On the Relevance of Various Cost Models of Complexity

W. Morven Gentleman

CS-73-14

Department of Applied Analysis and Computer Science

University of Waterloo
Waterloo, Ontario, Canada

ON THE RELEVANCE OF VARIOUS COST MODELS OF COMPLEXITY

W. Morven Gentleman
University of Waterloo

Abstract

While one possible goal of work in low level complexity is to study the intrinsic difficulty of computing certain functions, another more practical goal is to obtain sufficient understanding about the relative merits of complicated algorithms to be able to provide useful advice to someone undertaking an actual computation.

With respect to the latter goal, it is clear that if the conclusions of an analysis are to be applicable to a real computation, then the model being analyzed must meaningfully characterize the actual problems being solved and must accurately reflect the actual costs. Many complexity analyses today do not do this. This paper will discuss some, and suggest what can be done.

Why do Analysis of Algorithms?

It is, of course, a legitimate goal of research in the theory of computation to investigate the intrinsic difficulty of computing certain functions on machines of a given structure, either to classify the functions or the machines. In either case, the specification of the machine is part of the formulation of the problem, and is hardly open to discussion.

However, a quite different goal often motivates work in this area. In actual computation, design decisions and value judgments are continually having to be made: which algorithm should be used? which data representation should be employed? which part of the computation, being the most expensive part, should be most carefully done? how should the program be structured? how much can the program be improved? Rather than just using cut-and-try methods or worse, going on unfounded hunches and prejudices, it would be desirable to have some theoretical assistance with such questions.

What is wanted, then, is theoretical analysis of models that abstract real algorithms and problems. With this goal, it is clear that questions of how well the model analyzed reflects the real costs of an actual computation and how well the problems being solved are characterized are vital.

How well do our current style of analyses measure real costs?

With this in mind, let us ask how well our analyses do in some specific areas: matrix computations, rootfinding, sorting, and symbolic algebra computations.

In matrix computations, the traditional analysis has counted the number of multiplications required (or sometimes the numbers of multiplications and additions). If the process is iterative, the count is done per step, then some factor inversely proportional to the rate of convergence is applied to give the cost to convergence. Does this really indicate the costs?

As an illustration, I took an Algol program for iteratively solving a linear least squares problem, and counted the number of multiplications (6655) and additions (11651) per step. Knowing the floating point add and multiply times on my Honeywell 6050 computer (1.8 and 3.8 microseconds) I could conclude that the arithmetic cost per step was .04626 seconds. The measured time was .4165 seconds, so the overhead of bookkeeping, loop control, array element access, etc. made the computation <u>nine</u> times as expensive as the arithmetic costs alone would indicate. Moreover, if bounds checking is requested on array element accesses, the measured cost increases so this ratio becomes <u>sixty</u> times the cost of the arithmetic alone. And this is on straight line code using only simple loops, compiled by one of the better Algol compilers in the industry.

These results suggest that changes in the non-arithmetic structure of the algorithm might have significant impact, but changes in the amount of arithmetic are unlikely to. This suggestion is borne out in another experiment (on a different machine, an IBM 360/75) in which two different forms of Givens transformations without square roots were being tested. The interest in the second form was that ie by a minor rearrangement of algebra, it reduced the number of multiplications by 1/3. More precisely, the number of multiplications using the first form was $\frac{3}{2}(p+1)p + 6p$ and using the second form was $(p+1)p + 5p$. However the measured costs turned out to be $46p^2 + 716p - 2591$ μseconds and $43p^2 + 596p - 947$ μseconds respectively. While the multiplication count had predicted the form of the cost functions correctly, the relative magnitudes were wrong, leading to the erroneous conclusion that the second method was substantially superior.

The examples above are still relatively straightforward compared to what happens when large matrix computations are performed on a virtual memory machine. It is not difficult to find cases where the conventional algorithms access data in such an unfavourable manner that almost every array element reference requires another page pull, but where reordering the indexing, in a way that does not change the amount of arithmetic, can dramatically change the cost, e.g. Moler [2]. In fact Singleton [3] found that using a version of the fast Fourier transform that was inferior in terms of the number of trigonometric function computations required was vastly superior in terms of real cost when working on the virtual memory Burroughs B5500. (Even using this best available version, an FFT on 2^{15} points was taking almost 50% as much channel time as CPU time).

We might conclude from such cases that multiplication count alone is too simple a model to use to predict what really happens in matrix computation.

What about rootfinding? The traditional analyses have been in terms of order, p, with cost θ being measured by the cost of various function and derivative values needed at each step. For one step iterations with no memory, where p function and derivative values are needed to obtain order p, we can readily make the assumption that all function and derivative values are equally expensive to obtain, and conclude that the efficiency, $p^{1/\theta}$ is minimized at p=3, indicating cubically convergent rules should be used. If derivative values become more complicated to compute as the degree increases, specifically if computing the function and the first s-1 derivatives is more expensive than $s^{1.136}$, second order rules become preferable. Only if derivatives become progressively cheaper to compute as the degree increases do higher order rules pay: if the cost grows less than $s^{.8}$, fourth order rules are preferable, if less than $s^{.5}$, seventh order rules are preferable, and if less than $s^{.333}$, twentieth order rules are preferable.

In fact there is not much in all this. In the first place the efficiencies are not much different: when the cost grows like s, p=3 is only 2% larger than p=2 or p=4. Moreover, two assumptions are buried in it whose violation invalidates the application in practice. First, that costs of function evaluations can be discussed, independent of the abscissa. This can easily be false, particularly when the function evaluation is done to guaranteed relative accuracy (as the author found when computing zeros of Bessel functions). Secondly, that only asymptotic behaviour need be considered: not only can starting and stopping be ignored, but even asymptotic error constants can be neglected. This is, of course, imposed for mathematical convenience, because these features are hard to discuss and depend on quantities not readily available a priori. Nevertheless, in real computations on a finite wordlength machine it is most unusual to ever take more than a few steps, and the asymptotic situation never holds: conclusions based on it are often simply false.

Perhaps the situation is better for some non-numeric computations like sorting. Here the traditional measure is the number of comparisons and it is well known that asymptotically $n \log_2 n$ comparisons are needed, and that this number is achievable. Van Emden's modification of Quicksort [4] achieves an average of $1.140\ n \log_2 n$ comparisons and appears to be fastest published program by measurement. Of course, again overheads dominate the real cost.

Sorting too has its troubles, however, for once sorts become too large to be in core, the real costs are wholly associated with I/O, and only recently [1] have models become sophisticated enough in terms of queuing theory and models of I/O systems to predict the observed behaviour. And sorting on virtual memory machines seems not yet to have been well modeled.

Our final area is symbolic algebra computations

Two very surprising results have been obtained here in the past few years, first, that the application of modular arithmetic homeomorphisms can asymptotically make dramatic reductions in the maximum number of operations required in various computations, and second, that the conclusions of analyses based on models of sparse polynomials, i.e. those with only a few nonzero terms, can be radically different from those based on dense models. The second of these results has had impact immediately: sparse polynomials seem more typical of what is really seen by an algebraic manipulation system, and the improvements observable when using a method previously thought inferior can be striking. The first, however, has had a harder time gaining acceptance--the overhead is high in many data representations and the crossover points are unknown. Moreover, all advantage of sparseness is lost, and without exploiting sparseness, only the smallest problems can actually be solved.

How could our cost models be improved?

The root of the problem, indicated by the shortcomings of the theoretical analyses above, lies in the fact that the entire thrust of modern mathematics is toward abstraction. The algorithms we consider are mathematical transformations, and they can be applied to many classes of operands. For example, Gaussian elmination can be applied in any division ring, and a sorting algorithm can be applied to elements of any ordered set. In this context, we are naturally lead to measuring costs by operations appearing in the algorithm as mathematically formulated.

But whereas the algorithm is unaffected by which of the various isomorphic problems it is applied to, the cost of real computation is not. Indeed, one might almost argue that the abstraction in the algorithm is introduced explicitly to make it transparent to the very features on which real costs depend: program structure, data representation, data management, implementation of primitive operations, special structure in the data for problems of interest, etc.

Another thing which often makes our analyses inappropriate is that mathematical difficulty, combined with the desire to have results that can be expressed simply, often leads to asymptotic analyses. But the problems which are usually solved, indeed sometimes the only ones which can be solved, are often far from the range of validity of the asymptotic conclusions.

The above remarks might suggest that what is needed is a thorough analysis using more elaborate models. I should like to suggest this is not so, for the following reasons:

1. Since no model can reasonably be expected to treat exactly the specific problem that actual computation will be done on, what is wanted is not exact results, but intuition.
2. The more elaborate the model, the more likely it is to be mathematically intractable. While for simple models this may be coped with by finding approximate relationships, for many parameter models the situation may be hopeless. Choosing only models that are mathematically tractable can be quite misleading.
3. Even if results can be obtained, the relationship between a many parameter input and a complicated output can be so difficult to see that extracting intuition from the mess is impossible. In fact, relating a many parameter model of the data to the specific input for the computation in question is often impossible.
4. As the model becomes more elaborate and more detailed, the range of applicability narrows until it may not include the case at hand.
5. Even elaborate models of computations with the mathematical algorithm still omit details about program structure.

A new philosophy for analyzing algorithms

Recognizing that what is wanted is not necessarily precise analyses for specific cases but rather sharpened intuition, how can this be made available?

Since we cannot be sure which cost model will apply, analyzing several is desirable, and since what we really want to uncover are the sensitive parts of the computation, extreme models are often more informative than more general ones. Since program structure and data structure are vital to include in our models, something closer to analysis of programs rather than analysis of algorithms is needed. (Of course, even as several cost models should be tried, several program and data structures should be too). Because we are looking at programs, mechanical help is possible: for well structured programs it should be possible to have a processor that examines the program flow and produces a analysis in terms of parameters such as loop limits and the number of successes and failures of given tests for such cost factors as the number of primitive operations, e.g. arithmetics, array element references, elementary function calls, and so forth. Various models for the cost of these primitive operations can then be tried. (An algebraic manipulation system can be a big help here). In this context, it is useful to note the work of B.A. Wichmann on studying Algol execution speeds [5], in which he finds primitives which do not interact, whose relative speeds are almost independent of the particular machine and implementation, and which reasonably predict measured costs.

Finally, empirical evidence to support the intuition, not just to support the mathematical models, should be a mandatory part of the analysis. It should be checked by measurements on real computations, possibly cases too complicated to analyze directly, that the behaviour predicted really does occur--or more important, that unexpected behaviour does not. For this purpose isolated experiments can be useful, but empirically determined cost formulae are usually far more so.

References:

Kritzinger [1] Kritzinger, P., "An approach to the optimization of direct access merge peformance", unpublished Ph.D. thesis, University of Waterloo, 1972.

Moler [2] Moler, C., "Matrix Computations with Fortran and Paging", CACM, Vol.15, No.5, April 1972, pp. 268-270.

Singleton [3] Singleton, R.C., "On computing the fast Fourier transform", CACM, Vol.10, No. 10, October 1967, pp. 647-654.

Van Emden [4] Van Emden, M.H., "Increasing the efficiency of Quicksort", CACM, Vol. 13, No. 9, Sept. 1970, pp. 563-567.

Wichmann [5] Wichmann, B.A., "A comparison of Algol 60 execution speeds", National Physical Laboratory Report CCU3, 1969, also The Algol Bulletin.