linear algebra texts; it would have been well worth taking space to develop this theme at some length.

To summarize, after its renaissance in the 1950s, GI theory passed through its infancy in the 1960s and its adolescence in the 1970s. For the 1980s, it is reasonable to expect a coming-of-age in which abstract algebra, operator theory, and mathematical logic may begin to play a larger role on the theoretical side, while presumably also several significant and interesting new applications remain to be found as GIs become better understood and more widely known. To this end, [CM] deserves a place, together with [1], [2], and [3], on the shelf of every GI specialist and potential GI user (since each source offers much material not treated in the other three), and is also to be recommended to the interested general reader or student. While only a few readers will wish to follow every topic to its last details, this book has enough solid content to make it a valuable reference, and even the beginner should have little difficulty in selecting those sections most deserving of intensive study.

REFERENCES

1. A. Ben-Israel and T. N. E. Greville, *Generalized inverses: Theory and applications*, Wiley, New York, 1974.

2. M. Z. Nashed (ed.), *Generalized inverses and applications*, Academic Press, New York, 1976.

3. C. R. Rao and S. K. Mitra, *Generalized inverse of matrices and its applications*, Wiley, New York, 1971.

MICHAEL P. DRAZIN

BULLETIN (New Series) OF THE
AMERICAN MATHEMATICAL SOCIETY
Volume 3, Number 2, September 1980

0002-9904/80/0000-0415/$02.50

Computers and intractability: A guide to the theory of NP-completeness, by Michael R. Garey and David S. Johnson, W. H. Freeman and Company, San Francisco, 1979, xii + 338 pp., $10.00 (paper).

There is a class of algorithmic problems that is currently receiving a great deal of attention from computer scientists and applied mathematicians: the class of "*NP*-complete" problems. Examples of problems in this class are the satisfiability problem for conjunctive normal form statements in the propositional calculus, the three-colorability problem in graph theory, the travelling salesman problem, the three-dimensional matching problem (i.e., the generalization of the classical marriage problem in the setting of three sexes and three-way marriages), the bin packing problem, and the integer programming problem.[1] For each such problem an algorithm is known for solving all instances of the problem; the basis for the monograph reviewed here is the more refined question of whether the problem is tractable, i.e., whether an algorithm exists that solves all instances of the problem and that has running time bounded by a polynomial in the size of the input. (This interpretation of the notion of tractability is due to Cobham [1] and to Edmunds [2].) It is not

[1]At this time it is not known whether the linear programming problem is *NP*-complete, irrespective of the statements in *The New York Times*, November 7, 1979, p. 1.

known whether any problem in this class is tractable, but an important property of the class is that every problem in the class is tractable if and only if one such problem is tractable.

It is useful to discuss a specific example.

Let $U = \{u_1, \ldots, u_n\}$ be a finite set of Boolean variables. If u is a variable, then u and $\bar{u}$ are *literals*. A *clause* over U is a set of literals over U that are joined together by the Boolean *or*. A statement in *conjunctive normal form* is a finite collection of clauses joined together by the Boolean *and*. A *truth assignment* for U is a function t: $U \to \{T, F\}$. If $t(u) = T$, then u is "true" under that assignment; otherwise, u is "false". A literal u is true (false) under an assignment if and only if the literal $\bar{u}$ is false (true). A statement in conjunctive normal form is *satisfiable* if there is a truth assignment that simultaneously makes each clause true and so makes the entire statement true under the usual interpretation of Boolean *and* ($\wedge$) and *or* ($\vee$). The *satisfiability problem* is specified as follows: given a set U of variables and a conjunctive normal form statement over U, is that statement satisfiable? An equivalent formulation is to consider the set of all conjunctive normal form statements that are satisfiable and to ask if a given statement is in that set.

Consider the following examples over the set $U = \{p, q, r, s\}$:

(i) $(p \vee q \vee r) \wedge (\bar{p} \vee \bar{q} \vee \bar{s}) \wedge (p \vee \bar{r} \vee s) \wedge (q \vee r \vee s)$;

(ii) $(p \vee \bar{p} \vee q) \wedge (p \vee r \vee \bar{r}) \vee (q \vee r \vee s \vee \bar{s})$;

(iii) $(p \vee q) \wedge (\bar{p} \vee q) \wedge (p \vee \bar{q}) \wedge (\bar{p} \vee \bar{q})$. Example (i) is satisfiable by means of the truth assignment $f(p) = T, f(q) = T, f(s) = F, f(r) = F$. Example (ii) is satisfiable under every truth assignment–it is a tautology. Example (iii) is not satisfiable under any truth assignment–it is a contradiction.

The problem of determining whether a given conjunctive normal form statement is or is not satisfiable is a "decision" problem: one must decide whether the given statement is a member of the set of all satisfiable statements.

How can one decide whether a conjunctive normal form statement over a set U is satisfiable? One method is to systematically construct the table of all truth assignments for U. As each such truth assignment is generated, check each clause in turn and determine whether that assignment makes every clause true; if so, then the truth assignment makes the entire statement true; if not, then the truth assignment makes the entire statement false. If U is finite, then there are only finitely many possible truth assignments for U and so this process eventually terminates.

If a statement in the propositional calculus has n variables, then there are 2^n possible truth assignments, any one of which might be the only satisfying assignment. This suggests that, if one wishes to solve this problem deterministically, then an exponential number of steps may be required since it may be necessary to enumerate the entire set of truth assignments.

For the satisfiability problem, what is desired is a program or algorithm that will correctly solve all instances of the problem, not just some set of special cases or instances where the variables are taken from a fixed finite set. In particular, the set $U = \{u_i | i = 1, 2, 3, \ldots\}$ of variables is taken to be countably infinite, although any instance of the satisfiability problem contains only finitely many occurrences of finitely many variables. Any algo-

rithm to decide whether a statement is satisfiable must identify the occurrences of the propositional variables and must determine whether the statement itself is in conjunctive normal form.

It is necessary to develop some further definitions in order to describe the notion of "*NP*-complete set" and the related "$P = ?NP$" question. Consider the following informal description of a class of programs that serve as "acceptors", i.e., only decision problems are solved: with each program a set of inputs is associated so that an input is "accepted" if the program decides that the input is in the given set and is "rejected" otherwise. Choose a finite set of operations on strings of symbols that includes, for control, some type of conditional branch (which allows looping), START operation, and HALT AND ACCEPT and HALT AND REJECT operations, where each operation can be performed in a bounded amount of time on a modern digital computer. A *program* is a finite flowchart of instructions from the set with one occurrence of the START operation and at least one occurrence of each type of HALT operation. On each input there is exactly one "computation" of the program on that input; the *computation of a program on an input* is a path through the flowchart which begins with the START operation and either ends with one of the HALT operations or does not end. The *length of a computation* of a program is the number of instructions executed, i.e., the length of the path which is finite or infinite depending on whether or not the computation halts. Since the input to a program is a string of symbols, we take the size of the input to be the length of the string. A program *operates within* time $T(n)$ if for every input string of length n, the computation on that input has length at most $T(n)$. A program *accepts* (*rejects*) an input if the computation of the program on that input halts with a HALT AND ACCEPT (resp., HALT AND REJECT) instruction. The set of inputs accepted by a program π is denoted by $L(\pi)$.

Let us return to the satisfiability problem. A program to solve this might have the following components:

PHASE 1. Identify the occurrences of the propositional variables and determine whether the input string is in conjunctive normal form.

PHASE 2. Generate a truth assignment to the variables that has not been previously generated. If this can be done successfully, transfer to Phase 3; otherwise, transfer to Phase 4.

PHASE 3. Determine whether the truth assignment generated in Phase 2 satisfies the input statement, i.e., makes the entire statement true. If so, transfer to a HALT AND ACCEPT instruction since the conjunctive normal form statement of the input is satisfiable; otherwise, transfer to Phase 2.

PHASE 4. Since all possible truth assignments to the variables occurring in the input have been generated and none of these assignments has made the input statement true, the statement must not be satisfiable. Hence, transfer to a HALT AND REJECT instruction.

Now consider a modification of the notion of program. Allow a flowchart to have a finite number of CHOOSE operations, that is, binary branches that allow the flow of control to go one of two ways depending on which branch is taken. When a CHOOSE operation is executed, a "guess" is made as to which branch to take, and if that particular CHOOSE operation is reached at a later

time in the computation, then the guess made at the later time is independent of the guess made at the earlier time. The result is a *nondeterministic* program. Thus on any single input string a nondeterministic program may have many different computations that can be represented by a binary-branching computation tree with the property that any path from root (labeled START) to a leaf (labeled HALT) represents a finite computation of that program on that input and every such finite computation is so represented. The *length of a computation* of a nondeterministic program is the length of the path representing that computation in the computation tree. A nondeterministic program *operates within time* $T(n)$ if for every input string of length n the computation tree of the program on that input has height at most $T(n)$.

A nondeterministic program π *accepts* an input x if in the computation tree of π on x there exists a path from the root to a leaf labeled HALT AND ACCEPT. Once again, the set of inputs accepted by a program π is denoted by $L(\pi)$.

When considering nondeterministic programs, it is only the accepting computations of that program on a given input (if any exist) that are considered. A nondeterministic program rejects an input only if it has no accepting computations on that input.

Consider a nondeterministic program for the satisfiability problem. Such a program may have the following components:

PHASE 1. Identify the variables and the correct form of the input just as before.

PHASE 2. Use a sequence of CHOOSE operations to nondeterministically "guess" a truth assignment to the propositional variables.

PHASE 3. Determine whether the truth assignment nondeterministically generated in Phase 2 satisfies the input statement, i.e., makes the entire statement true. If so, transfer to a HALT AND ACCEPT instruction since the conjunctive normal form statement of the input is satisfiable; otherwise, transfer to a HALT AND DO NOTHING instruction.

If the truth assignment "guessed" in Phase 2 does not satisfy the input statement, then this is simply a "bad guess". If the input is a satisfiable conjunctive normal form statement, then there will be some computation which makes a "correct guess" in Phase 2 and this computation will accept the input string.

Nondeterminism is a mathematical construct that cannot be implemented by real computers except by making the "guesses" systematically and "backtracking" in order to eventually search the entire computation tree seeking a path from the START root to a HALT AND ACCEPT leaf. Also, nondeterminism does not represent unbounded parallelism, i.e., the idea of evaluating every node at the level at each step. The "guess" made at a CHOOSE operation is not determined by any probabilistic notion. To say that a nondeterministic program accepts an input is simply to say that *there exists* a sequence of guesses at the various times when CHOOSE operations are executed that lead to a HALT AND ACCEPT operation.

A program with no occurrence of the CHOOSE operation is a "degenerate" nondeterministic program and is called *deterministic* since on

each input there is exactly one computation (and at each step in the computation there is at most one possible next operation to perform).

Clearly, if π_1 is a nondeterministic program that operates within time $T_1(n)$, then from π_1 one can construct a deterministic program π_2 such that $L(\pi_2) = L(\pi_1)$ and the function $T_2(n) = 2^{T_1(n)}$ bounds the running time of π_2: the program π_2 deterministically generates the entire computation tree of π_1 on the given input by systematically generating the table of guesses; in each computation π_1 can execute at most $T_1(n)$ binary CHOOSE operations and so the size of T_1's computation tree on an input string of length n is at most $2^{T_1(n)}$.

A deterministic or nondeterministic program *operates in polynomial time* if it operates within a time bound that is a constant multiple of n^k for some integer $k > 0$. The collection of all sets $L(\pi)$ such that π is a deterministic (resp., nondeterministic) program that operates in polynomial time is called the class of *languages accepted deterministically* (resp., *nondeterministically*) *in polynomial time* and is denoted by P (resp., NP).

From the definitions it follows that P is a subclass of NP but it is not known whether P is equal to NP.

The informal description of programs given here can be replaced with other formalisms for algorithms (e.g., Turing machines) and precisely the same classes P and NP result. That is, these classes are invariant under a variety of changes in the model of computation.

Now that the classes P and NP have been defined, we shall describe the notion of "NP-complete set".

A set L_1 is *polynomial-time reducible* to a set L_2, $L_1 \propto L_2$, if there is a function f such that for all x, $x \in L_1$ if and only if $f(x) \in L_2$ (i.e., $f^{-1}(L_2) = L_1$), and such that there is a program (with output) that operates in polynomial time and computes the values of f. This relation is reflexive and transitive, and preserves tractability: if $L_1 \propto L_2$ and $L_2 \in P$, then $L_1 \in P$. A set L_0 is *NP-complete* if $L_0 \in NP$ and for every $L \in NP$, $L \propto L_0$. Thus, an NP-complete problem is one of "maximal complexity" in NP. Further, there exists an NP-complete set that is in P if and only if every NP-complete set is in P if and only if $P = NP$.

There exist NP-complete sets. This fact is due to Stephen Cook, whose 1971 paper [3] marks the "official" beginning of the study of the question $P = ?NP$ and of NP-completeness. The first problem shown by Cook to be NP-complete is the satisfiability problem for conjunctive normal form statements in the propositional calculus. Clearly, the satisfiability problem is in NP: the nondeterministic program described above operates in polynomial time. Cook (using the Turing machine formalism to specify sets in NP) showed that every set in NP is polynomial-time reducible to the set of all conjunctive normal form statements that are satisfiable, and therefore the satisfiability problem is NP-complete.

It should be emphasized that what is desired is a program that will correctly solve *all* instances of the satisfiability problem. It has been shown that if one is willing to assume one of several distributions on the set of problem instances, then there are algorithms for which a polynomial bound can be obtained on the expected time complexity [4]. It is the worst-case

analysis, not the average-case analysis, of a program that comes into the notion of *NP*-completeness.

In addition to the satisfiability problem, Cook showed that several other problems are *NP*-complete. Further, Cook emphasized the significance of polynomial time reducibility and focused attention on the class (*NP*) of decision problems that can be solved nondeterministically in polynomial time. With the understanding that a problem is tractable only if it can be solved in polynomial time, the classification of a decision problem as *NP*-complete is evidence for that problem to be considered intractable.

Since Cook's results were announced in 1971, there has been a great deal of effort expended in studying the $P = ?NP$ question and in identifying *NP*-complete problems. In 1972 Richard Karp presented a collection of results proving that the decision-problem version of many well-known combinatorial problems, including the travelling salesman problem, are *NP*-complete. Karp's paper [5] caused attention to be drawn to further examples from such areas as graph theory, number theory, mathematical programming, automaton theory, covering and partitioning, and scheduling theory as well as to many problems of computer science related to compiler design, manipulation of data structures, operating systems, and data base management. Since that time certain decision problems have been proven to be intractable by showing that not only are they not in *P* but also they are not even in *NP*.

The question of whether $P = ?NP$ and thus whether the *NP*-complete problems are tractable is considered to be one of the foremost open problems of computer science and of modern applied mathematics. If *P* is equal to *NP*, then current methods for solving a wide range of computational problems will drastically change–many exponential search or unbounded backtracking procedures would be eliminated in favor of deterministic polynomial-time processes. Already the theory of *NP*-completeness has had significant impact on fields such as operations research.

The monograph reviewed here is appropriately subtitled: it is a guide to the theory of *NP*-completeness. Several basic *NP*-complete problems are discussed and techniques useful for establishing *NP*-completeness are studied as are subproblems of *NP*-complete problems and variations of *NP*-completeness. Of particular interest are the discussion of optimization problems and their interpretations as decision problems, approximation algorithms, and the application of *NP*-completeness to approximation problems. The extremely useful Appendix contains descriptions of over 300 problems (plus spinoffs of these problems) most of which are *NP*-complete or *NP*-hard.

This monograph is of value for those interested in the design and analysis of algorithms and in computational complexity. It can serve as a vehicle for anyone who wishes to learn about the subject of *NP*-completeness. Overall it is quite a good book, with only Chapter 7 (whose topics are several steps away from the authors' areas of expertise) being weak. Computer science needs more books like this one.

REFERENCES

1. A Cobham, *The intrinsic computational difficulty of functions*, Y. Bar-Hillel (ed.), (Proc. 1964 Internat. Congr.), Logic, Methodology and Philos. Sci., North-Holland, Amsterdam, 1965, pp. 24–30.

2. J. Edmunds, *Paths, trees and flowers*, Canad. J. Math. 17 (1965), 449–467.

3. S. Cook, *The complexity of theorem-proving procedures*, Proc. 3rd ACM Sympos. on Theory of Computing, J. Assoc. Comput. Mach. (1971), 151–158.

4. A. Goldberg, *On the complexity of the satisfiability problem*, Ph.D. dissertation, Courant Institute of Mathematical Sciences, 1979.

5. R. Karp, *Reducibility among combinatorial problems*, R. Miller and J. Thatcher (eds.), Complexity of Computer Computations, Plenum Press, New York, 1972, pp. 85–103.

RONALD V. BOOK

BULLETIN (New Series) OF THE
AMERICAN MATHEMATICAL SOCIETY
Volume 3, Number 2, September 1980

0002-9904/80/0000-0416/$01.75

The theory of Lie superalgebras; *an introduction*, by M. Scheunert, Lecture Notes in Math., vol. 716, Springer-Verlag, Berlin-Heidelberg-New York, vi + 271 pp.

A Lie superalgebra, or ($\mathbf{Z}_2$-) graded Lie algebra, is a vector space $\mathfrak{G} = \mathfrak{G}_0 \oplus \mathfrak{G}_1$ with a bilinear multiplication, $\langle\ ,\ \rangle$, satisfying the graded versions of the axioms for Lie algebras: if $X \in \mathfrak{G}_\alpha$, $Y \in \mathfrak{G}_\beta$, and $Z \in \mathfrak{G}_\gamma$ ($\alpha, \beta, \gamma \in \{0, 1\}$), then

(1) $\langle X, Y\rangle = (-1)^{\alpha\beta}[Y, X]$ ("graded antisymmetry");

(2) $(-1)^{\alpha\gamma}\langle X, \langle Y, Z\rangle\rangle + (-1)^{\beta\alpha}\langle Y, \langle Z, X\rangle\rangle + (-1)^{\gamma\beta}\langle Z, \langle X, Y\rangle\rangle = 0$ (the "graded Jacobi identity").

Note that $\mathfrak{G}_0$ is a Lie algebra (in the ordinary sense). In what follows, it will always be tacitly assumed that $\mathfrak{G}$ is finite dimensional and is defined over a field of characteristic 0.

The standard example of an ordinary Lie algebra is $gl(n)$, the space of all $n \times n$ matrices, with $[X, Y] = XY - YX$. (For instance, a representation of a Lie algebra is a homomorphism into $gl(n)$.) There is a corresponding standard example of a Lie superalgebra; it, too, is used to define representations. Let $V = V_0 \oplus V_1$ be a $\mathbf{Z}_2$-graded vector space. We define $pl(V) = pl(V)_0 \oplus pl(V)_1$, where

$$pl(V)_0 = \{V \to V,\ T(V_j) \subseteq V_j, j = 0, 1\};$$

$$pl(V)_1 = \{S: V \to V:\ S(V_j) \subseteq V_{1-j}, j = 0, 1\};$$

thus $pl(V)_0$ consists of the linear maps on V taking each distinguished subspace to itself, and $pl(V)_1$ consists of the linear maps on V taking each to the other. The multiplication is given as follows: if X, Y are each in $pl(V)_0$ or $pl(V)_1$, where

$$\langle X, Y\rangle = XY - YX \quad \text{if either } X \text{ or } Y \in pl(V)_0;$$

$$\langle X, Y\rangle = XY + YX \text{ if } X, Y \in pl(V)_1.$$

Thus the multiplication in $pl(V)$ consists of both commutators and anticommutators. It is this fact which explains the sudden interest in Lie superalgebras among physicists; they offer a mathematical framework for combining various symmetry theories. (It seems to be somewhere between unclear and dubious, however, whether the resulting supersymmetry theories do jibe with