BULLETIN (New Series) OF THE
AMERICAN MATHEMATICAL SOCIETY
Volume 2, Number 1, January 1980

0002-9904/80/0000-0009/$01.75

BOOK REVIEWS

An introduction to the general theory of algorithms, by Michael Machtey and Paul Young, North-Holland, New York, Oxford, Shannon, 1978, vii + 264 pp.

In order to discuss this book properly, a brief recounting of some of the history of the theory of algorithms is in order.

Although the notion of mathematical algorithm had been used for centuries, the present rigorous definition was not accomplished until Gödel, Church, Turing and Kleene gave their formulations in the 1930's. That all these formulations, and several later ones, proved to be equivalent is a kind of event that is indeed rare in mathematics. The assertion that these formulations coincide with the intuitive notion of mathematical algorithm is known as Church's Thesis. In the 1930's Gödel also demonstrated the existence of algorithmically unsolvable problems, particularly the unsolvability of the decision problem for arithmetic (which dashed the hopes of Hilbert's Program, and at the same time was the first application of Cantor's diagonalization technique in this area).

In the 1940's the construction of computers that could implement algorithms had a profound effect on the development of the theory. Von Neumann explored the program-data dichotomy (one man's program is another man's data) and produced the notion of a stored program computer and the manipulation of one program by another. Finally, in the 1960's, after a decade of programming experience, the realization of the importance of an algorithm's complexity lead to the replacement of the notion of algorithmic unsolvability by algorithmic intractability. That is: while certain problems may be theoretically algorithmically solvable, their intrinsic computational complexity prevents any algorithm from running to completion.

As in most texts in this area, the invariance of the notion of algorithmic computability is demonstrated by introducing several models of computation and showing them to be equivalent. Unlike most texts, however, the present one accomplishes this in the first fifty pages, within the first chapter. While many details of this equivalence are omitted and left to the reader to supply, anyone with formal mathematical experience or reasonable computer programming experience can readily follow the outline of the simulations, even if he cannot completely fill the gaps. The other significant feature of the first chapter is that measures of computational complexity for the various models considered are introduced there as well.

Not only are the usual models of computation equivalent, they are effectively intertranslatable. The properties common to the standard models that are responsible for such intertranslatability can be abstracted axiomatically through what are called acceptable programming systems (or acceptable Gödel numberings). An acceptable (universal) programming system (§3.1) is one in which there exists (1) a universal program, which, given any program and any input, computes the output of that program on that input, and (2) an

effective transformation of any pair of programs into a program that computes the composition of their corresponding functions. The usual definition of acceptable programming system differs from this one (although they are equivalent) in that (2) above is replaced by: (2′) an effective transformation of any program and any data into a program that consists of the given program with the given data incorporated into it. Composition, of course, is mathematically fundamental, but the notion of a program with incorporated data is computationally fundamental. The latter is apparent from Kleene's Recursion Theorem (§3.4), which in one form states that for any computable function it is possible to construct a program with a description of itself incorporated into it, which computes the given function. In another form it states that every computable transformation of programs has a fixed point, i.e., a program whose input-output behavior is unaffected by that transformation. The Recursion Theorem is often used to construct computable functions all of whose programs satisfy certain conditions. The intertranslatability of acceptable programming systems mentioned at the beginning of this paragraph achieves its strongest form in Rogers' Isomorphism Theorem (§3.4), which states that between any two acceptable programming systems there is a computable input-output behavior preserving bijection, i.e., a computable bijection between equivalent programs.

The concept of algorithmic unsolvability (a view of computability from without) is touched on at several points in this book, and, except for degrees of unsolvability, is complete with respect to the fundamental issues. Most fundamental, of course, is the existence (§2.5, §2.6) of algorithmically unsolvable problems, the most widely known being the halting problem (§3.2). The standard technique for demonstrating the unsolvability of a problem is to reduce a known unsolvable problem to it (§3.2), i.e., to establish the contradictory argument that an algorithm for the given problem would provide an algorithm for the known unsolvable problem. The decision problems of most interest to computer scientists concern the functional properties of programs, but Rice's Theorem (§3.2) shows that "in any acceptable programming system there are *no* nontrivial properties of the input-output behavior of programs, that is, properties of [computable] *functions* which can be decided by looking at the *programs*."

As indicated above, the notions of computability and algorithmic unsolvability trace their formal beginnings to Gödel's work on mathematical logic in the 1930's, and it is appropriate that one chapter of the book is devoted to this. Among the results considered are two fundamental ones: Gödel's Undecidability for Theorems (§4.3), which states that in any formal system that is capable of representing arithmetic (and hence the computable functions), the problem of deciding whether a given statement is a theorem of that system is algorithmically undecidable (this can be viewed as an analog of the halting problem); and Gödel's Incompleteness Theorem (§4.3), which states that for such systems, since the provably false statements can be effectively enu merated as well as the theorems, there must exist statements that can be neither proved nor disproved within the system.

There are aspects of computable functions other than their input-output behavior that are also of great importance, namely, the time taken or memory

used by programs for computing them. This notion of computational complexity can be studied axiomatically by means of axioms given by Blum for acceptable programming systems. Such a general approach is justified by an equivalence result (§5.2) which states that given any two acceptable programming systems and any two computational complexity measures for them, one can effectively translate results (i.e., upper or lower bounds) for the complexity of functions with respect to one system into corresponding results for the other system. The first observation to be made regarding complexity is that there exist computable functions that are arbitrarily difficult to compute, i.e., all of whose programs exceed an a priori specified amount of computation resources. This naturally leads to the establishment of a hierarchy of the computable functions based on their computational complexity. The Gap Theorem (§5.3) warns that care must be taken in choosing the strata of such a hierarchy. Perhaps the deepest result in axiomatic computational complexity theory is Blum's Speedup Theorem (§5.3), which states that there exist computable functions that are so difficult to compute that corresponding to any program for such a function there is another program for it that is more efficient than the first by an a priori arbitrarily specified amount. The impact of this result is that in general, for any computable function, there is no single complexity function that can be associated with it.

For real computations, of course, the computational complexity of a program is of practical interest. Any problem that cannot be solved in polynomial time is for all practical purposes unsolvable. Examples of problems that are of practical interest but that are intractable are given in Chapter six of the text. There are, moreover, a number of real problems of great importance, such as the travelling salesman problem, whose tractability is at present unknown. They can, however, be solved in polynomial time if arbitrary parallelism is permitted, which brings the present discussion to the forefront of theoretical computer science research and introduces the current most outstanding open problem, namely, $P = NP$?(§7.3). At this point, the book closes and so too does our discussion of its contents.

Since the book is intended to be an introductory graduate level textbook, some comments on its use are in order. It is an excellent text for beginning graduate students with a good formal mathematical background. For computer science students, however, care must be exercised (by the instructor) to make sure that they do not run into possibly insurmountable difficulties of notation and abstraction. With proper care and perhaps "hand waving", instead of some formal proofs, these potential dificulties can be avoided. The book exhibits much pedagogical concern with providing abundant motivation for many of the formal investigations, and there are numerous exercises involving further development of the text material.

In conclusion, this book is significant in its early incorporation of computational complexity into the study of the theory of computability. Perhaps future texts will involve computational complexity not only as an object of investigation but also as a tool in the investigation of computability itself.

ROBERT P. DALEY