Experimental Analysis of Algorithms

Catherine C. McGeoch

As far as the laws of mathematics refer to reality, they are not certain; and as far as they are certain, they do not refer to reality. —*Albert Einstein*

In theory there is no difference between theory and practice. In practice there is. —*Yogi Berra*

The algorithm and the program. The abstract and the physical. If you want to understand the fundamental and universal properties of a computational process, study the abstract algorithm and prove a theorem about it. If you want to know how the process *really* works, implement the algorithm as a program and measure the running time (or another quantity of interest).

This distinction between the abstract model and the physical artifact exists in the study of computational processes just as in every other area of mathematical modeling. But algorithmic problems have some unusual features. For example, we usually build models to serve as handy representations of natural phenomena that cannot be observed or manipulated directly. But programs and computers are completely accessible to the researcher and are far more manipulable than, say, weather patterns. They are also much easier to understand: hypothetically, one could obtain complete information about the behavior of a program by consulting technical documents and code. And finally, algorithms are usually invented *before* programs are implemented, not the other way around.

This article surveys problems and opportunities that lie in the interface between theory and practice in a relatively new research area that has been called *experimental algorithmics*, *experimental analysis of algorithms*, or *algorithm engineering.* Much work in this field is directed at finding and evaluating state-of-the-art implementations for given applications. Another effort focuses on using experiments to extend and improve the kinds of results obtained in traditional algorithm analysis. That is, rather than having a goal of *measuring programs*, we develop experiments in order to better *understand algorithms*, which are abstract mathematical objects. In this article we concentrate on examples from this latter type of research in experimental algorithmics.

It is natural to wonder whether such an effort is likely to bear fruit. If the ultimate goal of algorithm analysis is to produce better programs, wouldn't we be better off studying programs in their natural habitats (computers) rather than performing experiments on their abstractions? Conversely, if the goal is to advance knowledge in an area of mathematical research (algorithm analysis), are we wise to study abstract objects using imperfect, finite, and ever-changing measurement tools?

One answer to the first question is that reliable program measurement is not as easy as it sounds. Running times, for example, depend on complex interactions between a variety of products that comprise the *programming environment*, including the computer chip (perhaps Intel Inside), memory sizes and configurations, the operating system (such as Windows 98 or Unix), the programming language (maybe Java or C), and the brand of compiler (like CodeWarrior).[1] These products are sophisticated, varied in design, and, especially when

Catherine C. McGeoch is an associate professor of computer science at Amherst College. She has been active in the development of an experimental tradition in algorithmic studies. Her e-mail address is `ccm@cs.amherst.edu`.

[1] *Intel Inside™ is a registered trademark of the Intel Corporation. Windows™ 98 is a registered trademark of Microsoft Corporation. Unix™ is a registered trademark of The Open Group. Java™ is a registered trademark of Sun Microsystems, Inc. C is not a trademark. CodeWarrior™ is a trademark of Metrowerks, Inc.*

```
1. Array $A[1 \ldots n]$ contains $n$ distinct numbers. We want the $r^{\text{th}}$ smallest. Set $lo \leftarrow 1$ and $hi \leftarrow n$.
2. **Partition.** Set $x \leftarrow A[lo]$. Rearrange the numbers in $A[lo \ldots hi]$ into three groups around some index $p$ such that:
   a. For $lo \le i < p$, we have $A[i] < x$.
   b. $A[p] = x$.
   c. For $p < j \le hi$, we have $A[j] > x$.
3. **Check Indices.** If $p = r$, stop and report the element $A[p]$. Otherwise, if $p < r$, set $lo \leftarrow p + 1$ and go to Step 2.
   Otherwise, $p > r$, so set $hi \leftarrow p - 1$ and go to Step 2.
```

Figure 1: Selection Algorithm S. The algorithm reports the r^{th}-smallest number from array $A[1 \ldots n]$.

used in combination, extremely difficult to model. There is no known general method for making accurate predictions of performance in one programming environment based on observations of running times in another. Experimental analysis of the abstraction allows us to have more control over the trade-off between generality and accuracy when making predictions about performance.

An answer to the second question is straight from the mathematician: algorithms and their analyses are beautiful and fundamental, and they deserve study by any means available, including experimentation.

Certainly algorithms existed long before the first computing machine was a gleam in Charles Babbage's eye. For example, Euclid's *Elements*, circa 300 B.C., contains an algorithm for finding the greatest common divisor of two numbers. While many algorithms have been discovered and published over the centuries, it is only with the advent of computers that the notion of analyzing algorithmic efficiency has been formalized.

Perhaps the first real surprise in algorithm analysis occurred with Strassen's discovery in 1968 of a new method for multiplying two $n \times n$ matrices. While the classic algorithm we all learned in high school requires $2n^3 - n^2$ scalar arithmetic operations, Strassen's algorithm uses fewer than $7n^{\log_2 7} - 6n^2$ operations, where $\log_2 7 < 2.808$. Therefore, this new algorithm uses fewer operations than the classic method when n is greater than 654. Strassen's discovery touched off an intensive search for asymptotically better matrix multiplication algorithms. The current champion requires no more than $cn^{2.376}$ scalar operations for a known constant c. It is an open question whether better algorithms exist. (See any algorithms textbook, such as [1], for more about matrix multiplication.) These fancy algorithms are not much use in practice, however. It would be a daunting prospect even to write an error-free program for one of them, and the extra computation costs imposed by their complexity make them unlikely to outperform the classic method in typical scenarios.

Algorithm analysis is a vigorous and vital subdiscipline of computer science, providing deep new insights into the fundamental power and limitations of computation, fodder for the development of new mathematical techniques (mostly combinatorial), and, not infrequently, efficient algorithms that can have substantial impact on practice. It is clear, however, that our analytical techniques are far too weak to answer all our questions about algorithms. Computational experiments are being used to suggest new directions for research, to support or disprove conjectures, and to guide the development of new analytical tools and proof strategies.

This article presents examples from two broad research efforts in experimental algorithmics: first, to develop accurate models of computation that allow closer predictions of performance, and second, to extend abstract analyses beyond traditional questions and assumptions. We start with a short tutorial on the notations and typical results obtained in algorithm analysis.

A Tutorial on Algorithm Analysis

The *selection problem* is to report the r^{th}-smallest number in a collection of n numbers. For example, $r = 1$ refers to the minimum of the collection, and $r = (n+1)/2$ is the median when n is odd. For convenience we assume that no duplicate numbers appear in the collection.

Figure 1 presents a well-known selection algorithm. The n numbers are placed in an *array* called A in no particular order. Each number has some position from 1 to n in the array: the notation $A[i]$ refers to the number in position i, and i is called an *index*. The notation $\ell \leftarrow lo$ means "set ℓ equal to the value of lo".

The main operation of algorithm S is to choose a *partition element* x from a contiguous subarray of A defined by indices lo and hi and to *partition* the subarray by rearranging its contents so that numbers smaller than x are to its left and numbers larger are to its right. This puts x at some location p; that is, $A[p] = x$. After partitioning, we know that x is the p^{th}-smallest number in A. We are looking for the r^{th}-smallest number: depending on the relationship of p to r, the algorithm either stops or repeats the process on one side or the other of p.

To analyze the algorithm, we derive a function that relates input size to the cost of the computational resources used by the algorithm. The precise meanings of "input size" and "cost" depend upon the *model of computation* being assumed. Here we shall use the simple RAM (random access machine) model under which all scalar numbers have unit size and all basic operations on scalar values—such as arithmetic, comparison, and copying of values—have unit cost. Therefore the input size is n.

For our purposes it is sufficient to define cost as the number of times x must be compared to elements from $A[lo \dots hi]$ during the partitioning step. A partitioning method is known (described later in this article) that uses $hi - lo$ comparisons to partition subarray $A[lo \dots hi]$ of size $hi - lo + 1$. The only problem remaining is to count up total costs.

Obviously, the total cost depends on which partition element $x = A[lo]$ is used each time: we may get lucky and find $p = r$ after just one partitioning operation, or we may have to repeat the process several times. The *worst-case* cost, $C_w(n)$, is the maximal number of comparisons over all arrays of size n and all r: a worst-case scenario holds, for example, when $r = n$ and p becomes $1, 2, \dots, n$ in successive partitioning stages. Letting t denote the cost of a single comparison of x to an array element, we have $C_w(n) = \sum_{i=1}^{n} t(n-i)$.

The *average-case* cost, $C_a(n,r)$, is found by averaging over some probability distribution on arrays of size n. Assume here that every number in the collection is equally likely to be in position lo and selected as the partition element x. Then we have:

$$\begin{aligned} C_a(1,1) &= 0 \\ C_a(n,r) &= t(n-1) \\ &\quad + \frac{1}{n}\left(\sum_{p=1}^{r-1} C_a(n-p, r-p) + \right. \\ &\qquad\qquad \left. \sum_{p=r+1}^{n} C_a(n-p, r) \right). \end{aligned}$$

Note that establishing the correctness of this recursive formula requires a proof (which exists) that the partitioning operation preserves the property that each number in the subset is equally likely to end up in position $A[lo]$ and be chosen as the partition element at a later stage.

Note also that the cost of this, and any, algorithm is described by a *collection* of cost functions that correspond to different scenarios. Another cost function might be obtained with other probabilistic assumptions, or a different definition of cost might be used under some other model of computation.

The first goal in algorithm analysis is the classification of cost functions according to *complexity classes* as defined below. Throughout, we assume that $f(n)$ and $g(n)$ map nonnegative integers to positive real numbers.

Definition: *$f(n) \in O(g(n))$ if there exist positive constants c and n_0 such that $f(n) \le c \cdot g(n)$, $\forall n \ge n_0$.*

Definition: *$f(n) \in \Omega(g(n))$ iff $g(n) \in O(f(n))$.*

The statements below are typical of the kinds of results obtained in classic algorithm analysis.

- Any correct selection algorithm must at least examine every element of A. Therefore, any algorithm that solves the selection problem must have a *worst-case* cost function in $\Omega(n)$. This is called a *lower bound* on the problem of selection.
- It is easy to see that $C_w(n) \in O(n^2)$. We have a *complexity gap* between the $O(n^2)$ worst-case bound and the $\Omega(n)$ lower bound on the problem. Complexity gaps generate research questions: Does an $O(n)$ algorithm exist? Or should the lower bound be raised to, say, $\Omega(n^2)$, because selection is fundamentally harder than the first lower bound indicates? Or is the lower bound somewhere in between, perhaps at $\Omega(n \log n)$?
- In fact, a variation on algorithm S is known that has $O(n)$ worst-case cost, thus closing the complexity gap. The variation uses an elaborate strategy for choosing the partition element at each stage and is generally considered too complicated and expensive to be useful in practice.

The goal of the *asymptotic* analyses above is to place cost functions into complexity classes, while the goal of *exact* analysis is to find closed forms that retain constant factors in the leading terms. A closed form for C_w is easy to obtain. Also, it can be shown that:

$$\begin{aligned} C_a(n,r) = t\,[2\,[(n+1)H_n &- (n+2-r)H_{n+1-r} \\ &-(r+1)H_r + n + 5/3] \\ - \delta_{rn}/3 - \delta_{r1}/3 &- 2\delta_{rn}\delta_{r1}/3], \end{aligned}$$

where H_n is the nth harmonic number and δ_{rn} is the Kronecker delta function. This implies that $C_a(n,r) \in O(n)$ (see Knuth [5], Exercise 5.2.2.31, for details).

Memory Sensitive Models of Computation

In the good old days (say around 1970) it was possible to obtain quite accurate predictions of program running time by inserting appropriate time units for the constants (like t above) in an exact analysis using the RAM model of computation. Nowadays, however, the RAM model is inadequate to the task.

For example, consider the seemingly innocuous instruction $x \leftarrow A[lo]$ in Step 2 of our example algorithm. The actual work to carry out this instruction is performed in a computer by the Central Processing Unit (CPU). The instruction involves about three CPU operations: (1) calculate the memory address of $A[lo]$, (2) fetch the value from that memory address, and then (3) store the value at the memory address associated with x. (Normally, array addresses need to be calculated, but scalar addresses do not.)

A fast modern CPU is capable of performing each operation in about a nanosecond. But the CPU may be forced to wait upon a slow memory, and the additional time needed for the fetch and store operations can vary by enormous factors, depending on exactly where $A[lo]$ and x reside in the *memory hierarchy*. Figure 2 gives a simplified diagram of a memory hierarchy ranging from a small, fast, and expensive *register set*, to a huge, slow, and inexpensive *secondary memory*.

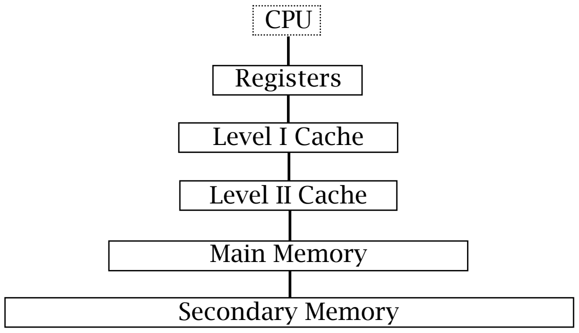

Figure 2: The memory hierarchy, simplified. This drawing is not to scale.

The idea is to keep values that the CPU needs close to it in the fastest memory, but of course not everything will fit. Therefore, memories are designed to retain values that will be needed soon and to evict unneeded values by moving them back to slower memory. It is not always possible to predict which values will be needed next, so most memory levels operate under some kind of *principle of locality*, which says that values that have been recently accessed, or that are near values recently accessed, are likely to be needed soon. There may be a different version of this principle at each level of the hierarchy, and wide variations exist in computing environments with respect to number of levels, sizes of memories, and the eviction policies adopted.

Consider now the cost of fetching the value $A[lo]$. If that value has been used recently, it may be in a register, and the fetching cost is negligible. Decisions about which elements get to stay in registers are primarily made by the *compiler,* a translation program that converts the original program to machine-readable language. It is an NP-hard problem (about which more later) to make optimal decisions about placing variables in registers; therefore efficient optimal algorithms are not likely to be found. Modern compilers incorporate heuristic strategies for deciding which values are to be stored in registers.

If $A[lo]$ is not in a register, it may be in the Level I or Level II cache. Typical cache access times on workstations are around 5 to 10 nanoseconds. Cache sizes and eviction policies are part of the overall design of the computer. If not in a cache, $A[lo]$ is likely to be in main memory, with access times around 50 to 100 nanoseconds. But it is possible that $A[lo]$ has been placed in secondary memory, which may have access times greater than 10^6 nanoseconds. Main-to-secondary eviction policies are incorporated into the operating system.

Whereas the abstract RAM model assigns unit cost to one access of $A[lo]$, true memory access times can vary by factors as large as a million. Without a decent model of the memory hierarchy, we cannot predict whether a given program will take one minute or two years to run. (Of course, this worst-case scenario is not typical, but predictions about program running times that are off by three or more orders of magnitude are not unusual.)

Several threads of research in experimental algorithmics are concerned with developing new models of computation that incorporate aspects of the memory hierarchy, especially with respect to caches and to policies at the main-to-secondary memory levels.

In 1996, for example, LaMarca and Ladner introduced a RAM model with a two-layer memory (cache and main memory). They and several other researchers have since been able to reanalyze classic algorithms and data structures under the new model. These results, obtained through a combination of analytical and experimental approaches, have produced new theorems and new analysis techniques, better predictions of program performance, faster programs, and clear indications that our “common sense” understanding of what contributes to program efficiency is due for revision.

To get a feeling for the type of mathematics involved, consider reanalyzing our selection algorithm under the new two-layer model. We need to be more explicit about the partition step: Figure 3 shows one well-known partitioning method that performs $hi - lo$ array-element comparisons for subarray $A[lo \dots hi]$ of size $hi - lo + 1$.

The new model organizes main memory and the cache into contiguous blocks of scalar values. Main memory contains M blocks named m_0 through m_{M-1}, and the cache holds C blocks named c_0 to c_{C-1}, with $C << M$. Suppose an operation accesses element i which resides in block m_x. Then i and *the entire block containing it* are transferred to cache block c_y, where $y \equiv x \bmod C$. This transfer incurs a cost of t_{out}, but any subsequent access of an element in block c_y will have smaller cost t_{in}. This block will be evicted from the cache and sent back to main memory when another element is accessed from some block m_z such that $y \equiv z \mod C$.

```
Partition.
2.1. Set $x \leftarrow A[lo]$. Set $\ell \leftarrow lo + 1$ and $h \leftarrow hi$. Iterate the following three steps.
    1. Use $\ell$ to scan up from the left, comparing each $A[\ell]$ to $x$. Stop when either $A[\ell] > x$ or $\ell = h$.
    2. Use $h$ to scan down from the right, comparing each number $A[h]$ to $x$. Stop when $A[h] < x$ or $h = \ell$.
    3. If $\ell < h$, swap (exchange) the numbers $A[\ell]$ and $A[h]$ and go to Partition Step 2.1. Otherwise, if $\ell = h$, go to
    Step 2.2 below.
2.2. The above process maintains the invariant that all numbers in $A[lo + 1 \ldots \ell - 1]$ are less than $x$ and numbers in
    $A[h + 1 \ldots hi]$ are greater than $x$. At this point we also have $\ell = h$. If $A[\ell] < x$, swap $A[lo]$ and $A[\ell]$ and set $p \leftarrow \ell$;
    otherwise, swap $A[lo]$ and $A[\ell - 1]$ and set $p \leftarrow \ell - 1$. Partitioning is now complete.
```

Figure 3. Partitioning $A[lo \ldots hi]$ around $x = A[lo]$.

Now suppose subarray $A[i \ldots j]$ resides in memory block m_x. The access cost for a particular array element $A[k]$, with $i \leq k \leq j$, is therefore

- t_{out} if this is the first access of $A[k]$ or any of its block neighbors $A[i \ldots j]$.
- t_{out} if an array element from any block y such that $y \equiv x \bmod C$ has been accessed more recently than any element from block x (causing an eviction).
- t_{in} otherwise.

Homework Problem. Write new formulas for $C_w(n)$ and $C_a(n, r)$ under this model. Perform an exact analysis on both formulas. Would an alternative partitioning method, say one that traverses array elements from left-to-right rather than both-ends-toward-the-middle, give a lower memory access cost?

Clearly even this simple two-level model greatly complicates the analysis task. But the tighter analysis does appear to be well worth the effort in many cases. Jon Bentley (personal communication) reports that simple program modifications can reduce cache effects and improve running times by factors as large as 16. For a nice introduction to memory-sensitive models of computation, see LaMarca and Ladner's [6] analysis of four sorting algorithms under the cacheing model. They draw conclusions about efficiency that flatly contradict a common lore based on traditional analyses and obtain much tighter predictions of program running times than had been possible before.

Another research effort in memory-sensitive analysis concerns algorithms for data sets that must reside in secondary memory because they are too large to fit in main memory. Important applications for such algorithms include Web search engines that peruse directories containing hundreds of millions of entries, many-body simulation algorithms for research problems in the natural sciences, and algorithms for image processing and scientific visualization. The RAM model can also be extended to models that account for data transfers between main and secondary memory (which follow different rules from caches). New algorithms and analyses of old algorithms under these new models can result in huge improvements to program running times. Reductions of running times from several weeks to a few hours have been reported in the literature.

Heuristics for NP-Hard Problems

We now turn to another research thread in experimental algorithmics. The most interesting question about any solvable computational problem is whether it is *tractable* or *intractable*. A problem is tractable if it can be solved in polynomial time; that is, there exists an algorithm for the problem that has worst-case cost in $O(n^k)$ for some constant k. A problem is intractable if no such algorithm exists. For convenience we define intractable problems to be those having exponential lower bounds in $\Omega(c^n)$ for some constant $c > 1$ and ignore the fact that there exist functions like $n^{\log n}$ that are neither polynomial nor exponential.

Perhaps the most important technical idea to arise from the study of algorithms and problem complexity is the identification of problems that are *NP-complete* and *NP-hard*. Hundreds of practical problems that arise naturally in all spheres of industry, commerce, science, and government work have been identified as NP-complete and/or NP-hard. The technical definitions of these two problem classes are rather too involved to fit into this article, but the point is this: it is an open question whether these problems are tractable or intractable. That is, for every problem in these classes there is a huge complexity gap between an exponential worst-case bound on the best algorithm known, and a polynomial lower bound on the problem complexity. Do efficient algorithms exist for these problems, or are they fundamentally too hard to be solved in polynomial time?

For technical reasons, "NP-complete" refers to *decision problems*, which are always phrased as yes-or-no questions (Does input instance I have property P?). "NP-hard" can refer to more general types of problems, such as *optimization problems* that involve minimizing or maximizing some quantity associated with the desired solution. The two classes are related as follows: Let X and Y be NP-complete problems, and let Z be NP-hard but not a decision problem. Then (1) a polynomial time algorithm for X exists if and only if one exists for Y; (2) if a polynomial time algorithm to produce optimal solutions for Z exists, then polynomial algorithms exist for X and Y.

Thus, discovery of a polynomial-time algorithm for any NP-complete or NP-hard problem would have profound implications about the complexity of several hundred other problems of great practical interest, besides earning the discoverer a cool million dollars. (To learn more about the million-dollar Millennium Prize Problems announced by the Clay Mathematics Institute, visit www.claymath.org/prize_problems/.) Proof of an exponential lower bound for any NP-complete problem would reverberate similarly. To learn more about the theory of NP-completeness, see the classic text by Garey and Johnson [3]. Three famous NP-hard problems are described below.

Traveling Salesperson. You are given a graph or digraph $G = (V, E)$ with positive-valued weights on its edges. A *Hamiltonian tour* of G is a closed path over vertices and edges that visits each vertex of G exactly once. The *cost* of the tour is the sum of the weights of edges traversed on the path. The problem is to find a minimum-cost Hamiltonian tour of any given graph G.

This problem is of interest to traveling salespersons who wish to tour all cities in a sales region while minimizing total travel cost. It is also of interest to any organization that must schedule regular tours of delivery trucks, like the U.S. Post Office and your favorite grocery store chain.

Bin Packing. Given a list of n *weights*, all from the real number range $(0, 1)$, the problem is to organize the weights into unit-capacity bins so as to minimize the number of bins used. For example, the weight list containing $0.4, 0.3, 0.6, 0.5$ could be grouped into $(0.4, 0.3), (0.6), (0.5)$, which uses three bins, or into $(0.4, 0.6), (0.3, 0.5)$, which uses only two bins.

This problem is of interest to anyone walking into a lumberyard with a list of board lengths needed for a construction project who must group the lengths so as to minimize the number of standard 8-foot boards that must be purchased. It is of interest to any construction supply firm that must cut stock from unit-sized pieces. The problem is also of interest to anyone who wants to back up files by copying onto unit-sized floppy disks while minimizing the total number of disks needed. A two-dimensional version of the problem (where weights come in (x, y) pairs) is of interest to any printer or pattern cutter who must lay out parts on unit-sized rectangular sheets so as to minimize the total number of sheets needed.

Graph-Coloring. Given a graph G, a *coloring* of G is an assignment of colors to vertices under the constraint that no two vertices sharing an edge can have the same color. The graph-coloring problem is, for any given graph G, to find a coloring that minimizes the total number of distinct colors used. Graph-coloring is a generalization of the famous map-coloring problem to nonplanar graphs.

Graph-coloring is of interest to a university registrar who assigns times (colors) to courses (nodes), under the constraint that no two courses with the same professor can meet at the same time. It is of interest to anyone who needs to create work schedules or timetables that avoid certain kinds of conflicts. And graph-coloring arises in the context of the compiler task, mentioned earlier in this article, of assigning values to registers so as to minimize memory access costs.

Recall that the CPU can work only on values that are in registers. Memory costs are incurred when a value must be transferred from another memory to a register (here we assume that this transfer cost is constant).

Consider the sequence of instructions that occurs at the beginning of our selection algorithm S:

$$\begin{array}{lll} (1) & lo & \leftarrow & 1 \\ (2) & hi & \leftarrow & n \\ (3) & x & \leftarrow & A[lo] \\ (4) & \ell & \leftarrow & lo+1 \\ (5) & h & \leftarrow & hi \end{array}$$

Two values *interfere* if one must be accessed between two accesses of the other. When two values interfere, we want to place them in different registers so as to avoid the cost of evicting one to make room for the other. For example, lo and hi interfere because accesses of hi on line (2) come between accesses of lo on lines (1) and (3) and also because lo is accessed between lines (2) and (5). If both lo and hi were assigned to register $R1$, then these values would incur the following memory transfer costs: (1) fetch lo; (2) evict lo, fetch hi; (3) evict hi, fetch lo; (5) evict lo, fetch hi; for a total of seven memory transfers. These two values could coexist peacefully if placed in registers $R1$ and $R2$, although this might cause interference difficulties with other values.

More generally, given a sequence of instructions in a program, we can build an interference graph G where nodes represent program values and an edge is placed between two nodes if their values interfere. The problem is to assign a minimum number of registers (colors) to values (nodes) such that interfering values receive distinct register assignments.

As is the case with all NP-hard problems, no polynomial-time algorithm for finding minimal colorings is known, and it is not known whether such an algorithm exists. While the theoreticians struggle with the general problem, the compiler writers must settle for coloring algorithms that use a "small number" of colors, if not the absolute minimum. Dozens of algorithms have been proposed, which fall into some broad categories, broadly sketched in the list below. (Colors are assumed to be numbered 1 through k, for some appropriate k.)

Greedy Algorithms. A greedy algorithm may iterate over nodes, assigning to each the least-numbered color that does not violate any edge constraint. Or an algorithm may iterate over colors, assigning each color to a large independent set (finding a maximum independent set is an NP-hard problem). Greedy algorithms are often repeated with different node or color orderings each time, and the best solution found is saved. These algorithms are fast but may produce colorings using many more colors than necessary.

Branch-and-Bound. Systematically generate colorings of G, avoiding redundant colorings and unproductive starts, and save the best one found within a given time limit. Note that even on moderate-sized graphs only a tiny fraction of colorings can be generated from the enormous space of possibilities; various strategies for generating the best ones first may be considered.

Heuristic Search Methods. Start with a valid coloring. "Step" to another valid coloring by randomly modifying the current coloring in some small way (perhaps by changing the color of one node). Continue stepping until a good coloring is found. Steps toward better colorings are of course preferred, but it is important occasionally to allow steps toward worse colorings to avoid being trapped in a local minimum. A rich variety of stepping rules and strategies for controlling the stepping process have been proposed.

These general strategies can be modified to fit many varieties of NP-hard problems, including Traveling Salesperson and Bin Packing. In some cases (like these two), strategies that exploit special problem structures are known that outperform the general techniques. Note that the analysis of algorithms for NP-hard problems usually involves finding functions and complexity classes for two interesting quantities: algorithm efficiency and the quality of solution produced by the algorithm.

In the case of heuristic algorithms like those sketched above, average-case analyses are extremely difficult to obtain even under the simplest computational and probabilistic models. Analytical statements about how a coloring algorithm performs on, say, random graphs with edge probability p, are rare, and statements about performance on "graph structures typical of register interference problems" are far beyond our analytical capabilities.

Computational experiments can be used to enrich our understanding of these heuristic algorithms for NP-hard problems. Experiments have been used to direct the discovery of new algorithms and new analyses, to allow prediction of algorithmic responses to variations in input properties, and to sort out which approaches work best in which kinds of scenarios.

Indeed, graph-coloring has been the focus of what might be termed a new mode of research in algorithmic studies: a coordinated, multiparticipant effort to identify the state-of-the-art in algorithmic performance for some given problem. Since 1992 the DIMACS Center (an NSF- and corporate-funded center for research in discrete mathematics and theoretical computer science, located at Rutgers University) has sponsored an annual Implementation Challenge to encourage and promote experimental research on algorithms for specific problems. The focus on a particular problem allows a great deal more coordination and interaction than might otherwise be possible by independent researchers.

The second DIMACS Challenge (1993) [4] concerned algorithms for three NP-hard problems: graph-coloring, finding large cliques in graphs, and finding satisfying truth assignments for Boolean formulas. While a great deal of progress was made, one conclusion drawn at that workshop was that huge gaps remain in our understanding of the "best algorithms" for particular applications. Much more work remains to be done.

Besides the DIMACS Challenges, several other venues for experimental research on algorithms have appeared in the past decade. Two annual conferences (WAE in Europe and ALENEX in North America), the electronic *Journal on Experimental Algorithmics*, and numerous small workshops have all been developed to provide forums for researchers to report on their experimental observations.

Questions of Methodology

Research in experimental algorithmics produces new understanding of models, problems, algorithms, and program efficiency. Another important component of experimental algorithmics has been the development of methods and techniques for performing experiments on algorithms. As suggested in the beginning of this article, the relationship between algorithms and programs and the usual types of questions asked about algorithms are not typical of those that arise in mathematical modeling. Consequently, new problems of designing and conducting experiments and of developing appropriate statistical tools can be identified. Some research questions concerning methodologies for experimental analysis of algorithms are sketched below. See [7] for more discussion.

How to Escape the Artifact? We study a perfect abstract object (the algorithm) using an imperfect measuring device (the computer). Difficulties associated with numerical precision, nonrandomness in pseudo-random number generators, and inaccurate measuring tools can produce spurious and erroneous results. Tools for measuring program running times are notoriously quirky, and in some contexts it can be quite difficult to find a definition of "cost" that is useful, measurable, and generalizable. New techniques are needed for identifying and minimizing the impact of artifact on observation.

Sampling from Large and Infinite Spaces. It is not possible to measure the performance of a given graph-coloring algorithm over the space of all graphs of a given size n, nor is it always feasible to sample from the space of all colorings of a given graph. We need good methods for defining and generating input classes and input samples to support observations that can be generalized *outside* the realm of observation. For example, very little is known about methods for extrapolating from a finite data set to obtain the kind of asymptotic statements about complexity classes (e.g., does the data come from a function that is $O(n^2)$?) that are of interest to computer scientists.

New Data Analysis Techniques. Computational experiments on algorithms can produce enormous data sets. Some sets are too massive to be amenable to standard statistical techniques, so we must either make do with fewer measurements or develop faster techniques. Computational experiments can be very interactive and iterative, and it seems likely that new methods of data analysis can be developed to take advantage of these properties.

How Important Are Algorithms?

As the discussion in this article suggests, sufficiently precise results for all but the simplest algorithms, input models, and models of computation are quite difficult to obtain by purely analytical methods. With our current techniques, asymptotic analyses of algorithms are just barely adequate for making very rough predictions about program running times. Some researchers have wondered if we would be better off discarding our analytical models and building new empirical models based entirely on observations.

In an experiment to test the relevance of big-oh algorithm analysis to practice, Bentley [2] describes a race between two algorithms implemented under extreme conditions. He implemented an $O(n)$ algorithm (under the RAM model) using 1980's low-end technology, which required $19.5n$ milliseconds to run, and an $O(n^3)$ algorithm in high-end 1999 technology, which required $0.58n^3$ nanoseconds to run. At small n the cubic algorithm dominates, and at large n the linear algorithm dominates: Bentley found that even with this maximal difference in computing environments, the crossover point was low, near $n = 5,800$, where both implementations required around 2 minutes of running time. At $n = 10^5$ the $O(n^3)$ implementation with small coefficients required 7 days, while the $O(n)$ implementation with huge coefficients took only 32 minutes.

This and other experiences reported in the literature suggest some rough guidelines about the relative importance of algorithmic and environmental factors: In typical scenarios changes in environment (such as switching computers, compiler optimization level, or operating system; or improving programmer skills) can affect running times by factors in ranges between 2 and 100. Improvements on a larger scale can be obtained through asymptotic improvements to algorithms, by factors of $O(n)$ or more, and algorithmic improvements by factors below, say, $O(\log n)$ are not likely to have much real impact on performance.

Continuing research in the new and rapidly developing field of algorithm experimentation will allow us to refine these guidelines, to produce better understanding of environmental effects, and to generate better algorithms and tighter analyses.

Acknowledgment. I thank the editor, Harold Boas, and an anonymous reviewer for several suggestions for improving the clarity of this article.

References

[1] S. Baase and A. Van Gelder, *Computer Algorithms: Introduction to Design and Analysis, Third Edition*, Addison-Wesley, 2000. A very readable undergraduate textbook.

[2] J. L. Bentley, *Programming Pearls, Second Edition,* ACM Press and Addison-Wesley, 2000. Short articles about the interface between theory and practice in computer programming.

[3] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, 1979. A classic introduction to the field.

[4] D. S. Johnson and M. Trick, eds., *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 26, Amer. Math. Soc., 1996. Papers presented at the workshop.

[5] D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, Addison-Wesley, 1973. Classic results in algorithm analysis.

[6] A. LaMarca and R. E. Ladner, The influence of caches on the performance of sorting, *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms*, ACM, 1997, pp. 370–379. A very readable technical paper.

[7] C. McGeoch, Towards an experimental method for algorithm simulation, *INFORMS J. Comput.* **8** (Winter 1995). A survey of methodological issues, with an extensive bibliography.