CUCS-8-80

COMPUTATIONAL COMPLEXITY

J.F. Traub
Departments of Computer Science and Mathematics
Columbia University
New York, N.Y. 10027

July 1980

To appear in
ENCYCLOPEDIA OF COMPUTER SCIENCE

This research was supported in part by the National Science Foundation under Grant MCS-7823676.

COMPUTATIONAL COMPLEXITY

J.F. Traub

For articles on related subjects see

ALGORITHMS, ANALYSIS OF

ALGORITHMS, THEORY OF

FAST FOURIER TRANSFORMATION

MATHEMATICAL PROGRAMMING

TURING MACHINE

The subject matter of computational complexity is the determination of the intrinsic difficulty of mathematically posed problems arising in many disciplines. The study of complexity has led to more efficient algorithms than those previously known or suspected. We begin by illustrating some of the important ideas of computational complexity with the example of matrix multiplication.

COMPUTATIONAL COMPLEXITY OF MATRIX MULTIPLICATION. Consider the multiplication of 2 x 2 matrices. Let

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}, \qquad B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}, \qquad C = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}.$$

Given A, B, we seek C = AB.

The classical algorithm computes C by

$$c_{11} = a_{11}\,b_{11} + a_{12}\,b_{21}, \qquad c_{12} = a_{11}\,b_{12} + a_{12}\,b_{22},$$

$$c_{21} = a_{21}\,b_{11} + a_{22}\,b_{21}, \qquad c_{22} = a_{21}\,b_{12} + a_{22}\,b_{22},$$

at a cost of eight multiplications.

Until the late sixties no one seems to have been asked whether two matrices could be multiplied in fewer than eight scalar multiplications. Then Strassen showed that 7 scalar multiplications are sufficient by introducing the following algorithm:

$$p_1 = (a_{11} + a_{12})(b_{11} + b_{22}), \qquad p_2 = (a_{21} + a_{22})\,b_{11},$$

$$p_3 = a_{11}(b_{12} - b_{22}), \qquad p_4 = a_{22}(-b_{11} + b_{21}),$$

$$p_5 = (a_{11} + a_{12})b_{22}, \qquad p_6 = (-a_{11} + a_{21})(b_{11} + b_{12}),$$

$$p_7 = (a_{12} - a_{22})(b_{21} + b_{22}),$$

$$c_{11} = p_1 + p_4 - p_5 + p_7, \qquad c_{12} = p_3 + p_5,$$

$$c_{21} = p_2 + p_4, \qquad p_{22} = p_1 + p_3 - p_2 + p_6.$$

Consider next the multiplication of N x N matrices. The classical algorithm uses N^3 arithmetic operations. (In this article we disregard multiplicative constants in giving

algorithm cost.) By repeated partitioning of N by N matrices into 2 by 2 submatrices, two matrices can be multiplied in $N^{\log_2 7} \sim N^{2.81}$ arithmetic operations.

After a decade during which there was practically no progress on decreasing the number of arithmetic operations used in matrix multiplication, Schönhage and Pan (1979) showed $N^{2.52}$ arithmetic operations are sufficient. This number, 2.52, is the current state of our knowledge and researchers expect the exponent will be further decreased.

We must emphasize that the above results are of theoretical rather than practical value. The value of N has to be enormous before the new algorithm would be faster than the classical one. On the other hand, there are some problems for which new algorithms have had profound influence. A good example is provided by the finite Fourier transform on N points. The fast Fourier transform uses only N log N arithmetic operations compared to N^2 for the classical algorithms. Since N log N is much smaller than N^2 for even moderate values of N and since the finite Fourier transform is often needed for a large number of points, the introduction of the fast Fourier transform has revolutionized computation in a number of scientific fields.

Using the matrix multiplication example we can now introduce some basic terminology.

The minimal number of arithmetic operations is called the computational complexity (or problem complexity) of the matrix multiplication problem. We often write complexity for brevity.

The complexity of matrix multiplication is unknown. An upper bound is $N^{2.52}$. A lower bound is N^2. Since this lower bound is linear in the number of inputs and outpust we say it is a trivial lower bound, No non-trivial lower bound is known.

Algorithm complexity is the cost of a particular algorithm. This should be contrasted with problem complexity which is the minimal cost over all possible algorithms. People who do not work in complexity theory sometimes confuse these two terms.

Fast algorithm is a qualitative term meaning faster than a classical algorithm or faster than previously known algorithms. An optimal algorithm is one whose complexity equals the problem complexity.

Table 1 summarizes the present state of our knowledge concerning matrix multiplication.

SUMMARY OF MATRIX MULTIPLICATION

upper bound	$N^{2.52}$
lower bound	N^2
complexity	unknown
optimal algorithm	unknown

TABLE 1

COMPUTATIONAL COMPLEXITY IN GENERAL. To study computational complexity requires a model of computation stating which "operations" or "steps" are permissible and how much they cost. Using the model we can then ask the same questions as in the matrix multiplication example. For instance we seek:

problem complexity
upper bounds
lower bounds
fast algorithms
optimal algorithms

Typically, an upper bound is the cost of the fastest known algorithm for solving the problem. A lower bound can only be established through a theorem that states there does not exist an algorithm whose cost is less than the lower bound. Not surprisingly, lower bounds are far harder to establish than upper bounds.

Numerous models of computation have been studied. In our matrix multiplication example we counted arithmetic operations. In the study of combinatorial problems we typically count comparisons. Very significant results have been obtained for space and time complexity in a Turing Machine model. Another important model is a random access machine (RAM). Other models are appropriate for studying parallel, asynchronous, or VLSI computation.

Often we assign a "size" N to a problem. If the number of operations or steps required to solve a problem is an exponential function of N we say the problem has exponential time complexity. If the problem requires a number of operations which is a polynomial function of N we say the problem has polynomial time complexity.

TYPICAL APPLICATIONS OF COMPUTATIONAL COMPLEXITY. The complexity of numerous problems has been studied. To illustrate the variety of problems we exhibit a dozen drawn from various areas.

1. Compute the finite Fourier transform at N points.
2. Determine if an N digit integer is prime; if not, determine the factors.
3. Solve an elliptic partial differential equation to within an error ε.
4. Compute the Kendall rank correlation at N points.
5. Generate a function with error less than ε from values of a function at N points.
6. Multiply two polynomials of degree N.
7. Prove all theorems which can be stated in at most N symbols in a certain axiom system.
8. Solve the traveling salesman problem on N cities.
9. Solve to within ε a large sparse linear system of order N whose matrix is positive, definite, and has condition number bounded by M.
10. Find the closest neighbor of P points in K dimensions.
11. Compute the first N terms of the Qth composite of a power series.
12. Compute the first N digits of π (for, say, N = 20, 000,000).

REDUCIBILITY AMONG PROBLEMS. There are many problems for which the best algorithm known costs exponential time. Such problems occur in operations research, computer design, data manipulation, graph theory and mathematical logic. Do there exist faster algorithms which solve these problems in polynomial time? We don't know. What we do know is that there is a large class of problems which are equivalent in that if one of them can be solved in polynomial time, they all can.

For technical reasons this class of problems is said to be NP-complete (q.v.). Because no one has succeeded in devising a polynomial time algorithm for any of these problems, many researchers believe that NP-complete problems are exponentially hard. There is no proof of this and settling this question is one of the most important open problems in computational complexity.

ANALYTIC COMPUTATIONAL COMPLEXITY. For the matrix multiplication problem we are interested in the minimal number of arithmetic operations to multiply two matrices exactly. This is a typical problem of "algebraic complexity". However, many problems can be only approximately solved. Examples are optimal estimation, solution of nonlinear equation, optimization, and the solution of partial differential equations. Indeed, most problems occurring in mathematics, science, engineering, risk assessment, decision theory, and economics can be only approximately solved. Furthermore, to lower the cost we may choose to approximately solve problems which could be exactly solved. Important examples are provided by the approximate solution of NP-complete problems and the iterative solution of large sparse linear problems. Analytic Computational Complexity is the study of optimal algorithms for problems which are solved approximately.

Above we illustrated some of the important ideas of computational complexity with the example of matrix multiplication. Here we will use integration as a simple prototypical example and use it to define basic terminology. Consider the computation of $\int_0^1 f(x)dx$ given the information $[f(x_1), f(x_2),\ldots, f(x_n)]$. This information is partial because there are many integrands which are indistinguishable using this information. If the information has errors (due, for example, to measurement) the information is approximate. It is clear that partial or approximate information causes uncertainty in the integral. This uncertainty is intrinsic and caused by the limited information.

The optimal information is the choice of sample points which minimizes this intrinsic uncertainty.

An algorithm is any procedure for approximating the integral using the information. Any algorithm must have error at least as large as the intrinsic uncertainty. An optimal algorithm is one whose error achieves the intrinsic uncertainty. The computational complexity of the integration problem is the minimal cost of computing the integral to whithin ε.

The information is nonadaptive if the sample points x_i are independently chosen. It is adaptive if we choose x_i only after we know $f(x_1)$, $f(x_2)$, ..., $f(x_{i-1})$. Nonadaptive information is desirable on a distributed computer system since the information can then be computed independently on various processors.

We list some of the questions studied in analytic complexity. Althouth these questions are listed here in the context of integration, the same questions can be asked and have been answered in great generality. For integration, some of the answers depend on the nature of the integrand.

1. What is the minimal number of function samples for which the integral can be computed to within ε.
2. Is adaptive information "better" than nonadaptive information? (Surprisingly, the answer is no for integration and many other problems).
3. What is the optimal information?
4. What is the optimal algorithm?
5. What is the computational complexity of integration?

AXIOMATIC COMPLEXITY THEORY. We discuss an abstract complexity model based on two axioms. Let $T_A(x)$ denote the cost of algorithm A applied to the input of integer x. Assume that $T_A(x)$ satisfies the following two axioms:

1. $T_A(x)$ is finite if and only if algorithm A applied to input x eventually halts and gives an output. (In other words, an algorithm halts if and only if it halts after a finite number of steps.)
2. There is an algorithm which, given as inputs any integers x and y and any algorithm A, will determine whether or not $T_A(x) = y$.

These straightforward axioms are enough to imply, for example, that there are computable functions which cannot be computed rapidly by any algorithm, and that more functions can be computed if more time is allowed. They also imply a much less obvious fact, known as the "Speed-up Theorem": There is a computable function f with the property that given any algorithm A which computes f, there is another algorithm B which computes f "much faster" than A. "Much faster" is interpreted by choosing any rapidly growing computable function such as 2^w; then, according to the speed-up theorem, there is a function f such that if A is any algorithm for f, there is always another algorithm B for f such that $2^{T_B(x)} \leq T_A(x)$ for all large integers x. Thus, algorithm B requires at most the logarithm of the time required by A.

Of course, since B is itself an algorithm for f, there must be another algorithm C for f which requires only the logarithm

of the time for B, and so on. Clearly, there is no single most efficient way to compute such an f.

Also, notice that f must be hopelessly difficult to compute even though it has faster and faster programs. Each program for f must require more than 2^x, and more than 2^{2^x}, and so on, steps for all large inputs x; otherwise, the program could only be "sped-up" by an exponential a fixed number of times before "hitting bottom," after which it could not be sped up further.

These conclusions may seem to violate intuition, but they follow from the two simple axioms given above. The speed-up theorem is proved using diagonal arguments similar to those used to establish the existence of undecidable problems.

CONCLUSIONS. Computational complexity deals with the fundamental issues of determining the intrinsic difficulty of mathematically posed problems. Through the study of complexity it has been established that certain problems are intrinsically hard. On the other hand, for some problem areas new algorithms have been introduced which are far superior to any previously known. Problems occurring in a rich diversity of disciplines are being and will be subjected to complexity analysis.

REFERENCES

1968, 1969, 1973. Knuth, D. The Art of Computer Programming, Vol. I, II, III. Reading, Mass.: Addison-Wesley Publishing Co.

1974. Aho, A.V., J.E. Hopcroft, and J.D. Ullman. The Design and Analysis of Computer Algorithms. Reading, Mass.: Addison-Wesley Publishing Co.

1975. Borodin, A. and I. Munro. The Computational Complexity of Algebraic and Numeric Problems. New York, N.Y.: American Elsevier.

1979. Garey, M.R. and D.S. Johnson. Computers and Intractability. San Francisco, Calif.: W.H. Freeman.

1980. Arden, B. (editor). Chapter on Theory of Computation in What Can be Automated? The Computer Science and Engineering Research Study (COSERS). Cambridge, Mass.: MIT Press.

1980. Traub, J.F. and H. Woźniakowski. A General Theory of Optimal Algorithms. New York, N.Y.: Academic Press.