. Second, the data usually appeared in slices: smoothing was either trivial (taking means at each sample point) or unnatural (taking means across sample points or within sample points). Third, although smoothing gives a good picture of the relationship between variables, the smoothed curve has no concise mathematical description. In the case studies even an inaccurate regression fit was preferred because the fit as well as deficiencies could be characterized in functional terms. At times I used smoothed lines for quick

x	y	Smooth
1	1	
2	2	2.33
3	4	2.67
4	2	2.67
5	2	2.67
6	4	4.33
7	7	4.67
8	3	5.33
9	6	4
10	3	5.33
11	7	5.33
12	6	6
13	5	5.66
14	6	6.33
15	8	

a



b

Exhibit 7-6: Smoothing

assessment of regression models: rather than inspecting residuals, a smoothed fit was compared to a regression fit and relative curvature was then interpreted in a manner similar to that for residuals.

Graphical Techniques

An important use of graphs in studying functional relationships is in analysis of residuals (described earlier). Another common graphical technique for studying functional relationships is by the *scatterplot*, where pairs of values are plotted for a view of the relationship between the two.

A standard technique of EDA is to transform the data until a straight line is produced on the graph. The nature of the required transformations can then suggest functional relationships between the two data sets. For example, if a straight line is produced by squaring the y-values, then y grows as the square-root of x. A number of useful transformation rules are presented by Tukey [24] (Chapter 3). Fishman [15] (p. 337) gives a large table of rules of the form: if the true functional relationship is $y=f(x)$, then transforming y by $g(y)$ or x by $h(x)$ will produce a straight line.

Changing the scale of one or both of the axes is also a kind of transformation. Many of the graphs in the case studies have logarithmic scales. Logarithmic scales were used to "even out" the data, especially when input sizes doubled at each sample point. Logarithmic y-scales were often used when the ratio of the largest to the smallest value was high.

The rules for interpreting graphs with logarithmic scales are identical to those for transformed data. If the y-scale is logarithmic and a straight line is produced, for example, then y-values are growing exponentially as a function of x. If a straight line is produced on with a logarithmic-x scale, then y-values are growing as the logarithm of x. If both scales are logarithmic and a straight line is produced, then a power law is suggested: $\log_e(y) = a\log_e(x) + b$ implies $y = x^a \cdot e^b$.

Very often in algorithmic problems more than one input parameter is identified: how can functions of more than one variable be displayed in two-dimensional graphs? A number of options are available. Graph 2-10-b, for example, is a *coded* scatterplot from the FFD study. In this graph data points are coded by symbol to correspond to their u values, and are plotted against N. This graph allows easy comparison among the u values because the curves are superimposed. Usually such graphs require some summarization of data (by taking means in this example) so that the different curves may be easily seen.

Graph 2-10-a is a *multiple scatterplot*, an alternative to the coded scatterplot for displaying measurements in terms of two variables. Multiple scatterplots allow comparison of all the data rather than just summaries: a separate panel is produced for one parameter, and the data points are plotted against the other parameter. For easier comparison between panels, the scale should be identical in each. If this is not practical (because variance or mean differs widely among panels), then the scale should be clearly marked. Graph 2-10-a shows a multiple plot where differences in scale are indicated by horizontal lines within the panels.

Multiple scatterplots were used extensively in the case studies, especially in the studies of Quicksort and Sequential Search. A generalization is the *scatterplot matrix* of n rows and columns, where each i^{th} row/column corresponds to a variate (corresponding to either a parameter or a measure). The panel with index (i, j) displays variate i plotted against variate j.

7.9. Summary

Chapter 6 identifies four general principles for simulation analysis of algorithms: match experimental results to algorithmic models; find a good view of the data; analyze rather than measure the data; and iterate theoretical and experimental approaches. This chapter discusses issues and procedural steps that arise and presents techniques and guidelines for approaching these four principles. To summarize this chapter, the following list presents some guidelines for simulation research in the domain of algorithm analysis.

- Choose a measure that is both well-defined and practical for experimentation. Alter the theoretical model if necessary.
- Change the measure to obtain more detailed views as the study progresses.
- Ensure correctness of the simulation program by comparison to known formulas, by applying standard program verification and validation techniques, and by consideration of limitations imposed by machine precision.
- Replicate the experiments. At the very least, change the random number generator. Even better, alter the implementation, machine, and programming language.
- Apply variance reduction techniques such as Conditional Monte Carlo, control variates, or paired experiments. Make sure that the variance reduction is cost-effective.
- Measure the largest problem sizes possible. Doubling the input size at each sample point seems to be an efficient way to proceed.
- Sample many points within the range of the parameter values.
- Stratify the parameters to reduce randomness in the simulation. At later stages in the study it may be useful to stratify input instances beyond those properties described by parameters.
- Start with a complete factorial design with a few of settings per parameter, and progress to denser samples. Be careful with "one-at-a-time" approaches to experimental design.
- Implement a pilot study before beginning extensive simulation. Use the pilot implementation to check details of the model, to suggest coding improvements, to direct choices of sample points and trials, and to serve as a secondary system for replication of experiments.
- Efficient simulation programs are worth the effort: new ideas are likely to be pursued when results are obtained quickly.

- Exploit simulation shortcuts. The object is not to implement an algorithm, but to simulate its behavior.

- The simulation program should produce unsummarized data as much as the available technology will allow. Summarization and manipulation of results should not be performed before the researcher sees the raw data.

- For compatibility with many data analysis packages, produce data files with results of one trial per row. Input parameters should be listed on every row.

- Tools of Exploratory Data Analysis appear to be particularly useful in this domain. Read Tukey [24].

- Simulations of algorithms allow generation of large amounts of data: make extensive use of graphical tools.

- Summarization and transformation of results can give more manageable information. Measurements of algorithmic performance tend to be expressed as counts and amounts, suggesting logarithmic transformations.

- Tools such as the one-dimensional scatter plot, the stem-and-leaf chart, the box plot, and the empirical cumulative distribution chart are useful for studying distributions. Histograms are not recommended by modern statisticians. Techniques such as jittering may be applied to improve graphical clarity.

- Sets of data may be compared by juxtaposition of one-dimensional scatter plots, by plotting paired data values as points, or by examining the ratios or difference between paired data. Differences tend to give a better view of the data than do ratios.

- Be careful when interpreting regression analysis. Always examine the residuals.

- The EDA approach to studying functional forms is to look at the smooth and then look at the rough. Nonparametric smoothing techniques, however, were rarely of use in the case studies.

- Apply transformations to induce symmetry in the data set, to even out counted data, or to obtain a straight line in plots of functional relationships.

- Multiple scatterplots and coded scatterplots may be used to examine measurements in as functions of more than one parameter.

References

[1] N. R. Adam and A. Dogramaci, Eds.
Current Issues in Computer Simulation.
Academic Press, 1979.

[2] J. Beardwood, J. H. Halton, and J. M. Hammersley.
The shortest path through many points.
Proceedings of the Cambridge Philosophical Society 55:299-327, 1959.

[3] J. L. Bentley.
Progamming Pearls: Selection.
Communications of the ACM 28(11), November, 1985.

[4] J. Bentley, J. Faust.
Unpublished notes on simulations of FFD.
1980.

[5] J. L. Bentley, D. Haken, R. W. Hon.
Statistics on VLSI Designs.
Technical Report CMU-CS-80-111, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213, April, 1980

[6] J. L. Bentley.
Writing Efficient Programs.
Prentice-Hall, 1982.

[7] J. L. Bentley and J. B. Saxe.
Generating sorted lists of random numbers.
ACM Transactions on Mathematical Software 6(3):359-364, September, 1980.

[8] W. I. B. Beveridge.
The Art of Scientific Investigation.
Vintage Books, New York, 1957.

[9] P. Brately, B. L. Fox, and L. E. Schrage.
A Guide to Simulation.
Springer-Verlag, New York, 1983.

[10] J. M. Chambers, W. S. Cleveland, B. Kleiner, and P. A. Tukey.
The Wadsworth Statistics/Probability Series: Graphical Methods for Data Analysis.
Duxbury Press, Boston, 1983.
Hardback version published by Wadsworth International Group, Belmont, California.

[11] Cleveland, W. S.
The Elements of Graphing Data.
Wadsworth Publishing Company, 1985.

[12] M. H. DeGroot.
Probability and Statistics.
Addison-Wesley Publishing Company, Reading, MA, 1975.

[13] J. Eppinger.
An empirical study of insertion and deletion in binary trees.
Communications of the ACM 26(9), September, 1983.

[14] W. Feller.
An Introduction to Probability Theory and its Applications.
Wiley and Sons, New York, 1971.

[15] G. S. Fishman.
Concepts and Methods in Discrete Event Digital Simulation.
John Wiley & Sons, New York, 1972.

[16] M. Gnanadesikan and H. W. Gustafson.
Properties of Performance Measures.
1985.
Summary of poster presentation. Gnanadesikan is at Farleigh Dickinson University, Gustafson at AT&T Corporate Headquarters.

[17] J. M. Hammersley and D. C. Handscomb.
Monte Carlo Methods.
Wiley & Sons, New York, 1964.

[18] B. W. Kernighan and R. Pike.
The Unix Programming Environment.
Prentice-Hall, 1984.

[19] D. E. Knuth.
The Art of Computer Programming: Volume 2, Seminumerical Algorithms.
Addison-Wesley Publishing Company, Reading, MA, 1973.

[20] I. Miller and J. E. Freund.
Probability and Statistics for Engineers.
Prentice-Hall, Inc., Englewood Cliffs, NJ, 1977.

[21] F. Mosteller, S. E. Fienberg, R. E. K. Rourke.
Beginning Statistics with Data Analysis.
Addison-Wesley, Reading, MA, 1983.

[22] A. Nijenhuis and H. S. Wilf.
Combinatorial Algorithms for Computers and Calculators.
Academic Press, New York, 1978.

[23] P. W. Shor.
The average-case analysis of some on-line algorithms for bin packing.
In *Proceedings, 25th Symposium on Foundations of Computer Science*, pages 193-200. IEEE, October, 1984.

[24] J. W. Tukey.
Addison-Wesley Series in Behavioral Science: Quantitative Methods: Exploratory Data Analysis.
Addison-Wesley Publishing Company, Reading, MA, 1977.

Chapter 8
Conclusions

This thesis presents four case studies in experimental analysis of algorithms, along with a discussion of principles and techniques for experimental research. These are all studies of *algorithms*, not of *programs*. Perhaps because algorithm analysis is primarily a mathematical discipline, there has been no tradition of experimental research in this domain. In sciences with strong experimental traditions, fundamental principles such as rigorous analysis of results, and replication of experiments are well-recognized and regularly applied. Much of this tradition can be applied successfully to algorithmic problems.

On the other hand, although simulation has been applied in diverse areas such as economic forecasting, analysis of weather patterns, and benchmark testing of computer operating systems, the goals and procedural issues presented by algorithmic problems are in many ways atypical. For example, the underlying system is relatively simple: even a complex heuristic algorithm is likely to have a cleaner mathematical description than, say, an economic model. Algorithms also tend to have inexpensive implementations and relatively few parameters, so much more data can be gained per unit of computing effort. It is not obvious, however, that the questions posed in algorithmic studies are naturally answered by traditional experimental methods: standard tools of statistical analysis (such as analysis of variance) begin by assuming the functional relationship between input properties and performance measures, while the usual goal of algorithm analysis is to discover that functional relationship.

Nevertheless, the case studies demonstrate that simulation can provide a powerful tool for gaining insight into difficult analysis problems. Although the problems in the case studies have received a great deal of previous attention, many new results were gained by experimental methods. The simulation results led to new theorems, new arguments, and new insight, as well as to precise measurements and characterizations of algorithmic performance.

The limitations of experimental research in this domain are real: measurements at a finite set of sample points do not necessarily lead to theorems. While the difficulties can not be eliminated, much can be achieved by exploitation of simulation techniques, creative application of analysis tools, and an approach that iterates experimental and theoretical analyses.

8.1. Contributions of the Thesis

The main contributions of the thesis take two forms: new results in the case study domains, and a discussion of issues and techniques for improving simulation studies of algorithms. The following list restates the research goals from Section 1.3 and gives references to sections of the thesis that address each goal.

1. *To demonstrate that simulation can provide a useful, general tool for developing new understanding of algorithms.* Chapters 2, 3, 4, and 5 present results for the case studies. A list of specific contributions in these areas appears in Section 6.1.

2. *To identify common problems and assess the applicability of this approach.* Section 1.2 gives a critical survey of previous work and discusses problems and issues that arise. Section 6.2 discusses limitations and applications of experimental research in the context of algorithm analysis.

3. *To develop principles for successful experimental research in the domain of algorithm analysis.* Section 6.3 presents four general principles for successful experimental studies.

4. *To promote more general use of this approach by giving a "handbook" of useful tools and techniques.* The handbook appears in Chapter 7. Topics include accuracy and reliability of simulation results, variance reduction techniques, choice of sample points, and analytical tools appropriate for this domain. Section 7.9 summarizes the handbook by giving a list of rules-of-thumb for simulating algorithms. Also, the case studies provide a portfolio of examples: results were purposely presented in an evolutionary style so that the investigative nature of the research could be seen.

8.2. Future Work

More questions and open problems have been raised by this research than have been answered. Many conjectures and observations from the case studies await further analysis and experimentation. Also many issues of experimental procedure deserve further study. This section presents some of the more prominent open problems from the case studies and suggests future directions for the study of experimental techniques in this domain.

Bin Packing

There is at present no theoretical characterization of First Fit packings for $u < 1$. In particular, characterization of the nonmonotonicity in u remains an intriguing open problem. Do the local minimum and maximum move with n? What is the value of u that gives maximum empty space asymptotically?

A first step would be to formalize the argument for linear empty space when $u = 0.8$ (given in Section 2.3.1.) by proving that expected empty space in 2-item bins is bounded below by some small constant. The next step would be to extend the argument (in terms of k-item bins and empty space in k-item bins) to all values of u. The proportion of k-item bins for any u is suggested by Graph 2-6-a; further experiments would reveal appropriate values for the small constants. Limited experimental results give the weak conjecture that empty space does not grow linearly at small values of u; is there an abrupt change the asymptotic function at $u = 0.5$, as is the case with First Fit Decreasing?

Although the First Fit Decreasing algorithm has been theoretically characterized as a function of n for fixed values of u, a function in terms of n and u are not known. For example, when $u \leq 0.5$, empty space has been proven to be constant with respect to n. Experiments suggest that (for partial empty space) the constant depends upon u and grows approximately as $u^{1/3}$. A similar open problem exists for u between 0.5 and the critical region: here, empty space appears to grow linearly in u. The cyclic component observed as a function of u also remains unexplained.

Experimental results suggest that the Best Fit and Best Fit Decreasing algorithms produce packings with structure very similar to those for First Fit and First Fit Decreasing, respectively. Theoretical characterization of the former two algorithms seems to be a very difficult task. The only expected case result to date is Shor's [10] analysis of Best Fit packings when $u = 1$.

In addition to expected behavior in terms of n and u, variation from the mean is also of interest. The causes of very bad packings in the critical region are only partially understood. A promising experimental approach might be to stratify a variety of input properties to obtain a better view of the relationship between input properties and bad packings.

For all four packing algorithms, the next set of experiments should examine packing structure more closely. The most obvious characteristic to study is the interaction between number of k-item bins and empty space in k-item bins. A better view of this behavior could

lead to characterization of First Fit at all values of u, and, for First Fit Decreasing, to understanding of the cyclic component and of packings in the critical region.

An obvious direction for further experimental study is to examine a variety of input distributions and packing algorithms. Coffman, Garey, and Johnson [5] give a thorough survey of bin packing and related problems.

Greedy Matching

The primary open problem from the Greedy Matching study is to characterize the distribution of points after k levels of nearest-neighbor removal. Limited experimental study of the distribution of inter-point distances suggests that the shape is generally invariant over k, although the spread increases as points are eliminated.

An easier task might be to obtain a lower bound on the expected number of nearest-neighbor pairs removed at each level and an upper bound on the expected cost of edges removed. Bounds on these two quantities, combined with the arguments of Section 3.5, would lead to proofs of the conjectured logarithmic edge cost for Greedy Matching and linear expected running time of the matching algorithm.

Median-Selection in Quicksort

Doug Tygar and I recently proved the conjecture of Section 4.4 that a square-root selection strategy minimizes the total number of comparisons. We also showed that the square-root strategy has subquadratic worst-case performance. The following problems remain open: determining the improvement obtained by the square-root strategy over any fixed-T strategy, finding a closed form for total comparisons, and, for fixed-T strategies, determining the best choice of T as a function of N and M.

Another direction for extending this work is to give a complete analysis of square-root Quicksort: that is, to analyze measures A through F as determined by a specific implementation. The biggest difficulty may be the analysis of the median-selection algorithm. Hoare's algorithm was used in the experimental study because it has an exact analysis for number of comparisons. A similar result for the number of exchanges appears not to have been published. A selection algorithm by Floyd and Rivest [6, 7] gives fewer comparisons asymptotically but has not been analyzed exactly.

A number of implementation issues remain open for variable-sample Quicksort algorithms.

One interesting problem is how best to imbed a general median-selection algorithm into Quicksort: since median-selection algorithms partition their input, it might be profitable to avoid re-considering the sample during the partition stage. Also, an advantage of fixed-T strategies is that the median-selection code may be finely tuned, giving fewer comparisons for a specific sample size than a general algorithm would. Since Quicksort seems to be fairly robust with respect to small changes in sample size, perhaps some hybrid scheme, which contains a small set of finely-tuned selection subroutines and makes an intelligent choice of which to use, would be more efficient in practice than a straightforward square-root strategy[1].

An obvious open problem, ripe for experimental study, is to determine if an implementation of square-root Quicksort exists that is more efficient than standard implementations under realistic conditions.

Sequential Search

The conjecture that for any two Move-Ahead-k rules with different index, one will converge more quickly and the other will have better asymptotic cost, remains open. Experimental results support this conjecture for the family of distributions related to Zipf's law.

Standard theoretical analysis of the search rules has been based upon the asymptotic probability for each permutation of the search list. One reason for the difficulty of studying these probabilities by experimental methods is that the space of permutations is large. Future experiments may be designed to reduce this problem by grouping the permutations in related classes and by examining the distributions of the groups. For example, permutations might be grouped according to ranked costs, such as is displayed in Exhibit 5-8, or perhaps by location of the most commonly-requested item. An appropriate grouping might suggest an analytical shortcut for characterizing the Move-Ahead-k rules.

To simulate the asymptotic performance of the Move-to-Front rule, it is only necessary to generate requests until each has appeared at least once; Bitner [8] showed that the probabilities for search list permutations at this point are equivalent to their asymptotic probabilities. Although this very fast shortcut algorithm was not used in the case study (because of the paired experiments), it could be useful in future studies. Perhaps a shortcut to asymptotic behavior can be found for general Move-Ahead-k rules. For example, starting with the optimum search list order instead of random order may permit faster convergence to

---

[1] I thank Mike Langston for suggesting this hybrid scheme.

steady-state behavior. (Bitner [3] proved that the steady-state probabilities for Transpose are independent of initial list order, and this fact is obvious for Move-to-Front; a similar proof for the other rules would be required to justify the use of this shortcut.) Of course, it would be erroneous to draw conclusions about convergence properties for the standard analytical model from these simulations.

Simulation and Analysis of Algorithms

A common problem in experimental analysis of algorithms arises in the study of heuristics for NP-hard problems. Analytical results are often expressed as bounds on the ratio of heuristic performance to the optimal solution. Unfortunately, it is rarely possible to determine the optimal solution experimentally. In the Bin Packing study, a tight lower bound on the optimal solution was available; for what other NP-hard problems do such tight approximations exist? A promising alternative approach (discussed in Section 1.2) is to generate inputs with known optimal solutions: for what problems is this approach possible? Do the generation schemes preserve interesting input properties?

The idea that an algorithm is to be simulated, rather than implemented, can be exploited to produce very efficient simulation environments. Variance reduction techniques and shortcut algorithms, discussed in Sections 7.3 and 7.6, deserve more extensive application and study. Perhaps the idea of finding a shortcut to asymptotic behavior, discussed above in the context of Search rules, can also be applied to problems in other algorithmic domains.

Creative techniques for obtaining good views of algorithmic behavior also deserve further attention. In particular, tools of *algorithm animation* – producing movies of algorithms in action – can be quite powerful for giving insight into underlying processes. Animations of First Fit Decreasing packings were directly responsible for the proof of constant empty space for $u \leq 0.5$ (appearing in [1]). Animations and "snapshots" of an algorithm can also provide an excellent medium for conveying experimental results. Systems for algorithm animation have been developed by Brown and Sedgewick [4] and Bentley and Kernighan [2].

While a start was made at identifying properties of algorithm analysis that influence the choice of analysis tools, many questions remain open. This thesis only considered statistical and analytical tools found in advanced textbooks: perhaps newer techniques may be applied with success. In particular, the special tools for analysis of time-series data would probably be useful in the Sequential Search and the Greedy Matching studies, as well as in other analytical domains. What analysis tools give rigorous upper or lower bounds on function growth rather than approximate fits?

An interesting problem that was only slightly addressed in this research is the design of an environment to support simulation studies of algorithms. Although numerous simulation languages, random number generators, and statistical packages are available, the emphasis in such systems appears to be somewhat at odds with that of algorithm analysis. What features should be built into a statistical package to support experimental research in this domain? Experience with the case studies suggests that emphasis on graphical tools and exploratory data analysis is desirable, but many more than four case studies are required before a final determination may be made.

Much more experience is required if a rigorous experimental method for algorithmic problems, comparable to that for traditional experimental domains, is to be established. I hope that computer scientists will apply the tools of Chapter 7 (as well as others not discussed there) to their analysis problems and report upon their success. Statisticians can be of great help in identifying statistical and analytical tools that are particularly useful in this domain.

References

[1] J. L. Bentley, D. S. Johnson, F. T. Leighton, C. C. McGeoch, L. A. McGeoch.
Some unexpected expected-behavior results for bin packing.
In *Proceedings, 16th Symposium on Theory of Computation*. ACM, April, 1984.

[2] J. L. Bentley and B. W. Kernighan.
GRAP: a language for typesetting graphs.
Communications of the ACM 29(8):782-792, August, 1986.

[3] J. R. Bitner.
Heuristics that dynamically organize data structures.
SIAM Journal of Computing 8(1):82-110, February, 1979.

[4] M. H. Brown and R. Sedgewick.
Techniques for algorithm animation.
IEEE Software 2(1):28-39, January, 1985.

[5] E. G. Coffman, Jr, M. R. Garey, D. S. Johnson.
Approximation Algorithms for Bin-Packing – An Updated Survey.
Available from the authors at Bell Laboratories, Murray Hill, NJ 07974.

[6] R. W. Floyd and R. L. Rivest.
The algorithm SELECT – for finding the ith smallest of n elements (Algorithm 489).
Communications of the ACM 18(3):173, March, 1975.

[7] R. W. Floyd and R. L. Rivest.
Expected time bounds for selection.
Communications of the ACM 18(3):165-172, March, 1975.

[8] D. E. Knuth.
The Art of Computer Programming: Volume 2, Seminumerical Algorithms.
Addison-Wesley Publishing Company, Reading, MA, 1973.

[9] R. Rivest.
On self-organizing sequential search heuristics.
Communications of the ACM 19(2):63-67, February, 1976.

[10] P. W. Shor.
The average-case analysis of some on-line algorithms for bin packing.
In *Proceedings, 25th Symposium on Foundations of Computer Science*, pages 193-200. IEEE, October, 1984.