# NEW DIRECTIONS IN TEACHING THE FUNDAMENTALS OF COMPUTER SCIENCE -- DISCRETE STRUCTURES AND COMPUTATIONAL ANALYSIS*

by

R. T. Yeh, D. I. Good, and D. R. Musser

March 1973 TR-13

*Presented at the Third SIGSE Symposium in Columbus, Ohio, 1973.

## Abstract

Basic teaching philosophy and format of presentation of two undergraduate courses in computer science are discussed. These two courses, discrete structures and computational analysis, are meant to provide the mathematical and "programmatical" foundations for undergraduate students in computer science and computing engineering. Detailed sequencing of course material and optional teaching plans are also presented here.

Key Words and Phrases: undergraduate computer science education, discrete structures, computational analysis, Curriculum 68.

CR Categories: 1.52, 5.24, 5.25, 5.30.

## I. Introduction

Since the publication of the Curriculum 68 report of the ACM Committee [CUR 68], undergraduate computer science programs have undergone substantial changes. Notably, many courses previously taught at the graduate level have shifted down to the undergraduate level; new areas have developed and consolidated but were not covered by the ACM report. These facts make it necessary to revise and update the Curriculum 68 report. An important factor which could help the revision is the actual experience gained through the development of many new undergraduate computer science programs since the publication of Curriculum 68 report five years ago. More specifically, most recommendations of the ACM report contain just the subject areas to be taught. The actual teaching experiences of these courses have formulated certain basic philosophies and formats of presentation. It is helpful, in our opinion, to compare the merits of these philosophies and formats of presentation.

In this paper, we shall discuss the basic philosophies and formats of presentation of two undergraduate computer science courses -- Introduction to Discrete Structures, and Computational Analysis. The former coincides with the B3 course of the curriculum 68 report; the latter is a new course not covered by the ACM report.

## II. Motivation for These Two Courses

The need for an undergraduate course in discrete structures was clearly recognized by the ACM curriculum committee on computer science [CUR 68]. However, we would like to point out that the required mathematical training of most college curricula in computer science and computing

engineering is still largely confined to infinitesimal analysis and calculus, and each upper-division course is supposed to develop its own formal prerequisites. This policy clearly has two weaknesses: it is inefficient, because duplications of efforts are inevitable and precious time is subtracted from specific subjects, and it is ineffective, because understandably the presentation of background material for each individual course is kept to a minimum. Thus, the mathematical background of a majority of students is less than desirable; a clear effect of this piecemeal exposure to the pertinent mathematical material.

It is almost superfluous to point out that one of the prime objectives of undergraduate computer science education is for the students to learn to write "good" programs. This objective is becoming more important as the complexity and size of software systems grow. The training for programmers of the next generation should be that not only can they write programs, but must write programs that people other than themselves can understand. To be more specific, a programmer should be able to design programs systematically, and should be able to demonstrate that a program he writes is correct with respect to its intended goal. Furthermore, he should have some ideas about the behavior of the program in terms of, say, time and memory requirements. Presently, this background training of a good programmer is only obtained, and usually not effectively, after a student has gone through all the undergraduate programming courses. Thus, we find motivation to develop an undergraduate course on computational analysis[1] which provides students a working knowledge of some of the basic

---

[1] We would like to remark here that in proposing a course on computational analysis, we take the stand that computer programming belongs to the domain of science as well as art.

tools that can be used to systematically design computer programs and to analyze the computation performed by a computer program in much the same way as we abstract the basic mathematical background in the discrete structure course.

## III. Discussion of the Course on Discrete Structures

### A. Philosophy and Format of Presentation.

In order for students to have a clear understanding of various structures, the philosophy of our presentation is to proceed from general to specific. By starting with the simplest structures, that is, sets, one can add properties to reach more complicated structures. In this framework, graphs are viewed as relations on a set; lattices as a class of partially ordered sets, boolean algebras as a class of lattices, and so on. We feel that this approach can best lead students to appreciate how new properties add structure to formal systems.

The format of presentation is one in which we approach each new concept by means of real world examples, highlighting in simple language the important features of this concept in order to build up motivation so that formal statements and definitions come much easier.

### B. Contents of the Course.

Topics which we feel should be included in this course can be divided into three major subject areas, namely: algebraic structures, combinatorics, and basic computation theory. Before discussing any of the subject areas, we shall familiarize the students with the terminology and techniques of mathematical discourse. Although we are aware of the fact that one cannot formally teach logical deduction, we find definite merit in presenting

and discussing the meaning of various terms and phrases, such as axiom, lemma, theorem, proof by contradiction, inductive definitions, etc.

(I). Algebraic Structures

1. Sets and Binary Relations

a) Theory: We introduce sets and their main properties. Structure is added to sets via the concept of binary relations, whose properties are analyzed in detail. Binary relations are then specialized to functions and relations on a set: the latter lead to the important classes of compatibility and equivalence relations. We emphasize here that no mention should be made at this point of set algebra (a difference of philosophy from Curriculum 68 recommendations) although set union and intersection are defined here for useful didactic purposes. We contend that the full appreciation of the structural richness of set algebra can be reached only at a much later stage.

b) Applications: Finite state machines; document and information retrieval, Russell's paradox; complexity of representation of binary relations and functions.

2. Graphs

a) Theory: The graphical representations of binary relations are studied with the objective of investigating their topological properties. Directed and undirected graphs are discussed, and concepts relating to connectedness--such as paths, chains, cut-sets and articulation points--are studied. Graphs are then specialized to trees, and further specializations, such

as rooted trees and oriented rooted trees, are studied for their great usefulness as models. Path problems concerning eulerian and hamiltonian circuits and planarity and map coloration are also discussed.

b) Applications: State diagrams of finite state machines; scheduling by PERT-CPU; linear netowrk analysis; representation of data structures in computers; dominating and independent sets; decomposition of graphs into planar components.

3. Semigroups and Groups

a) Theory: Binary operations are at first defined, and by means of the concepts of associativity and inverses the students are led to appreciate the structural enrichment in passing from semigroups to groups. Rings and their substructures are also mentioned briefly. The important notions of isomorphism and homomorphism are analyzed, in this order for clear pedagogical reasons. Finally, the various concepts are synthesized in a coherent and compact view through the unifying notion of universal algebra.

b) Applications: The associated monoid of a finite state machine; permutations, communications and error-correcting codes; infix, prefix, and postfix notations for expressions; the algebra of words, phrase-structure grammars.

4. Lattices

a) Theory: The concept of partial ordering is refreshed at the start and its properties are brought to evidence. This leads to the study of posets; from here the notion of lattice

emerges naturally. The general structure of lattices is investigated and special attention is given to distributivity. Subsequently, the representation of lattices in terms of join-irreducible elements is presented in a very simple way. Finally, an important class of lattices, that is, partition lattices, are considered in connection with their relevance to the structural description of finite state machines.

b) Applications: Structural theory of finite state machines.

5. Boolean Algebra

a) Theory: Set algebra is first discussed, and the power set of a set is viewed as a special case of a finite distributive lattice. The intuitive notion of complement in power sets is made abstract in the context of boolean algebras. It is then shown that boolean algebras are isomorphic to power sets (Stone theorem), and the general representation problem is tackled. Subsequently, the structural and the manipulative aspects of an algebraic system are clearly separated, and the calculus of boolean algebras is introduced as the application of a set of identities on free boolean algebras. The study of boolean functions precedes the discussion of important boolean algebras, such as propositional calculus switching algebras.

b) Applications: Design of switching and sequential networks, algebra of logic.

Prerequisite organization of material in this section is illustrated by means of a directed graph given in Figure 1. Each box in the diagram represents a major concept and the arcs describe how a concept is a prerequisite of some other concepts.

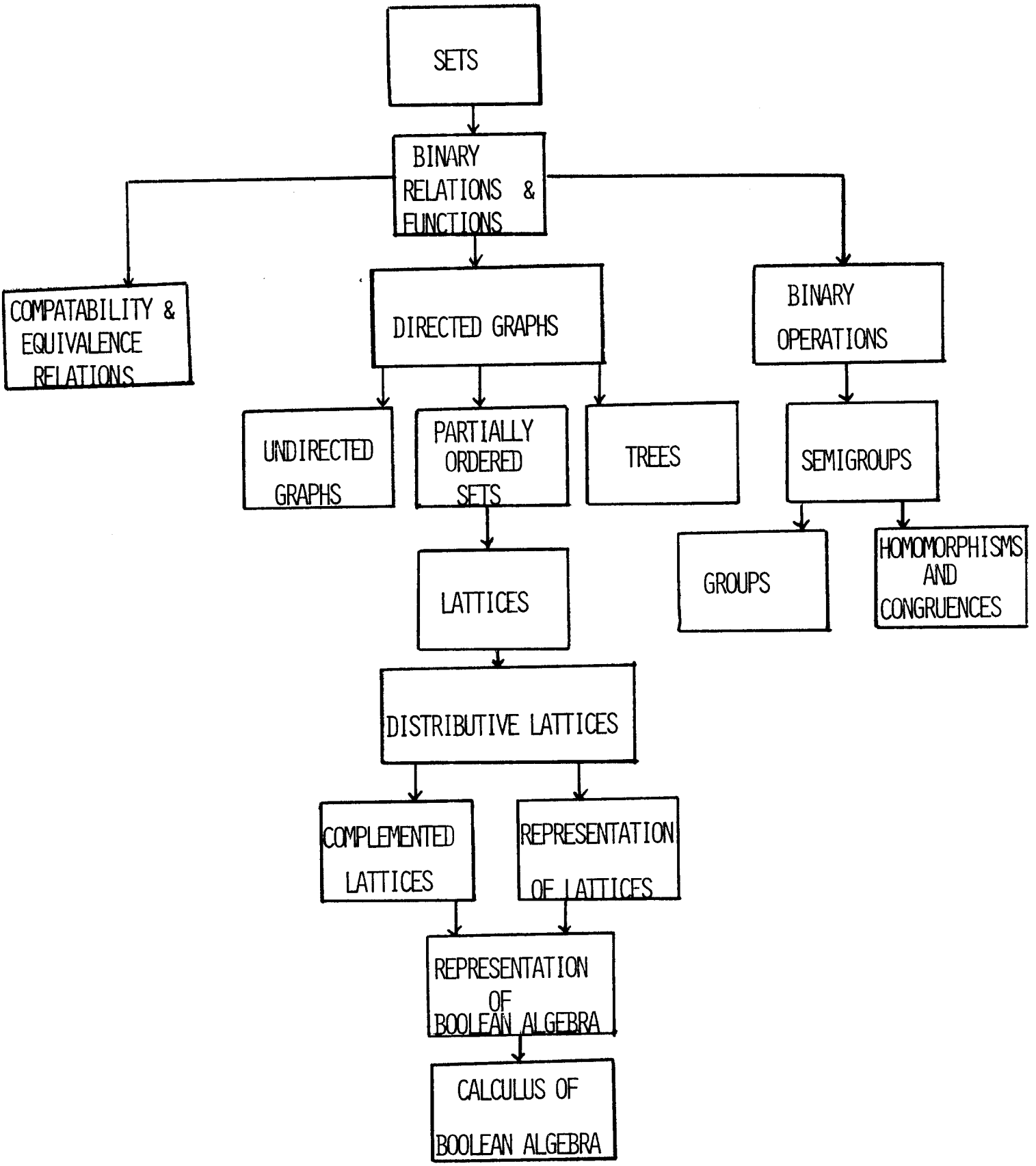


FIGURE 1. Core Material of the Algebraic Structures Portion of Discrete Structures Course.

(II). Combinatorics

1. Number Systems

a) Theory: Basic properties of integers are investigated. Prime numbers, congruences and residue numbers system are studied in more detail.

b) Applications: Computer arithmetics.

2. Counting Techniques

a) Theory: Elements of combinatorics are presented that should enable the reader to solve simple counting problems. Permutations, combinations, distributions, the principle of inclusion and exclusion, and enumeration by recursion are presented along with many simple, interesting applications. A simple introduction to Polya's theory is also included.

b) Applications: Bounds for error-correcting codes; complexity of travelling salesman problem, analysis of algorithms (Euclidean algorithm, sorting and searching algorithms).

3. Discrete Probability

a) Theory: Basic concepts of discrete probability, conditional probability, simple and compound experiments, and drawing with and without replacement are presented.

b) Applications: System Modelling.

Again, sequencing of the course material in this section is given as a directed graph in Figure 2.

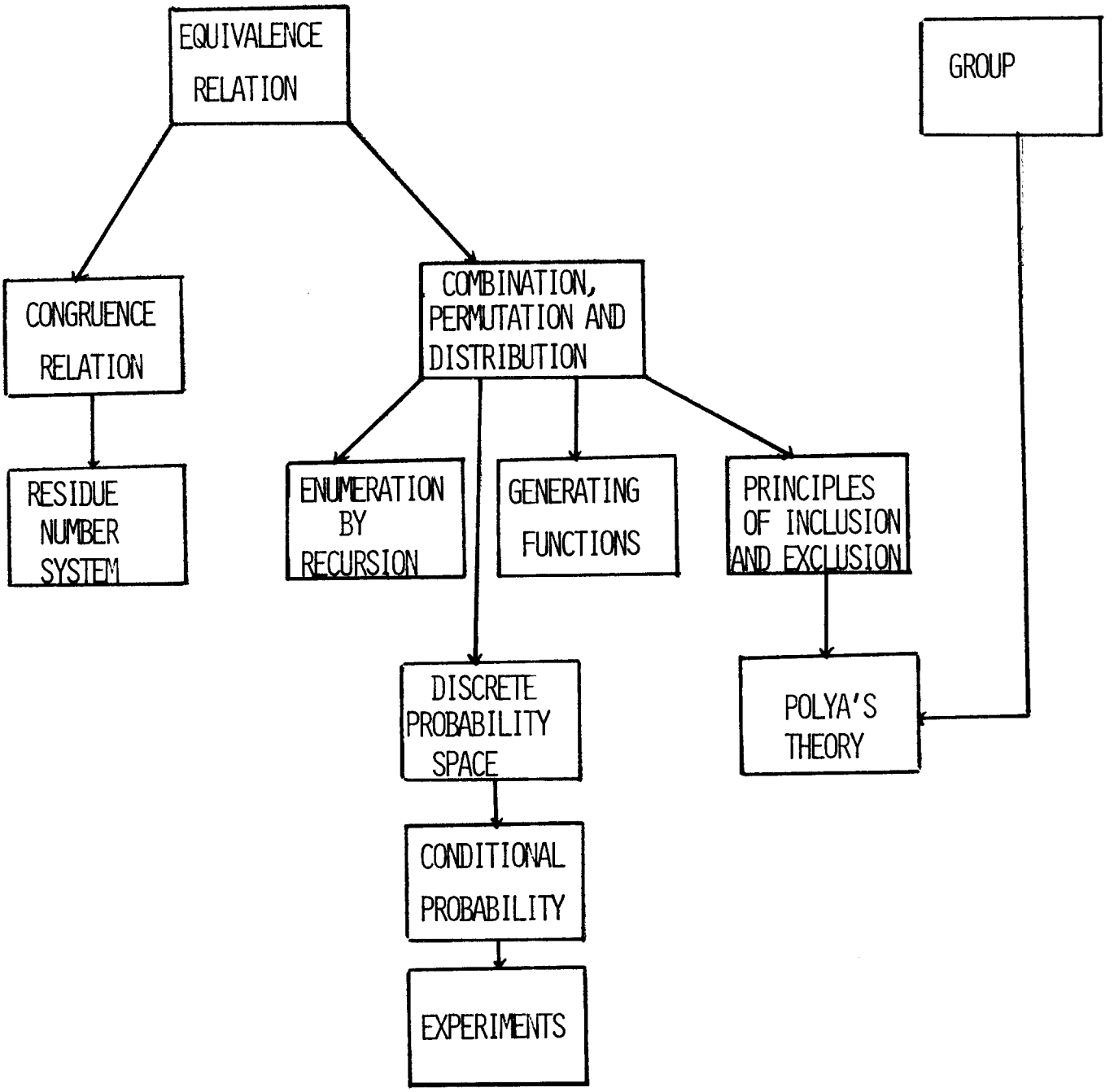


FIGURE 2. Core Material of the Combinatorics portion of Discrete Structures Course.

(III). Computation Theory

a) Theory: We begin with an informal discussion of the concept of algorithm, and through the notion of effective procedure we arrive at Turing machines. Turing machines are discussed in some detail and their halting problem is used as an eye-opener into the far-reaching field of algorithmic insolvability.

b) Applications: Finite state recognizers; formal grammars; post canonical systems.

Sequencing of material contained in this section is given by the directed graph in Figure 3.

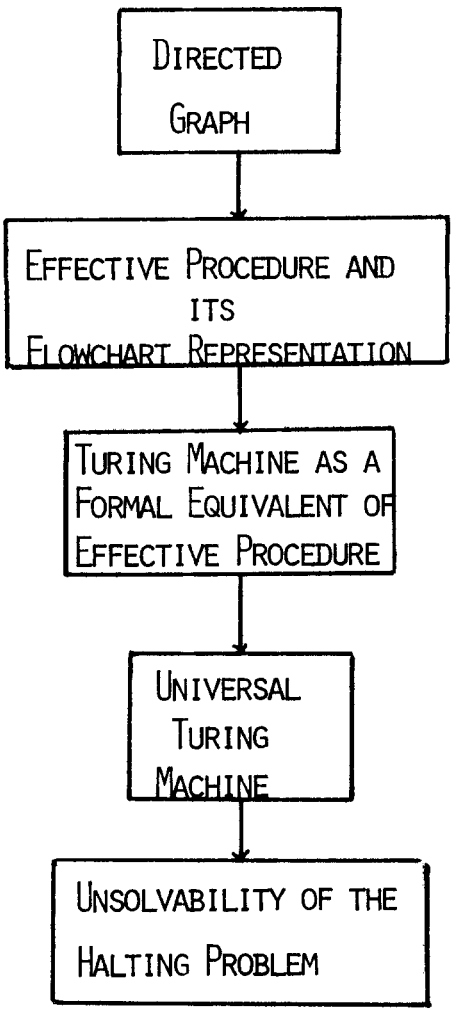


FIGURE 3. Core Material of the Computation Theory Portion of Discrete Structures Course.

C. Alternative Teaching Plans

Teaching plans and prerequisite organization of this course are illustrated by means of a directed graph. Figures 4 and 5 present the prerequisite organization of the course. Each box in the graph represents a major concept and the arcs describe how a concept is a prerequisite of some other concepts. In Figure 4, we indicate what we feel should be the essential core of a course in discrete structures: sets, relations, graphs, lattices, boolean algebras, semigroups, groups and basic counting techniques. Depending upon available time and the preference of the instructor, this basic material should be supplemented by additional topics mentioned in the previous section. In Figure 5, we illustrate three of these optional supplementary programs, such as algebraic structures, topics in boolean algebras, combinatorics, and algorithms.

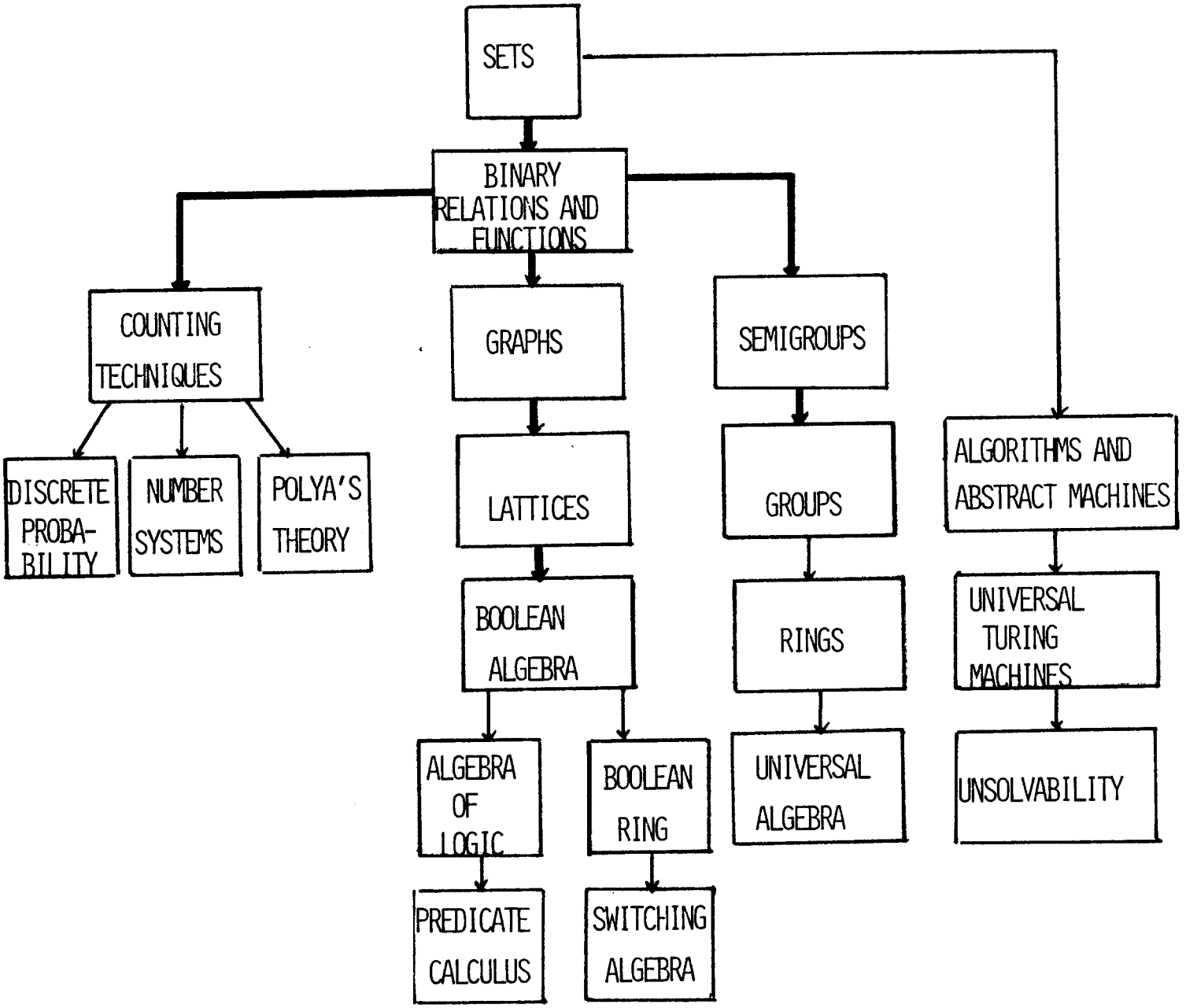


FIGURE 4. Core Material of the Discrete Structures Course.

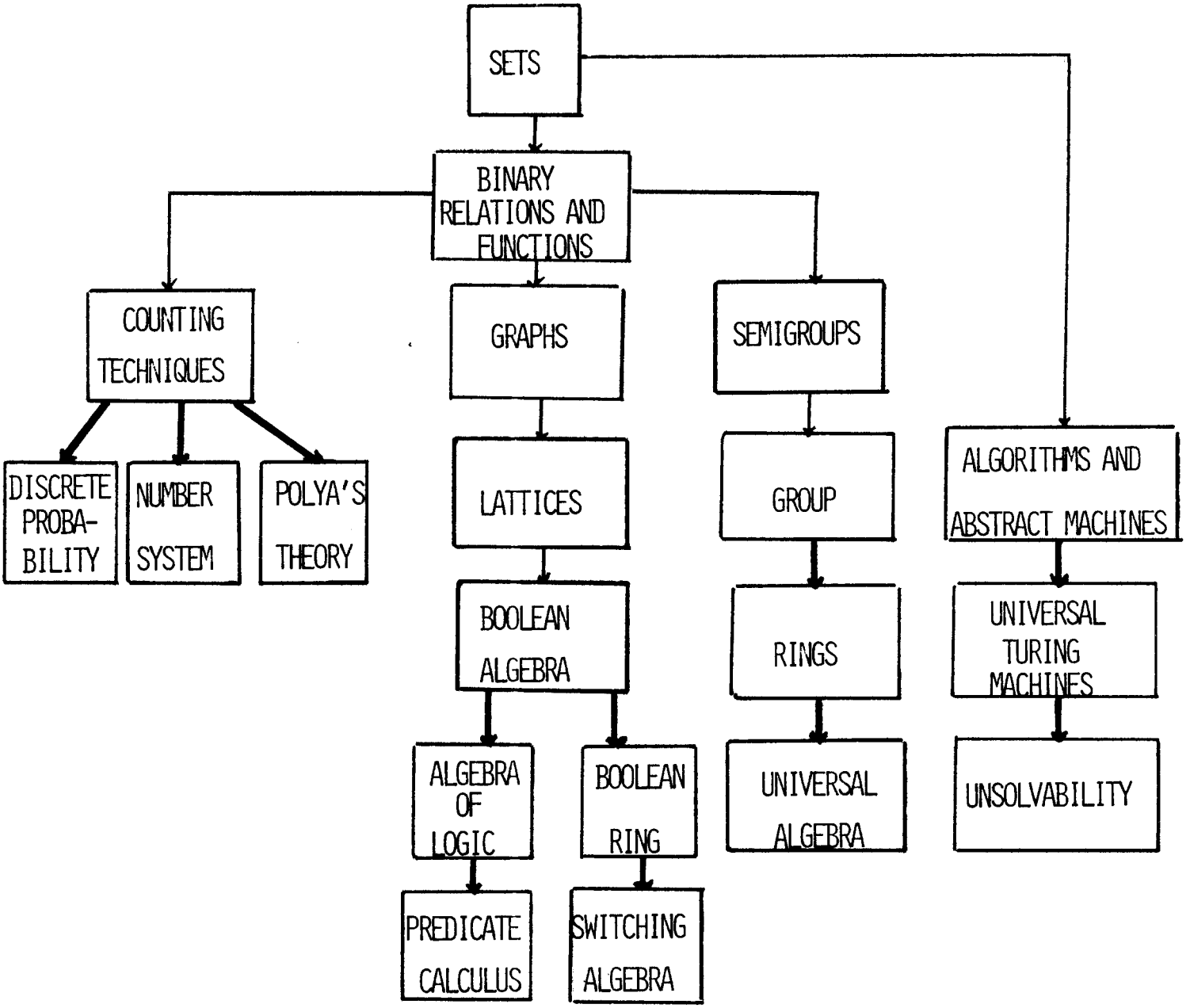


FIGURE 5. Optional Plans for Discrete Structures Course.

D. Bibliography

Currently there are three text books available for this course.

1. Birkhoff, G., and R. Bartee, Modern Applied Algebra, McGraw-Hill Book Company, 1970.

This book covers most of the topics mentioned in C and has some very extensive applications. However, while this book is definitely a good reference book, the level of the book exceeds the majority of undergraduate students in computer science and engineering.

2. Berztiss, A. T., Data Structures: Theory and Practice, Academic Press, 1971.

This book is rich in applications. However, the presentation is too compact for average undergraduate students. Again, it is good as a reference book but not suitable as an undergraduate text for computer science or engineering students.

3. Preparata, F. P., and R. T. Yeh, Introduction to Discrete Structure, to be published by Addison-Wesley, May, 1973.

This book covers all the topics of the B3 course in the Curriculum 68 report. The basic philosophy and sequencing of materials covered coincide with that discussed in III. A above.

## IV. Discussion of Course on Computational Analysis

The main prupose of an undergraduate course in computational analysis is to introduce the student to some basic tools that can be used to analyze the behavior of computational processes, be they specified by actual computer programs, Turing machines, or abstract mathematical operations such as are often encountered in numerical methods. The intent of this course is not to be exhaustive, but instead to present a few broadly applicable methods that can be used to analyze some of the more basic properties of computations and ultimately lead to the design of better quality programs. This particular course is designed with primary focus on just two types of properties, correctness and computing times. Granted, there are certainly other interesting types of properties, but the question of "Does the computation do what it is supposed to?" and "How long does it take?" clearly are of fundamental importance. By having available some basic tools for answering these questions about the programs he designs, the student should be able to produce programs of a high degree of reliability and also be able to choose rationally the more efficient of programs that perform the same computation. By collecting these tools in a course at as low a level as possible, the student will be able to use them in programming projects in later courses and thus gain valuable experience in the production of high quality programs.

### A. Suggested Place in the Curriculum.

This course is designed on the basis of two prerequisites, Curriculum 68 courses B2, Computers and Programming, and B3, Introduction to Discrete Structures. B2 provides the necessary level of knowledge of

what a computation is, and also the exposure to machine language in B2 leads to a natural abstraction to the idea of a state vector. This familiarity of machine language also helps students to understand the inherent difficulties of analyzing program performance in terms of computing time required. This gives motivation to analyze programs, independent of particular machines, in terms of maximum required computing time. Course B3 provides the necessary mathematical tools and maturity for understanding mathematical induction and constructing the proofs that are required in analyzing computations.

The intent of this course is to provide a "programmatical" foundation for computer science undergraduates in much the same way that the discrete structures course provides a mathematical foundation. This not only relieves the more advanced courses from the responsibility of teaching these foundations but also gives the student the opportunity to gain familiarity with many varied applications of these fundamental concepts. Thus in our view, the relation of the course to Curriculum 68 should be as shown in Figure 6.

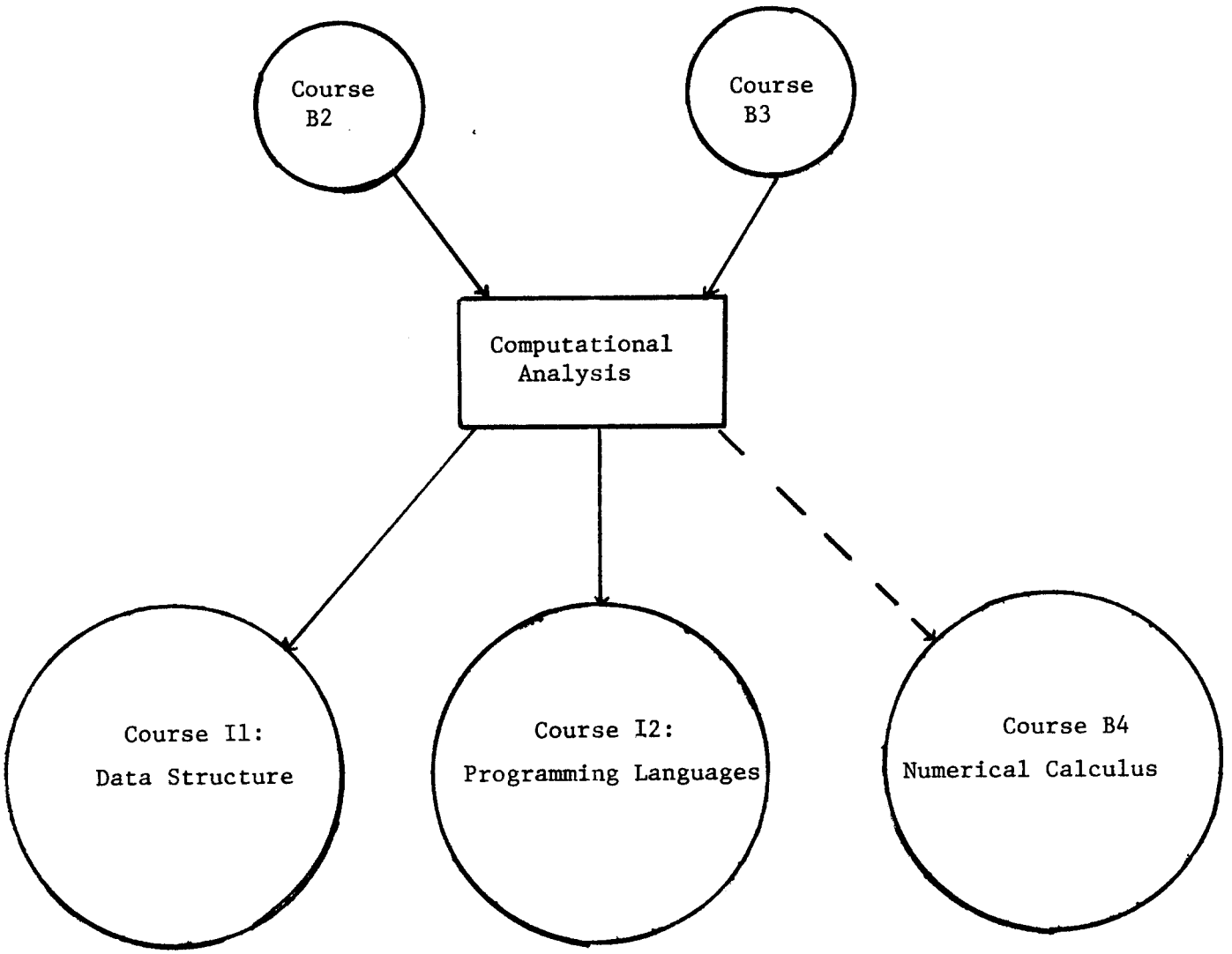


FIGURE 6. Relationship between the Proposed Course on Computational Analysis and Courses in the Curriculum 68 Report.

B. Contents of the Course.

The core of the course centers around three major areas; a computational model that will serve as a uniform vehicle for presenting the analysis methods, the inductive assertion method for proving computational correctness, and various algebraic and combinatoric methods for analyzing computing times. Before discussing these areas, we cannot overemphasize our belief that the student should have had extensive exposure to, and practice with, the ideas of structured programming and top down design of programs in courses B1 and B2. The top down design of structured programs not only has the advantages of inducing within the student a disciplined method of thinking through problems, and of resulting in significantly more reliable, understandable and easily modifiable programs, but also the structuring produces significant simplifying effects on the types of analysis to be considered in this course. Therefore, if these topics have not been included in B1 and B2, we believe that it is extremely important that they be presented in the computational analysis course so that the student know how to construct programs that are susceptible to analysis.

1. Computational Model

The necessity of presenting a computational model is due to the simple fact that before one can analyze the behavior of a computation, we must have a precise way of defining that computation, and we believe an appropriate choice of a model is crucial. The model should be as realistic as possible, yet be simple enough to be susceptable to mathematical analysis. The model proposed is that of a flowchart which specifies operations on a state vector. The use of flowcharts seems natural

as they should already be familiar to the students and they facilitate analysis of many properties of computations.

In this model a computation is said to terminate (with a given input) if the number of steps (visits to flowchart boxes) is finite and a computational method which terminates for all inputs is called an algorithm. The model does not require that the operations which appear in the flowchart boxes be from a pre-specified set of basic operations. This leaves open the question of effectiveness of the operations and proofs of effectiveness need not necessarily be considered in the course; rather the operations which are chosen in particular algorithms should be a) simple enough that their effectiveness should be intuitively obvious, or else b) operations specified by other algorithms.

The states of the computation could also be represented in many ways, but the simple idea of vectors of values seems most appropriate here. Again, the student is already familiar with various concrete forms of this idea; e.g. pencil and paper records of execution from hand simulations, or the idea of memory divided into cells.

After an informal discussion of the model, a mathematically rigorous formal definition can be given, using the concepts of set theory, functions and directed graphs which the student brings with him from the discrete structure course. It is then possible to discuss various properties of computations both intuitively and rigorously in terms of this model.

2. Computational Correctness

The inductive assertion method of proving program correctness is easily presented in terms of the flowchart and state vector model. The

statement of program correctness is phrased in terms of relations on the initial and final state vectors, and the inductive assertions themselves are simply properties of state vectors attained at intermediate points in the flowchart. The presentation can begin by introducing the idea of using a verification condition to prove relations between state vectors at the beginning and end of a particular path through the flowchart. From there one can move on to the complete analysis of loop-free programs by considering all possible paths through the flowchart. The consideration of programs with single loops provides a clear picture of the implicit mathematical induction on which the entire inductive assertion method is based. After the student understands the applications of the method to programs with arbitrary looping structures, one can then show how the method can be extended to include proofs of termination by constructing functions that map the state vectors into well ordered sets.

Being familiar with methods for proving correctness and termination of particular computations, it is natural to move on to consider properties of program schemata. With just a basic introduction to the idea of what a schemata is, useful termination and equivalence properties can be derived and presented.

3. Computing Time Analysis

The major emphasis under the topic of computing time and memory analysis of algorithms should be on analysis of particular algorithms rather than analysis of classes of algorithms, since the former type of analysis better fits the interests and needs of undergraduate students. Furthermore, more attention can be given to computing time than to memory requirements, since in many important applications allocation of memory is static and hence

the analysis is quite simple. Of course, the tradeoffs between efficient use of memory and computing time should be emphasized in at least some of the example algorithms considered.

Computing time analysis is probably best introduced via example algorithms which are susceptable to "exact analysis." By this is meant that the computing time can be expressed as a simple function of the input to the algorithm. Various algorithms to compute $x^n$ might be used for these examples. By expressing the algorithms with flowcharts, it is easy to introduce the basic idea of counting operation executions by attaching "counting functions" to each arc of the flowchart. These functions give the number of traversals of the arc as a function of the input and can be related and reduced in number using "Kirchoff's law" (of conservation of flow) [KNU68].

In cases in which each operation takes a fixed amount of time (as when the operations are machine language instructions), determination of the counting functions essentially determines the total computing time function, as could be illustrated with machine language versions of the example algorithms. By assuming fixed but unspecified operation times, machine-independent expressions of computing time functions could then be obtained (in terms of the fixed times as parameters). To simplify these expressions, traditional order notation could be introduced. However a superior alternative is provided by the concept of dominance, as defined by Collins [COL 71]. It is easy at this point to develop a simple algebra of dominance relations which greatly facilitates the simplification and comparison of computing time functions.

If some of the steps of an algorithm contain operations which require a varying amount of time (dependent on the current state vector),

then the analysis is more complex, since counting arc traversals no longer suffices. Good illustrations of this point might be obtained by considering algorithms for $x^n$ -- when the time for multiplication is fixed, the computing time is codominate with the number of multiplications, so that algorithms requiring only $\log_2 n$ multiplications are much faster than simpler algorithms requiring $n$ multiplications, but the opposite may be true when the time for multiplication depends on the size of the operands, as when $x$ is a polynomial [HEI72].

More complex algorithms could be introduced at this point in order to motivate the need for additional mathematical tools. Consideration of sorting and searching algorithms, for example, would motivate a study of basic counting principles, permutations, combinations and other combinational concepts. This study should actually be partly a review of material introduced in the discrete structures course. [KNU68,§1.2] gives a good discussion of the main combinatorial tools needed for computational analysis (but mostly without prior motivation by example algorithms). Additional material and exercises could be drawn from an introductory combinatorial mathematics text such as [LIU72]. The topics of generating functions, recurrence relations, and the principle of inclusion and exclusion should be included.

These tools, along with dominance relation algebra, suffice for analysis of the maximum computing time of many algorithms. Consideration of average computing time requires additional concepts from probability theory and additional practice with application of recurrence relations and generating functions to analysis of probabilities. The basic material is covered in [KNU68,§1.2].

Prerequisite organization of this course is given as a directed graph in Figure 7. Each box in the graph represents a major concept and the arcs describe how a concept is a prerequisite of some other concepts.

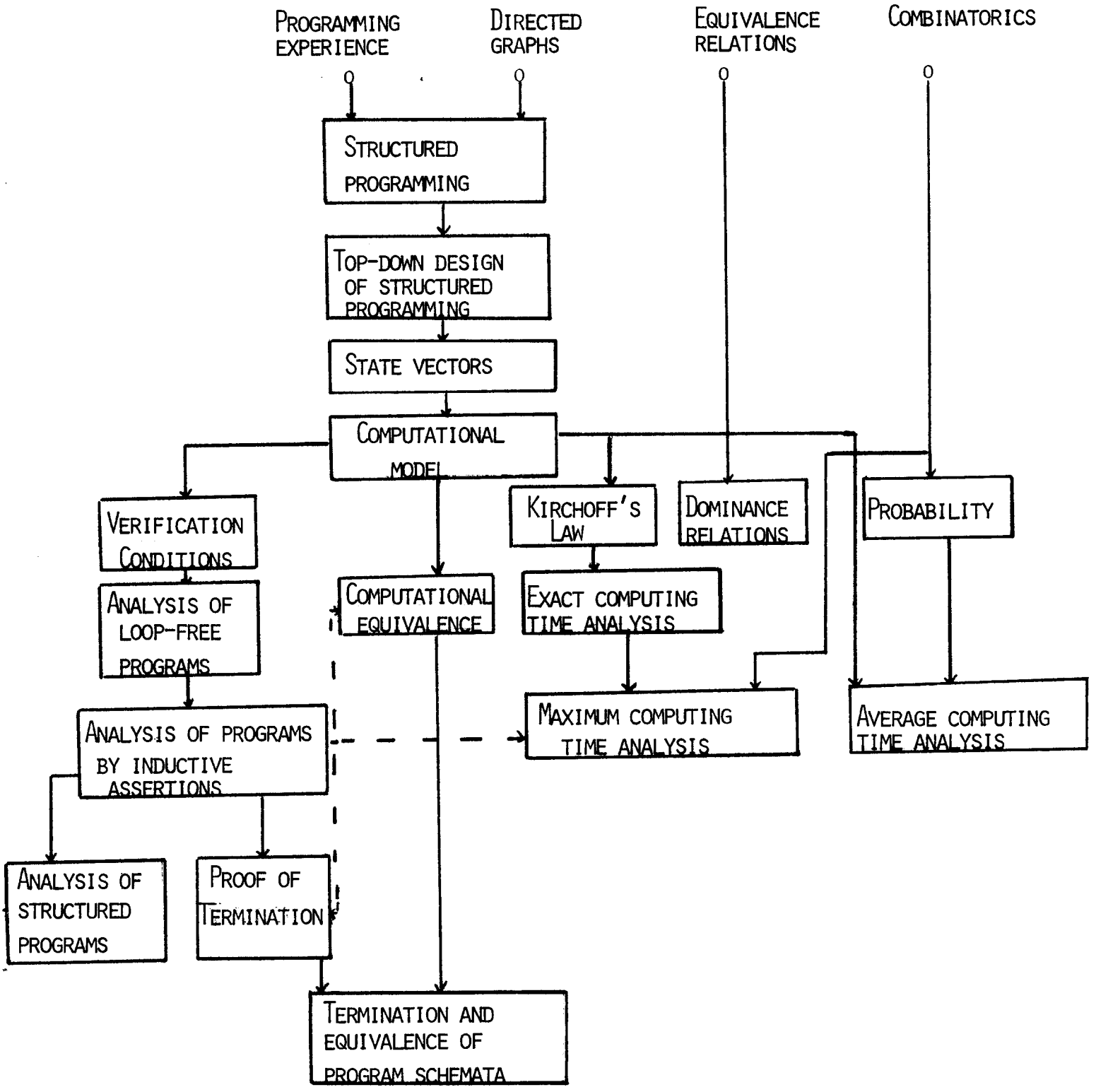


FIGURE 7. Core Material of Computational Analysis Course.

C. Bibliography

There is no suitable text for the course, but the following books and papers contain pertinent material. These are intended primarily as source material for instructors rather than as reading assignments for the students.

1. Structured Programming and Top Down Design

Baker, F. T., "Chief Programmer Team Management of Production Programming," IBM Systems Journal, 11, 1 (1972).

Dijkstra, E. W., "Notes on Structured Programming," TH-Report 70-WSK-03, Dept. of Mathematics, Technological University Eindhoven, The Netherlands, (April 1970).

Henderson, P. and Snowdon, R., "An Experiment in Structured Programming," BIT 12 (1972).

Mills, H. D., "Mathematical Foundations for Structured Programming," IBM Report FSC 72-6012, (May 1972).

Mills, H.D., "Top Down Programming in Large Systems," in Debugging Techniques in Large Systems, R. Rustin (Ed.), Prentice Hall, (1971).

Wirth, N., "Program Development by Stepwise Refinement," Comm. ACM 14, 4, (April 1971).

2. Computational Correctness

Burstall, R. M., "Proving Properties of Programs by Structural Induction," Computer Journal 12, 1 (Feb. 1969).

Elspas, B., Levitt, K., Waldinger, R., Waksam, A., "An Assessment of Techniques for Proving Program Correctness," Computing Surveys 4, 2 (June 1972).

Good, D. I., "Developing Correct Software," in Proc. of First Texas Symposium on Computer Systems (June 1972).

Hoare, C. A. R., "An Axiomatic Basis for Computer Programming," Comm. ACM 12, 10 (Oct. 1969).

Hoare, C. A. R., "Procedures and Parameters: An Axiomatic Approach, in Symposium on Semantics of Algorithmic Languages, E. Engeler (Ed.), Springer-Verlag (1970).

Hoare, C. A. R., "Proof of a Program: FIND," Comm. ACM 14, 1 (Jan. 1971).

Hull, T. E., Enright, W. H., Sedgwick, A. E., "The Correctness of Numerical Algorithms," Proc. of Conference on Proving Assertions about Programs, ACM (Jan. 1972).

King, J.C., "Proving Programs to be Correct," IEEE Transactions on Computers (Nov. 1971).

Lee, J. A. N.,"The Definition and Validation of the Radix Sorting Technique," Proc. of Conf. on Proving Assertions about Programs, ACM (Jan. 1972).

London, R. L., "Bibliography on Proving the Correctness of Computer Programs," in Machine Intelligence 5, B. Meltzer and D. Michie (Eds.) American Elsevier Publ. Co., New York, (1970).

London, R. L., "Proofs of Algorithms: A New Kind of Certification (Certification of Algorithm 245, Treesort 3)," Comm. ACM 13, 6 (June 1970).

London, R. L., "The Current State of Proving Programs Correct," in Proc. of ACM Annual Conference, ACM (Aug. 1972).

Manna, Z., "Properties of Programs and the First-Order Predicate Calculus," Journ. ACM 16, 2 (April 1969).

Manna, Z., Ness, S., Vuillen, J., "Inductive Methods for Proving Properties of Programs." In Proc. of Conf. on Proving Assertions about Programs, ACM (Jan. 1972).

Maurer, W. D., "State Vectors: A Preliminary Survey," Memo. No. ERL-M347, Univ. of California, Berkeley (Aug. 1972).

Naur, P., "Proof of Algorithms by General Snapshots," BIT 6, (1966).

3. Computing Time Analysis

Brown, W. S., "On Euclid's Algorithm and the Computation of Polynomial Greatest Common Divisors," Journ. ACM 18, 4 (Oct. 1971), pp. 478-504.

Collins, G. E., "The Calculation of Multivariate Polynomial Resultants," Journ. ACM 18, 4 (Oct. 1971), pp. 515-532.

Knuth, D.E., The Art of Computer Programming, Addison-Wesley, Vol. 1 (1968); Vol. 2 (1969); Vol. 3 (1972).

Knuth, D. E., "Mathematical Analysis of Algorithms," IFIP Congress Proceedings I, 135-143, (1972).

## VI. Concluding Remarks

In the previous sections, we have outlined basic philosophies, contents and formats of presentation of two undergraduate level courses which we believe are fundamental to any computer science and computing engineering program. For the discrete structures course, basically we agree with the contents of the B3 course recommended in Curriculum 68 report. However, because of our belief that a student can best learn to appreciate structures by building them up, from simple structures to more complex ones we differ slightly from the Curriculum 68 recommendations in our sequencing of the material to be presented. For example, in our presentation, graphs, considered as binary relations on a set, would come right after sets since it is only slightly richer in structure than sets. The course on computational analysis is a new course not covered by the Curriculum 68 report. This course, as pointed out at the beginning of this paper, is intended to provide a "programmatical" foundation of computer science undergraduates the same way discrete structure course provides a mathematical foundation for computer science undergraduate students.

## VII. Acknowledgment

The authors are indebted to Professor William Henneman for valuable suggestions and comments in the preparation of this paper.

VIII. References

[COL71] Collins, G.E., "The Calculation of Multivariate Polynomial Resultants," Journ. ACM 18, 4 (Oct. 1971), pp. 515-532.

[CURR68] Curriculum 68 Report of ACM Committee, Comm. ACM, Vol. 11, No. 3 (March 1968).

[HEI72] Heindel, L. E., "Computation of Powers of Multivariate Polynomials Over the Integers," JCSS 6, 1-8 (1972).

[KNU68] Knuth, D.E., The Art of Computer Programming, Vol. I: Fundamental Algorithms, Addison-Wesley Publishing Co., Reading, Mass. (1968).

[LIU68] Liu, C. L., Introduction to Combinatorial Mathematics, McGraw-Hill, (1968).