COMBINATORICS, COMPLEXITY, AND RANDOMNESS

The 1985 Turing Award winner presents his perspective on the development of the field that has come to be called theoretical computer science.

RICHARD M. KARP

This lecture is dedicated to the memory of my father, Abraham Louis Karp.

I am honored and pleased to be the recipient of this year's Turing Award. As satisfying as it is to receive such recognition, I find that my greatest satisfaction as a researcher has stemmed from doing the research itself, and from the friendships I have formed along the way. I would like to roam with you through my 25 years as a researcher in the field of combinatorial algorithms and computational complexity, and tell you about some of the concepts that have seemed important to me, and about some of the people who have inspired and influenced me.

BEGINNINGS

My entry into the computer field was rather accidental. Having graduated from Harvard College in 1955 with a degree in mathematics, I was confronted with a decision as to what to do next. Working for a living had little appeal, so graduate school was the obvious choice. One possibility was to pursue a career in mathematics, but the field was then in the heyday of its emphasis on abstraction and generality, and the concrete and applicable mathematics that I enjoyed the most seemed to be out of fashion.

And so, almost by default, I entered the Ph.D. program at the Harvard Computation Laboratory. Most of the topics that were to become the bread and butter of the computer science curriculum had not even been thought of then, and so I took an eclectic collection of courses: switching theory, numerical analysis, applied mathematics, probability and statistics, operations research, electronics, and mathematical linguistics. While the curriculum left much to be desired in depth and coherence, there was a very special spirit in the air; we knew that we were witnessing the birth of a new scientific discipline centered on the computer. I discovered that I found beauty and elegance in the structure of algorithms, and that I had a knack for the discrete mathematics that formed the basis for the study of computers and computation. In short, I had stumbled more or less by accident into a field that was very much to my liking.

EASY AND HARD COMBINATORIAL PROBLEMS

Ever since those early days, I have had a special interest in combinatorial search problems—problems that can be likened to jigsaw puzzles where one has to assemble the parts of a structure in a particular way. Such problems involve searching through a finite, but extremely large, structured set of possible solutions, patterns, or arrangements, in order to find one that satisfies a stated set of conditions. Some examples of such problems are the placement and interconnection of components on an integrated circuit chip, the scheduling of the National Football League, and the routing of a fleet of school buses.

Within any one of these combinatorial puzzles lurks the possibility of a combinatorial explosion. Because of the vast, furiously growing number of possibilities that have to be searched through, a massive amount of computation may be encountered unless some subtlety is used in searching through the space of possible solutions. I'd like to begin the technical part of this talk by telling you about some of my first encounters with combinatorial explosions.

My first defeat at the hands of this phenomenon came soon after I joined the IBM Yorktown Heights Research Center in 1959. I was assigned to a group headed by J. P. Roth, a distinguished algebraic topolo-

gist who had made notable contributions to switching theory. Our group's mission was to create a computer program for the automatic synthesis of switching circuits. The input to the program was a set of Boolean formulas specifying how the outputs of the circuit were to depend on the inputs; the program was supposed to generate a circuit to do the job using a minimum number of logic gates. Figure 1 shows a circuit for the majority function of three variables; the output is high whenever at least two of the three variables x, y, and z are high.

The program we designed contained many elegant shortcuts and refinements, but its fundamental mechanism was simply to enumerate the possible circuits in order of increasing cost. The number of circuits that the program had to comb through grew at a furious rate as the number of input variables increased, and as a consequence, we could never progress beyond the solution of toy problems. Today, our optimism in even trying an enumerative approach may seem utterly naive, but we are not the only ones to have fallen into this trap; much of the work on automatic theorem proving over the past two decades has begun with an initial surge of excitement as toy problems were successfully solved, followed by disillusionment as the full seriousness of the combinatorial explosion phenomenon became apparent.

Around this same time, I began working on the traveling salesman problem with Michael Held of IBM. This problem takes its name from the situation of a salesman who wishes to visit all the cities in his territory, beginning and ending at his home city, while minimizing his total travel cost. In the special case where the cities are points in the plane and travel cost is equated with Euclidean distance, the problem is simply to find a polygon of minimum perimeter passing through all the cities (see Figure 2). A few years earlier, George Dantzig, Raymond Fulkerson, and Selmer Johnson at the Rand Corporation, using a mixture of manual and automatic computation, had succeeded in solving a 49-city problem, and we hoped to break their record.

Despite its innocent appearance, the traveling salesman problem has the potential for a combinatorial ex-

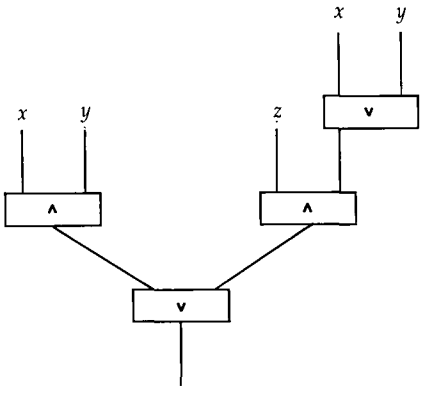


FIGURE 1. A Circuit for the Majority Function

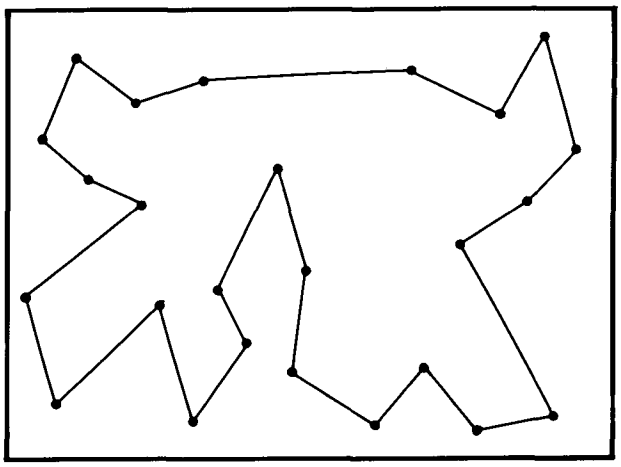


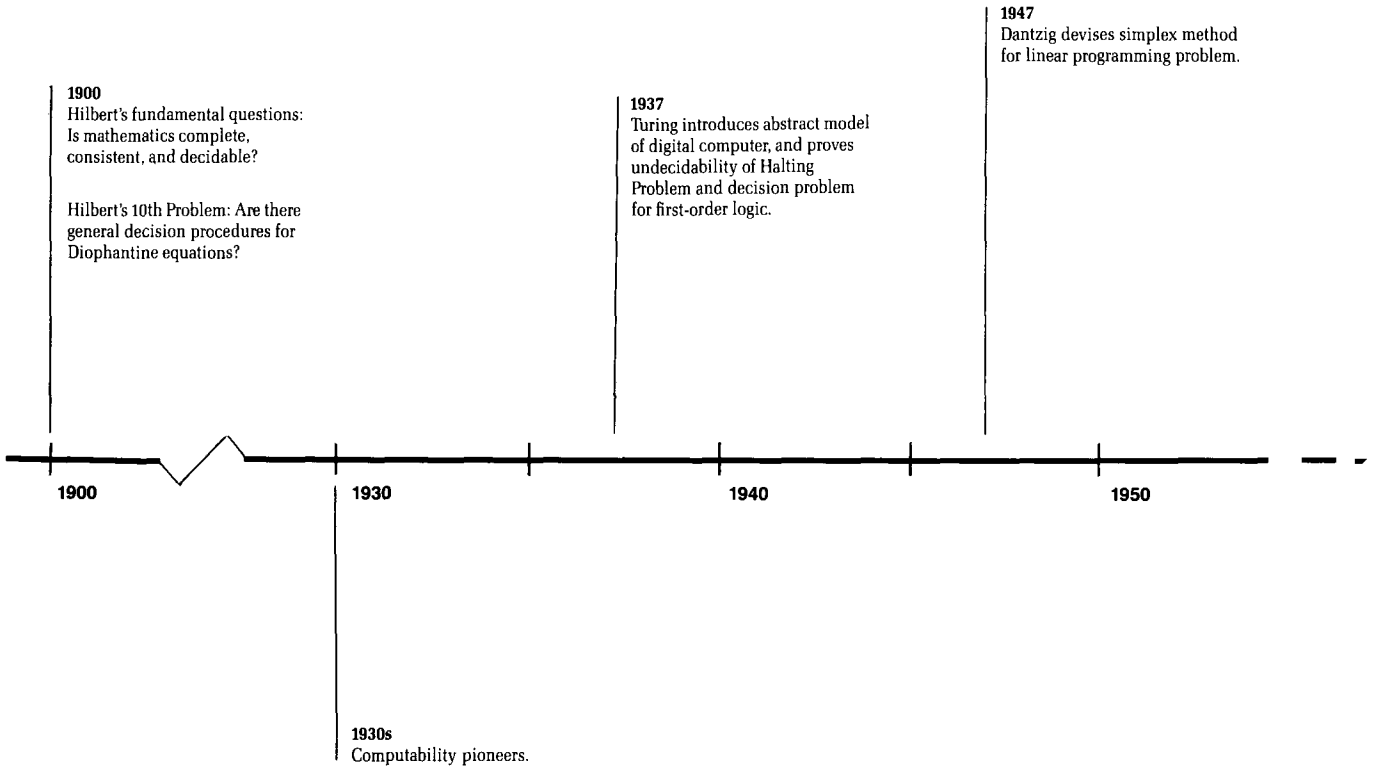
FIGURE 2. A Traveling Salesman Tour

plosion, since the number of possible tours through n cities in the plane is $(n - 1)!/2$, a very rapidly growing function of n. For example, when the number of cities is only 20, the time required for a brute-force enumeration of all possible tours, at the rate of a million tours per second, would be more than a thousand years.

Held and I tried a number of approaches to the traveling salesman problem. We began by rediscovering a shortcut based on dynamic programming that had originally been pointed out by Richard Bellman. The dynamic programming method reduced the search time to $n^2 2^n$, but this function also blows up explosively, and the method is limited in practice to problems with at most 16 cities. For a while, we gave up on the idea of solving the problem exactly, and experimented with local search methods that tend to yield good, but not optimal, tours. With these methods, one starts with a tour and repeatedly looks for local changes that will improve it. The process continues until a tour is found that cannot be improved by any such local change. Our local improvement methods were rather clumsy, and much better ones were later found by Shen Lin and Brian Kernighan at Bell Labs. Such quick-and-dirty methods are often quite useful in practice if a strictly optimal solution is not required, but one can never guarantee how well they will perform.

We then began to investigate branch-and-bound methods. Such methods are essentially enumerative in nature, but they gain efficiency by pruning away large parts of the space of possible solutions. This is done by computing a lower bound on the cost of every tour that includes certain links and fails to include certain others; if the lower bound is sufficiently large, it will follow that no such tour can be optimal. After a long series of unsuccessful experiments, Held and I stumbled upon a powerful method of computing lower bounds. This bounding technique allowed us to prune the search severely, so that we were able to solve problems with as many as 65 cities. I don't think any of my theoretical results have provided as great a thrill as the sight of the numbers pouring out of the computer on

THE DEVELOPMENT OF COMBINATORIAL OPTIMIZATION AND COMPUTATIONAL COMPLEXITY THEORY

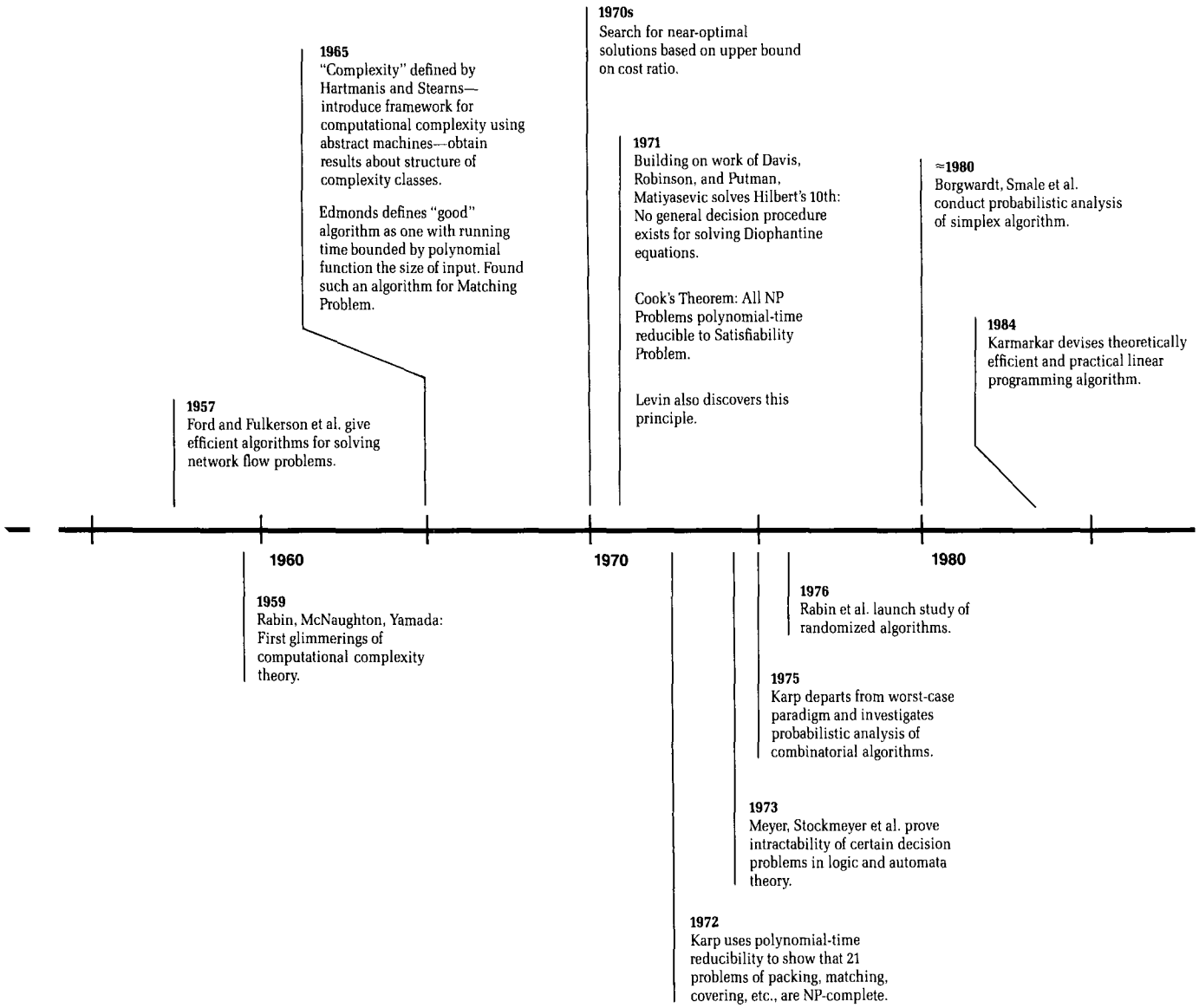


the night Held and I first tested our bounding method. Later we found out that our method was a variant of an old technique called Lagrangian relaxation, which is now used routinely for the construction of lower bounds within branch-and-bound methods.

For a brief time, our program was the world champion traveling-salesman-problem solver, but nowadays much more impressive programs exist. They are based on a technique called polyhedral combinatorics, which attempts to convert instances of the traveling salesman problem to very large linear programming problems. Such methods can solve problems with over 300 cities, but the approach does not completely eliminate combinatorial explosions, since the time required to solve a problem continues to grow exponentially as a function of the number of cities.

The traveling salesman problem has remained a fascinating enigma. A book of more than 400 pages has recently been published, covering much of what is known about this elusive problem. Later, we will discuss the theory of NP-completeness, which provides evidence that the traveling salesman problem is inherently intractable, so that no amount of clever algorithm design can ever completely defeat the potential for combinatorial explosions that lurks within this problem.

During the early 1960s, the IBM Research Laboratory at Yorktown Heights had a superb group of combinatorial mathematicians, and under their tutelage, I learned important techniques for solving certain combinatorial problems without running into combinatorial explosions. For example, I became familiar with Dantzig's famous simplex algorithm for linear programming. The linear programming problem is to find the point on a polyhedron in a high-dimensional space that is closest to a given external hyperplane (a polyhedron is the generalization of a polygon in two-dimensional space or an ordinary polyhedral body in three-dimensional



space, and a hyperplane is the generalization of a line in the plane or a plane in three-dimensional space). The closest point to the hyperplane is always a corner point, or vertex, of the polyhedron (see Figure 3). In practice, the simplex method can be depended on to find the desired vertex very quickly.

I also learned the beautiful network flow theory of Lester Ford and Fulkerson. This theory is concerned with the rate at which a commodity, such as oil, gas, electricity, or bits of information, can be pumped through a network in which each link has a capacity that limits the rate at which it can transmit the commodity. Many combinatorial problems that at first sight seem to have no relation to commodities flowing through networks can be recast as network flow problems, and the theory enables such problems to be solved elegantly and efficiently using no arithmetic operations except addition and subtraction.

Let me illustrate this beautiful theory by sketching the so-called Hungarian algorithm for solving a combinatorial optimization problem known as the marriage problem. This problem concerns a society consisting of n men and n women. The problem is to pair up the men and women in a one-to-one fashion at minimum cost, where a given cost is imputed to each pairing. These costs are given by an $n \times n$ matrix, in which each row corresponds to one of the men and each column to one of the women. In general, each pairing of the n men with the n women corresponds to a choice of n entries from the matrix, no two of which are in the same row or column; the cost of a pairing is the sum of the n entries that are chosen. The number of possible pairings is $n!$, a function that grows so rapidly that brute-force enumeration will be of little avail. Figure 4a shows a 3×3 example in which we see that the cost of pairing man 3 with woman 2 is equal to 9, the entry in the third row and second column of the given matrix.

The key observation underlying the Hungarian algo-

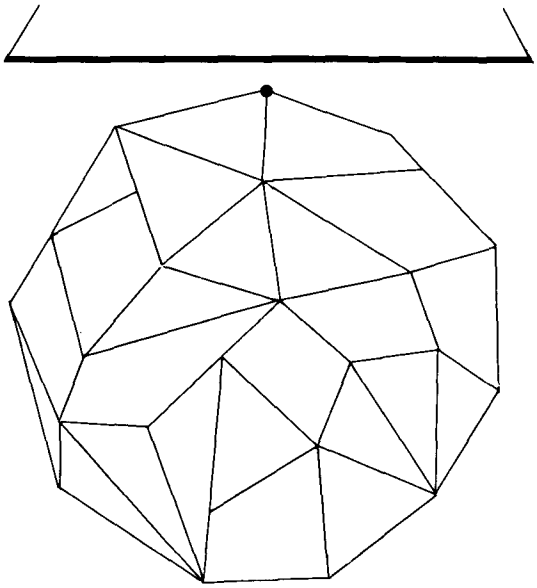


FIGURE 3. The Linear Programming Problem

rithm is that the problem remains unchanged if the same constant is subtracted from all the entries in one particular row of the matrix. Using this freedom to alter the matrix, the algorithm tries to create a matrix in which all the entries are nonnegative, so that every complete pairing has a nonnegative total cost, and in which there exists a complete pairing whose entries are all zero. Such a pairing is clearly optimal for the cost matrix that has been created, and it is optimal for the original cost matrix as well. In our 3 × 3 example, the algorithm starts by subtracting the least entry in each row from all the entries in that row, thereby creating a matrix in which each row contains at least one zero (Figure 4b). Then, to create a zero in each column, the algorithm subtracts, from all entries in each column that does not already contain a zero, the least entry in that column (Figure 4c). In this example, all the zeros in the resulting matrix lie in the first row or the third column; since a complete pairing contains only one entry from each row or column, it is not yet possible to find a complete pairing consisting entirely of zero entries. To create such a pairing, it is necessary to create a zero in the lower left part of the matrix. In this case, the algorithm creates such a zero by subtracting 1 from the first and second columns and adding 1 to the first row (Figure 4d). In the resulting nonnegative matrix, the three circled entries give a complete pairing of cost

$$\begin{bmatrix} 3 & 4 & 2 \\ 8 & 9 & 1 \\ 7 & 9 & 5 \end{bmatrix} \quad \begin{bmatrix} 1 & 2 & 0 \\ 7 & 8 & 0 \\ 2 & 4 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 0 & 0 \\ 6 & 6 & 0 \\ 1 & 2 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & \textcircled{0} & 1 \\ 5 & 5 & \textcircled{0} \\ \textcircled{0} & 1 & 0 \end{bmatrix}$$

(a) (b) (c) (d)

FIGURE 4. An Instance of the Marriage Problem

zero, and this pairing is therefore optimal, both in the final matrix and in the original one.

This algorithm is far subtler and more efficient than brute-force enumeration. The time required for it to solve the marriage problem grows only as the third power of n, the number of rows and columns of the matrix, and as a consequence, it is possible to solve examples with thousands of rows and columns.

The generation of researchers who founded linear programming theory and network flow theory had a pragmatic attitude toward issues of computational complexity: An algorithm was considered efficient if it ran fast enough in practice, and it was not especially important to prove it was fast in all possible cases. In 1967 I noticed that the standard algorithm for solving certain network flow problems had a theoretical flaw, which caused it to run very slowly on certain contrived examples. I found that it was not difficult to correct the flaw, and I gave a talk about my result to the combinatorics seminar at Princeton. The Princeton people informed me that Jack Edmonds, a researcher at the National Bureau of Standards, had presented very similar results at the same seminar during the previous week.

As a result of this coincidence, Edmonds and I began to work together on the theoretical efficiency of network flow algorithms, and in due course, we produced a joint paper. But the main effect of our collaboration was to reinforce some ideas about computational complexity that I was already groping toward and that were to have a powerful influence on the future course of my research. Edmonds was a wonderful craftsman who had used ideas related to linear programming to develop amazing algorithms for a number of combinatorial problems. But, in addition to his skill at constructing algorithms, he was ahead of his contemporaries in another important respect: He had developed a clear and precise understanding of what it meant for an algorithm to be efficient. His papers expounded the point of view that an algorithm should be considered "good" if its running time is bounded by a polynomial function of the size of the input, rather than, say, by an exponential function. For example, according to Edmonds's concept, the Hungarian algorithm for the marriage problem is a good algorithm because its running time grows as the third power of the size of the input. But as far as we know there may be no good algorithm for the traveling salesman problem, because all the algorithms we have tried experience an exponential growth in their running time as a function of problem size. Edmonds's definition gave us a clear idea of how to define the boundary between easy and hard combinatorial problems and opened up for the first time, at least in my thinking, the possibility that we might someday come up with a theorem that would prove or disprove the conjecture that the traveling salesman problem is inherently intractable.

THE ROAD TO NP-COMPLETENESS

Along with the developments in the field of combinatorial algorithms, a second major stream of research was

gathering force during the 1960s—computational complexity theory. The foundations for this subject were laid in the 1930s by a group of logicians, including Alan Turing, who were concerned with the existence or nonexistence of automatic procedures for deciding whether mathematical statements were true or false.

Turing and the other pioneers of computability theory were the first to prove that certain well-defined mathematical problems were undecidable, that is, that, in principle, there could not exist an algorithm capable of solving all instances of such problems. The first example of such a problem was the Halting Problem, which is essentially a question about the debugging of computer programs. The input to the Halting Problem is a computer program, together with its input data; the problem is to decide whether the program will eventually halt. How could there fail to be an algorithm for such a well-defined problem? The difficulty arises because of the possibility of unbounded search. The obvious solution is simply to run the program until it halts. But at what point does it become logical to give up, to decide that the program isn't going to halt? There seems to be no way to set a limit on the amount of search needed. Using a technique called diagonalization, Turing constructed a proof that no algorithm exists that can successfully handle all instances of the Halting Problem.

Over the years, undecidable problems were found in almost every branch of mathematics. An example from number theory is the problem of solving Diophantine equations: Given a polynomial equation such as

$$4xy^2 + 2xy^2z^3 - 11x^3y^2z^2 = -1164,$$

is there a solution in integers? The problem of finding a general decision procedure for solving such Diophantine equations was first posed by David Hilbert in 1900, and it came to be known as Hilbert's Tenth Problem. The problem remained open until 1971, when it was proved that no such decision procedure can exist.

One of the fundamental tools used in demarcating the boundary between solvable and unsolvable problems is the concept of reducibility, which was first brought into prominence through the work of logician Emil Post. Problem A is said to be reducible to problem B if, given a subroutine capable of solving problem B, one can construct an algorithm to solve problem A. As an example, a landmark result is that the Halting Problem is reducible to Hilbert's Tenth Problem (see Figure 5). It follows that Hilbert's Tenth Problem must be undecidable, since otherwise we would be able to use this reduction to derive an algorithm for the Halting Problem, which is known to be undecidable. The concept of reducibility will come up again when we discuss NP-completeness and the P : NP problem.

Another important theme that complexity theory inherited from computability theory is the distinction between the ability to solve a problem and the ability to check a solution. Even though there is no general method to find a solution to a Diophantine equation, it

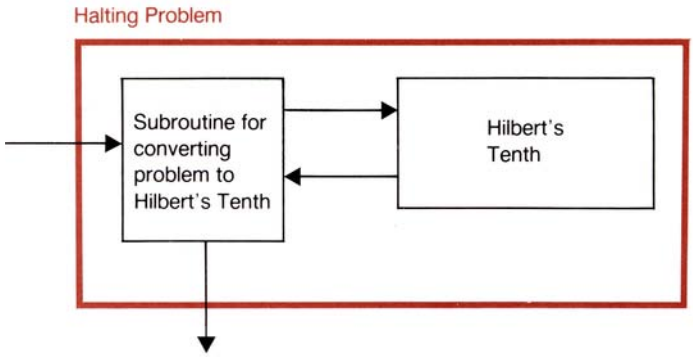


FIGURE 5. The Halting Problem Is Reducible to Hilbert's Tenth Problem

is easy to check a proposed solution. For example, to check whether $x = 3$, $y = 2$, $z = -1$ constitutes a solution to the Diophantine equation given above, one merely plugs in the given values and does a little arithmetic. As we will see later, the distinction between solving and checking is what the P : NP problem is all about.

Some of the most enduring branches of theoretical computer science have their origins in the abstract machines and other formalisms of computability theory. One of the most important of these branches is computational complexity theory. Instead of simply asking whether a problem is decidable at all, complexity theory asks how difficult it is to solve the problem. In other words, complexity theory is concerned with the capabilities of universal computing devices such as the Turing machine when restrictions are placed on their execution time or on the amount of memory they may use. The first glimmerings of complexity theory can be found in papers published in 1959 and 1960 by Michael Rabin and by Robert McNaughton and Hideo Yamada, but it is the 1965 paper by Juris Hartmanis and Richard Stearns that marks the beginning of the modern era of complexity theory. Using the Turing machine as their model of an abstract computer, Hartmanis and Stearns provided a precise definition of the "complexity class" consisting of all problems solvable in a number of steps bounded by some given function of the input length n. Adapting the diagonalization technique that Turing had used to prove the undecidability of the Halting Problem, they proved many interesting results about the structure of complexity classes. All of us who read their paper could not fail to realize that we now had a satisfactory formal framework for pursuing the questions that Edmonds had raised earlier in an intuitive fashion—questions about whether, for instance, the traveling salesman problem is solvable in polynomial time.

In that same year, I learned computability theory from a superb book by Hartley Rogers, who had been my teacher at Harvard. I remember wondering at the time whether the concept of reducibility, which was so central in computability theory, might also have a role to play in complexity theory, but I did not see how to make the connection. Around the same time, Michael Rabin, who was to receive the Turing Award in 1976,

was a visitor at the IBM Research Laboratory at Yorktown Heights, on leave from the Hebrew University in Jerusalem. We both happened to live in the same apartment building on the outskirts of New York City, and we fell into the habit of sharing the long commute to Yorktown Heights. Rabin is a profoundly original thinker and one of the founders of both automata theory and complexity theory, and through my daily discussions with him along the Sawmill River Parkway, I gained a much broader perspective on logic, computability theory, and the theory of abstract computing machines.

In 1968, perhaps influenced by the general social unrest that gripped the nation, I decided to move to the University of California at Berkeley, where the action was. The years at IBM had been crucial for my development as a scientist. The opportunity to work with such outstanding scientists as Alan Hoffman, Raymond Miller, Arnold Rosenberg, and Shmuel Winograd was simply priceless. My new circle of colleagues included Michael Harrison, a distinguished language theorist who had recruited me to Berkeley, Eugene Lawler, an expert on combinatorial optimization, Manuel Blum, a founder of complexity theory who has gone on to do outstanding work at the interface between number theory and cryptography, and Stephen Cook, whose work in complexity theory was to influence me so greatly a few years later. In the mathematics department, there were Julia Robinson, whose work on Hilbert's Tenth Problem was soon to bear fruit, Robert Solovay, a famous logician who later discovered an important randomized algorithm for testing whether a number is prime, and Steve Smale, whose ground-breaking work on the probabilistic analysis of linear programming algorithms was to influence me some years later. And across the Bay at Stanford were Dantzig, the father of linear programming, Donald Knuth, who founded the fields of data structures and analysis of algorithms, as well as Robert Tarjan, then a graduate student, and John Hopcroft, a sabbatical visitor from Cornell, who were brilliantly applying data structure techniques to the analysis of graph algorithms.

In 1971 Cook, who by then had moved to the University of Toronto, published his historic paper "On the Complexity of Theorem-Proving Procedures." Cook discussed the classes of problems that we now call P and NP, and introduced the concept that we now refer to as NP-completeness. Informally, the class P consists of all those problems that can be solved in polynomial time. Thus the marriage problem lies in P because the Hungarian algorithm solves an instance of size n in about n^3 steps, but the traveling salesman problem appears not to lie in P, since every known method of solving it requires exponential time. If we accept the premise that a computational problem is not tractable unless there is a polynomial-time algorithm to solve it, then all the tractable problems lie in P. The class NP consists of all those problems for which a proposed solution can be checked in polynomial time. For example, consider a version of the traveling salesman problem in which the input data consist of the distances between all pairs of cities, together with a "target number" T, and the task is to determine whether there exists a tour of length less than or equal to T. It appears to be extremely difficult to determine whether such a tour exists, but if a proposed tour is given to us, we can easily check whether its length is less than or equal to T; therefore, this version of the traveling salesman problem lies in the class NP. Similarly, through the device of introducing a target number T, all the combinatorial optimization problems normally considered in the fields of commerce, science, and engineering have versions that lie in the class NP.

So NP is the area into which combinatorial problems typically fall; within NP lies P, the class of problems that have efficient solutions. A fundamental question is, What is the relationship between the class P and the class NP? It is clear that P is a subset of NP, and the question that Cook drew attention to is whether P and NP might be the same class. If P were equal to NP, there would be astounding consequences: It would mean that every problem for which solutions are easy to check would also be easy to solve; it would mean that, whenever a theorem had a short proof, a uniform procedure would be able to find that proof quickly; it would mean that all the usual combinatorial optimization problems would be solvable in polynomial time. In short, it would mean that the curse of combinatorial explosions could be eradicated. But, despite all this heuristic evidence that it would be too good to be true if P and NP were equal, no proof that $P \neq NP$ has ever been found, and some experts even believe that no proof will ever be found.

The most important achievement of Cook's paper was to show that P = NP if and only if a particular computational problem called the Satisfiability Problem lies in P. The Satisfiability Problem comes from mathematical logic and has applications in switching theory, but it can be stated as a simple combinatorial puzzle: Given several sequences of upper- and lowercase letters, is it possible to select a letter from each sequence without selecting both the upper- and lowercase versions of any letter? For example, if the sequences are Abc, BC, aB, and ac it is possible to choose A from the first sequence, B from the second and third, and c from the fourth; note that the same letter can be chosen more than once, provided we do not choose both its uppercase and lowercase versions. An example where there is no way to make the required selections is given by the four sequences AB, Ab, aB, and ab.

The Satisfiability Problem is clearly in NP, since it is easy to check whether a proposed selection of letters satisfies the conditions of the problem. Cook proved that, if the Satisfiability Problem is solvable in polynomial time, then every problem in NP is solvable in polynomial time, so that P = NP. Thus we see that this seemingly bizarre and inconsequential problem is an archetypal combinatorial problem; it holds the key to the efficient solution of all problems in NP.

Cook's proof was based on the concept of reducibility that we encountered earlier in our discussion of computability theory. He showed that any instance of a problem in NP can be transformed into a corresponding instance of the Satisfiability Problem in such a way that the original has a solution if and only if the satisfiability instance does. Moreover, this translation can be accomplished in polynomial time. In other words, the Satisfiability Problem is general enough to capture the structure of any problem in NP. It follows that, if we could solve the Satisfiability Problem in polynomial time, then we would be able to construct a polynomial-time algorithm to solve any problem in NP. This algorithm would consist of two parts: a polynomial-time translation procedure that converts instances of the given problem into instances of the Satisfiability Problem, and a polynomial-time subroutine to solve the Satisfiability Problem itself (see Figure 6).

Upon reading Cook's paper, I realized at once that his concept of an archetypal combinatorial problem was a formalization of an idea that had long been part of the folklore of combinatorial optimization. Workers in that field knew that the integer programming problem, which is essentially the problem of deciding whether a system of linear inequalities has a solution in integers, was general enough to express the constraints of any of the commonly encountered combinatorial optimization problems. Dantzig had published a paper on that theme in 1960. Because Cook was interested in theorem proving rather than combinatorial optimization, he had chosen a different archetypal problem, but the basic idea was the same. However, there was a key difference: By using the apparatus of complexity theory, Cook had created a framework within which the archetypal nature of a given problem could become a theorem, rather than an informal thesis. Interestingly, Leonid Levin, who was then in Leningrad and is now a professor at Boston University, independently discovered essentially the same set of ideas. His archetypal problem had to do with tilings of finite regions of the plane with dominoes.

I decided to investigate whether certain classic combinatorial problems, long believed to be intractable,

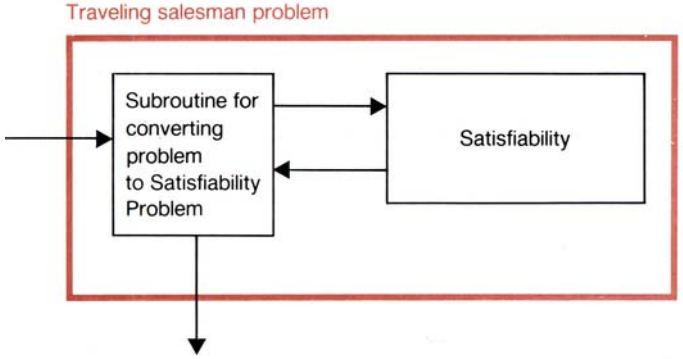


FIGURE 6. The Traveling Salesman Problem Is Polynomial-Time Reducible to the Satisfiability Problem

were also archetypal in Cook's sense. I called such problems "polynomial complete," but that term became superseded by the more precise term "NP-complete." A problem is NP-complete if it lies in the class NP, and every problem in NP is polynomial-time reducible to it. Thus, by Cook's theorem, the Satisfiability Problem is NP-complete. To prove that a given problem in NP is NP-complete, it suffices to show that some problem already known to be NP-complete is polynomial-time reducible to the given problem. By constructing a series of polynomial-time reductions, I showed that most of the classical problems of packing, covering, matching, partitioning, routing, and scheduling that arise in combinatorial optimization are NP-complete. I presented these results in 1972 in a paper called "Reducibility among Combinatorial Problems." My early results were quickly refined and extended by other workers, and in the next few years, hundreds of different problems, arising in virtually every field where computation is done, were shown to be NP-complete.

COPING WITH NP-COMPLETE PROBLEMS

I was rewarded for my research on NP-complete problems with an administrative post. From 1973 to 1975, I headed the newly formed Computer Science Division at Berkeley, and my duties left me little time for research. As a result, I sat on the sidelines during a very active period, during which many examples of NP-complete problems were found, and the first attempts to get around the negative implications of NP-completeness got under way.

The NP-completeness results proved in the early 1970s showed that, unless P = NP, the great majority of the problems of combinatorial optimization that arise in commerce, science, and engineering are intractable: No methods for their solution can completely evade combinatorial explosions. How, then, are we to cope with such problems in practice? One possible approach stems from the fact that near-optimal solutions will often be good enough: A traveling salesman will probably be satisfied with a tour that is a few percent longer than the optimal one. Pursuing this approach, researchers began to search for polynomial-time algorithms that were guaranteed to produce near-optimal solutions to NP-complete combinatorial optimization problems. In most cases, the performance guarantee for the approximation algorithm was in the form of an upper bound on the ratio between the cost of the solution produced by the algorithm and the cost of an optimal solution.

Some of the most interesting work on approximation algorithms with performance guarantees concerned the one-dimensional bin-packing problem. In this problem, a collection of items of various sizes must be packed into bins, all of which have the same capacity. The goal is to minimize the number of bins used for the packing, subject to the constraint that the sum of the sizes of the items packed into any bin may not exceed the bin capacity. During the mid 1970s, a series of papers on approximation algorithms for bin packing culminated in David Johnson's analysis of the first-fit-decreasing algo-

rithm. In this simple algorithm, the items are considered in decreasing order of their sizes, and each item in turn is placed in the first bin that can accept it. In the example in Figure 7, for instance, there are four bins each with a capacity of 10, and eight items ranging in size from 2 to 8. Johnson showed that this simple method was guaranteed to achieve a relative error of at most 2/9; in other words, the number of bins required was never more than about 22 percent greater than the number of bins in an optimal solution. Several years later, these results were improved still further, and it was eventually shown that the relative error could be made as small as one liked, although the polynomial-time algorithms required for this purpose lacked the simplicity of the first-fit-decreasing algorithm that Johnson analyzed.

The research on polynomial-time approximation algorithms revealed interesting distinctions among the NP-complete combinatorial optimization problems. For some problems, the relative error can be made as small as one likes; for others, it can be brought down to a certain level, but seemingly no further; other problems have resisted all attempts to find an algorithm with bounded relative error; and finally, there are certain problems for which the existence of a polynomial-time approximation algorithm with bounded relative error would imply that P = NP.

During the sabbatical year that followed my term as an administrator, I began to reflect on the gap between theory and practice in the field of combinatorial optimization. On the theoretical side, the news was bleak. Nearly all the problems one wanted to solve were NP-complete, and in most cases, polynomial-time approximation algorithms could not provide the kinds of performance guarantees that would be useful in practice. Nevertheless, there were many algorithms that seemed to work perfectly well in practice, even though they lacked a theoretical pedigree. For example, Lin and Kernighan had developed a very successful local improvement strategy for the traveling salesman problem. Their algorithm simply started with a random tour and kept improving it by adding and deleting a few links, until a tour was eventually created that could not be improved by such local changes. On contrived instances, their algorithm performed disastrously, but in practical instances, it could be relied on to give nearly optimal solutions. A similar situation prevailed for the simplex algorithm, one of the most important of all computational methods: It reliably solved the large linear programming problems that arose in applications, despite the fact that certain artificially constructed examples caused it to run for an exponential number of steps.

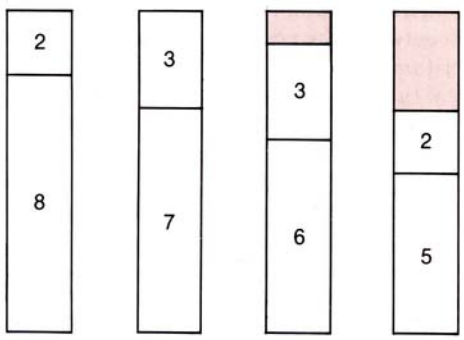


FIGURE 7. A Packing Created by the First-Fit Decreasing Algorithm

It seemed that the success of such inexact or rule-of-thumb algorithms was an empirical phenomenon that needed to be explained. And it further seemed that the explanation of this phenomenon would inevitably require a departure from the traditional paradigms of complexity theory, which evaluate an algorithm according to its performance on the worst possible input that can be presented to it. The traditional worst-case analysis—the dominant strain in complexity theory—corresponds to a scenario in which the instances of a problem to be solved are constructed by an infinitely intelligent adversary who knows the structure of the algorithm and chooses inputs that will embarrass it to the maximal extent. This scenario leads to the conclusion that the simplex algorithm and the Lin–Kernighan algorithm are hopelessly defective. I began to pursue another approach, in which the inputs are assumed to come from a user who simply draws his instances from some reasonable probability distribution, attempting neither to foil nor to help the algorithm.

In 1975 I decided to bite the bullet and commit myself to an investigation of the probabilistic analysis of combinatorial algorithms. I must say that this decision required some courage, since this line of research had its detractors, who pointed out quite correctly that there was no way to know what inputs were going to be presented to an algorithm, and that the best kind of guarantees, if one could get them, would be worst-case guarantees. I felt, however, that in the case of NP-complete problems we weren't going to get the worst-case guarantees we wanted, and that the probabilistic approach was the best way and perhaps the only way to understand why heuristic combinatorial algorithms worked so well in practice.

Probabilistic analysis starts from the assumption that the instances of a problem are drawn from a specified probability distribution. In the case of the traveling salesman problem, for example, one possible assumption is that the locations of the n cities are drawn independently from the uniform distribution over the unit square. Subject to this assumption, we can study the probability distribution of the length of the optimal tour or the length of the tour produced by a particular algorithm. Ideally, the goal is to prove that some simple algorithm produces optimal or near-optimal solutions with high probability. Of course, such a result is meaningful only if the assumed probability distribution of problem instances bears some resemblance to the population of instances that arise in real life, or if the probabilistic analysis is robust enough to be valid for a wide range of probability distributions.

Among the most striking phenomena of probability theory are the laws of large numbers, which tell us that the cumulative effect of a large number of random events is highly predictable, even though the outcomes of the individual events are highly unpredictable. For example, we can confidently predict that, in a long series of flips of a fair coin, about half the outcomes will be heads. Probabilistic analysis has revealed that the same phenomenon governs the behavior of many combinatorial optimization algorithms when the input data are drawn from a simple probability distribution: With very high probability, the execution of the algorithm evolves in a highly predictable fashion, and the solution produced is nearly optimal. For example, a 1960 paper by Beardwood, Halton, and Hammersley shows that, if the n cities in a traveling salesman problem are drawn independently from the uniform distribution over the unit square, then, when n is very large, the length of the optimal tour will almost surely be very close to a certain absolute constant times the square root of the number of cities. Motivated by their result, I showed that, when the number of cities is extremely large, a simple divide-and-conquer algorithm will almost surely produce a tour whose length is very close to the length of an optimal tour (see Figure 8). The algorithm starts by partitioning the region where the cities lie into rectangles, each of which contains a small number of cities. It then constructs an optimal

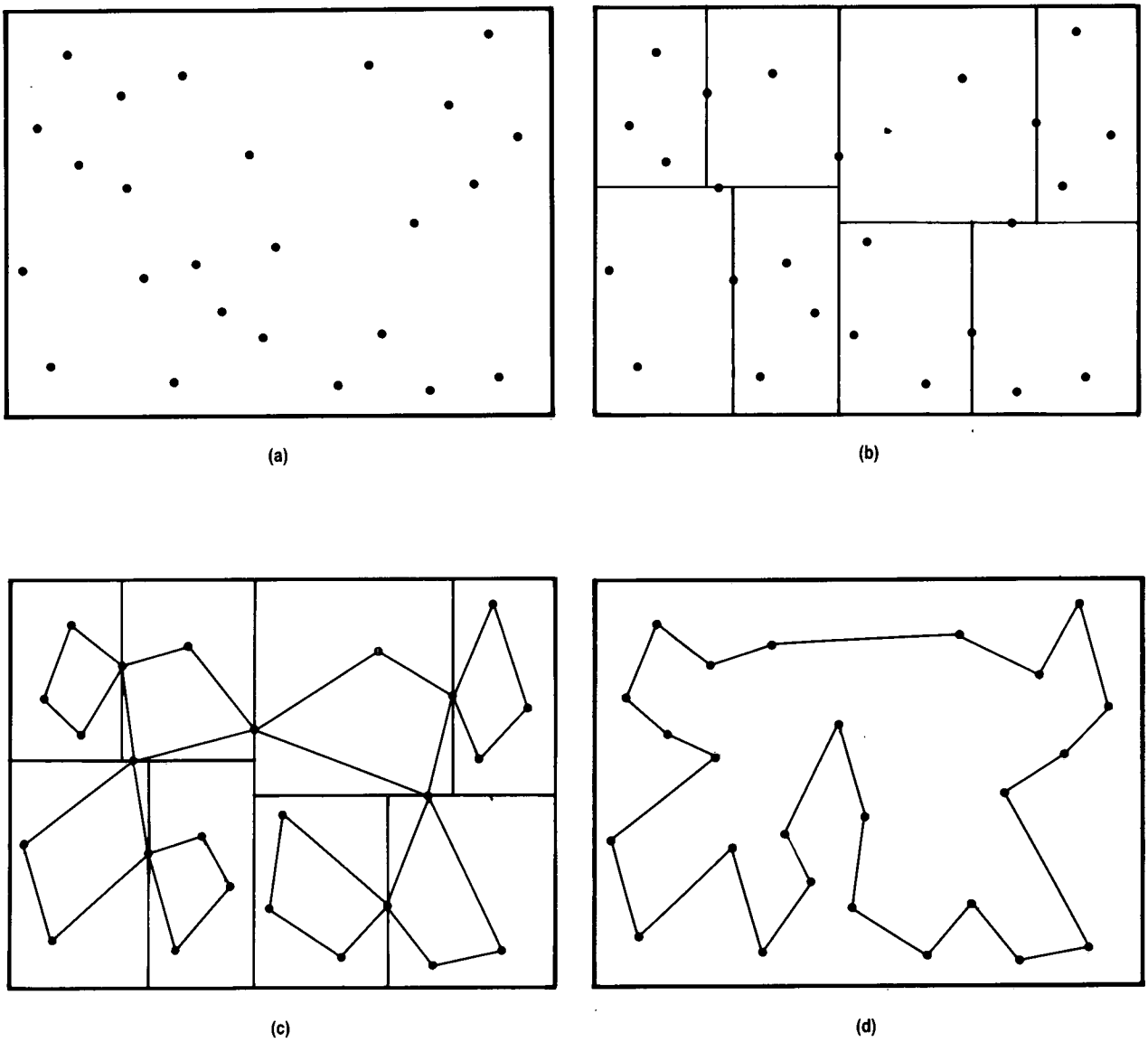


FIGURE 8. A Divide-and-Conquer Algorithm for the Traveling Salesman Problem in the Plane

tour through the cities in each rectangle. The union of all these little tours closely resembles an overall traveling salesman tour, but differs from it because of extra visits to those cities that lie on the boundaries of the rectangles. Finally, the algorithm performs a kind of local surgery to eliminate these redundant visits and produce a tour.

Many further examples can be cited in which simple approximation algorithms almost surely give near-optimal solutions to random large instances of NP-complete optimization problems. For example, my student Sally Floyd, building on earlier work on bin packing by Bentley, Johnson, Leighton, McGeoch, and McGeoch, recently showed that, if the items to be packed are drawn independently from the uniform distribution over the interval (0, 1/2), then, no matter how many items there are, the first-fit decreasing algorithm will almost surely produce a packing with less than 10 bins worth of wasted space.

Some of the most notable applications of probabilistic analysis have been to the linear programming problem. Geometrically, this problem amounts to finding the vertex of a polyhedron closest to some external hyperplane. Algebraically, it is equivalent to minimizing a linear function subject to linear inequality constraints. The linear function measures the distance to the hyperplane, and the linear inequality constraints correspond to the hyperplanes that bound the polyhedron.

The simplex algorithm for the linear programming problem is a hill-climbing method. It repeatedly slides from vertex to neighboring vertex, always moving closer to the external hyperplane. The algorithm terminates when it reaches a vertex closer to this hyperplane than any neighboring vertex; such a vertex is guaranteed to be an optimal solution. In the worst case, the simplex algorithm requires a number of iterations that grow exponentially with the number of linear inequalities needed to describe the polyhedron, but in practice, the number of iterations is seldom greater than three or four times the number of linear inequalities.

Karl-Heinz Borgwardt of West Germany and Steve Smale of Berkeley were the first researchers to use probabilistic analysis to explain the unreasonable success of the simplex algorithm and its variants. Their analyses hinged on the evaluation of certain multidimensional integrals. With my limited background in mathematical analysis, I found their methods impenetrable. Fortunately, one of my colleagues at Berkeley, Ilan Adler, suggested an approach that promised to lead to a probabilistic analysis in which there would be virtually no calculation; one would use certain symmetry principles to do the required averaging and magically come up with the answer.

Pursuing this line of research, Adler, Ron Shamir, and I showed in 1983 that, under a reasonably wide range of probabilistic assumptions, the expected number of iterations executed by a certain version of the simplex algorithm grows only as the square of the number of linear inequalities. The same result was also obtained via multidimensional integrals by Michael Todd and by Adler and Nimrod Megiddo. I believe that these results contribute significantly to our understanding of why the simplex method performs so well.

The probabilistic analysis of combinatorial optimization algorithms has been a major theme in my research over the past decade. In 1975, when I first committed myself to this research direction, there were very few examples of this type of analysis. By now there are hundreds of papers on the subject, and all of the classic combinatorial optimization problems have been subjected to probabilistic analysis. The results have provided a considerable understanding of the extent to which these problems can be tamed in practice. Nevertheless, I consider the venture to be only partially successful. Because of the limitations of our techniques, we continue to work with the most simplistic of probabilistic models, and even then, many of the most interesting and successful algorithms are beyond the scope of our analysis. When all is said and done, the design of practical combinatorial optimization algorithms remains as much an art as it is a science.

RANDOMIZED ALGORITHMS

Algorithms that toss coins in the course of their execution have been proposed from time to time since the earliest days of computers, but the systematic study of such randomized algorithms only began around 1976. Interest in the subject was sparked by two surprisingly efficient randomized algorithms for testing whether a number n is prime; one of these algorithms was proposed by Solovay and Volker Strassen, and the other by Rabin. A subsequent paper by Rabin gave further examples and motivation for the systematic study of randomized algorithms, and the doctoral thesis of John Gill, under the direction of my colleague Blum, laid the foundations for a general theory of randomized algorithms.

To understand the advantages of coin tossing, let us turn again to the scenario associated with worst-case analysis, in which an all-knowing adversary selects the instances that will tax a given algorithm most severely. Randomization makes the behavior of an algorithm unpredictable even when the instance is fixed, and thus can make it difficult, or even impossible, for the adversary to select an instance that is likely to cause trouble. There is a useful analogy with football, in which the algorithm corresponds to the offensive team and the adversary to the defense. A deterministic algorithm is like a team that is completely predictable in its play calling, permitting the other team to stack its defenses. As any quarterback knows, a little diversification in the play calling is essential for keeping the defensive team honest.

As a concrete illustration of the advantages of coin tossing, I present a simple randomized pattern-matching algorithm invented by Rabin and myself in 1980. The pattern-matching problem is a fundamental one in text processing. Given a string of n bits called

Pattern 11001

Text 011011101 11001 00

FIGURE 9. A Pattern-Matching Problem

the pattern, and a much longer bit string called the text, the problem is to determine whether the pattern occurs as a consecutive block within the text (see Figure 9). A brute-force method of solving this problem is to compare the pattern directly with every n-bit block within the text. In the worst case, the execution time of this method is proportional to the product of the length of the pattern and the length of the text. In many text processing applications, this method is unacceptably slow unless the pattern is very short.

Our method gets around the difficulty by a simple hashing trick. We define a "fingerprinting function" that associates with each string of n bits a much shorter string called its *fingerprint*. The fingerprinting function is chosen so that it is possible to march through the text, rapidly computing the fingerprint of every n-bit-long block. Then, instead of comparing the pattern with each such block of text, we compare the fingerprint of the pattern with the fingerprint of every such block. If the fingerprint of the pattern differs from the fingerprint of each block, then we know that the pattern does not occur as a block within the text.

The method of comparing short fingerprints instead of long strings greatly reduces the running time, but it leads to the possibility of false matches, which occur when some block of text has the same fingerprint as the pattern, even though the pattern and the block of text are unequal. False matches are a serious problem; in fact, for any particular choice of fingerprinting function it is possible for an adversary to construct an example of a pattern and a text such that a false match occurs at every position of the text. Thus, some backup method is needed to defend against false matches, and the advantages of the fingerprinting method seem to be lost.

Fortunately, the advantages of fingerprinting can be restored through randomization. Instead of working with a single fingerprinting function, the randomized method has at its disposal a large family of different easy-to-compute fingerprinting functions. Whenever a problem instance, consisting of a pattern and a text, is presented, the algorithm selects a fingerprinting function at random from this large family, and uses that function to test for matches between the pattern and the text. Because the fingerprinting function is not known in advance, it is impossible for an adversary to construct a problem instance that is likely to lead to false matches; it can be shown that, no matter how the pattern and the text are selected, the probability of a false match is very small. For example, if the pattern is 250 bits long and the text is 4000 bits long, one can work with easy-to-compute 32-bit fingerprints and still guarantee that the probability of a false match is less than one in a thousand in every possible instance. In many text processing applications, this probabilistic guarantee is good enough to eliminate the need for a backup routine, and thus the advantages of the fingerprinting approach are regained.

Randomized algorithms and probabilistic analysis of algorithms are two contrasting ways to depart from the worst-case analysis of deterministic algorithms. In the former case, randomness is injected into the behavior of the algorithm itself, and in the latter case, randomness is assumed to be present in the choice of problem instances. The approach based on randomized algorithms is, of course, the more appealing of the two, since it avoids assumptions about the environment in which the algorithm will be used. However, randomized algorithms have not yet proved effective in combating the combinatorial explosions characteristic of NP-complete problems, and so it appears that both of these approaches will continue to have their uses.

CONCLUSION

This brings me to the end of my story, and I would like to conclude with a brief remark about what it's like to be working in theoretical computer science today. Whenever I participate in the annual ACM Theory of Computing Symposium, or attend the monthly Bay Area Theory Seminar, or go up the hill behind the Berkeley campus to the Mathematical Sciences Research Institute, where a year-long program in computational complexity is taking place, I am struck by the caliber of the work that is being done in this field. I am proud to be associated with a field of research in which so much excellent work is being done, and pleased that I'm in a position, from time to time, to help prodigiously talented young researchers get their bearings in this field. Thank you for giving me the opportunity to serve as a representative of my field on this occasion.

CR Categories and Subject Descriptors: A.0 **[General Literature]**: *biographies/autobiographies*; F. 0 **[Theory of Computation]**: General; F.1.1 **[Computation by Abstract Devices]**: Models of Computation—*computability theory*; F.1.2 **[Computation by Abstract Devices]**: Modes of Computation—*parallelism, probabilistic computation*; F.1.3 **[Computation by Abstract Devices]**: Complexity Classes—*reducibility and completeness, relations among complexity classes*; F.2.0 **[Analysis of Algorithms and Problem Complexity]**: General; G.2.1 **[Discrete Mathematics]**: Combinatorics; G.2.2 **[Discrete Mathematics]**: Graph Theory; K.2 **[History of Computing]**: *people*

General Terms: Performance, Theory

Additional Key Words and Phrases: Richard Karp, Turing Award

Author's Present Address: Richard Karp, 571 Evans Hall, University of California, Berkeley, CA 94720.