

INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

U·M·I

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313.761-4700 800.521-0600



Order Number 9121171

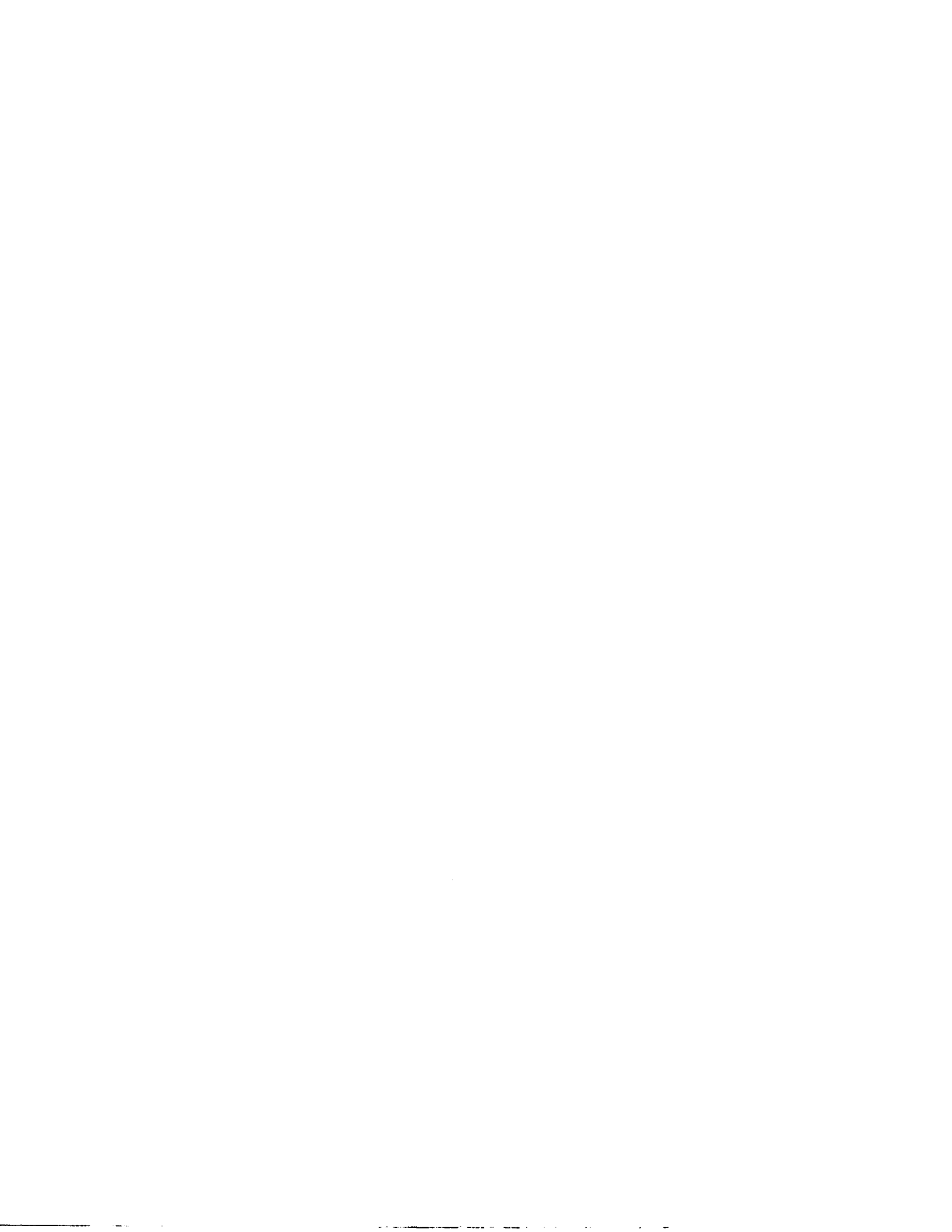
**Graph theory in an undergraduate lower-division computer
science algorithms course**

Courtney, Mary Fleming, Ed.D.

Columbia University Teachers College, 1991

Copyright ©1991 by Courtney, Mary Fleming. All rights reserved.

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106



GRAPH THEORY IN AN UNDERGRADUATE LOWER-DIVISION
COMPUTER SCIENCE ALGORITHMS COURSE

by

Mary Fleming Courtney

Dissertation Committee:

Professor Henry O. Pollak, Sponsor
Professor Bruce Vogeli

Approved
by the Committee on the Degree of Doctor of Education
Date JAN 28 1991

Submitted in partial fulfillment of the
requirements for the Degree of Doctor of Education in
College Teaching of an Academic Subject in
Teachers College, Columbia University

1991

c Copyright Mary Fleming Courtney 1991
All Rights Reserved

ABSTRACT

GRAPH THEORY IN AN UNDERGRADUATE LOWER-DIVISION COMPUTER SCIENCE ALGORITHMS COURSE

Mary Fleming Courtney

Since the field of computer science has grown dramatically, a need exists to design good curricula. Much has been written about teaching programming courses, but less discussion has taken place concerning algorithms courses. In preparing to teach an algorithm, the teacher must be concerned with motivating the students, eliciting an algorithm from the students, and deciding on a good data structure.

The field of graph theory offers fascinating algorithms to teach. Unfortunately, it is only recently that the study of graph theory and discrete mathematics has been established in college curricula. Therefore, many of the faculty in computer science have modest backgrounds in this field. With this in mind, the author wrote a sourcebook on graph theory to aid faculty in preparing their lectures for a computer science algorithms course. Included in the sourcebook are historical anecdotes, problems for classwork and homework, different types of mathematical proofs, techniques for teaching algorithms, and a Pascal program demonstrating the algorithms. The sourcebook contains

such topics as isomorphism, computer representation of graphs, planar graphs, graph traversals, Euler and Hamilton circuits, graph coloring, minimum spanning tree algorithms, Dijkstra's shortest path algorithm, topological sort and efficiency and classification of algorithms.

The sourcebook was submitted to two different juries for evaluation. The first jury, consisting of computer science faculty from Pace University, used the book as a reference for their lectures during the spring semester of 1990. The second jury, consisting of mathematics and computer science faculty in the metropolitan area, performed a critical reading. Both juries responded to the same survey asking technical, pedagogical and theoretical questions.

Most jury members agreed to the combining of the mathematical and computing aspects of graph theory in an algorithms course. There was some disagreement as to the use of tracing code during class. A few jury members preferred leaving an algorithm in pseudo-code. Others felt that for some algorithms the students needed the detailed explanation of executable code. The teaching of efficiency of algorithms and NP completeness is difficult, yet the graph theory offers rich examples for this topic. All of the jury members were grateful for the opportunity to discuss pedagogy.

Acknowledgements

There are so many people without whom I could not have completed this work.

I thank God for the talents He has given to me.

I thank my parents, Peter and Frances Fleming, for encouraging my educational pursuits at such an early age.

I thank my children Kathleen, Kevin, Michael and Peter for their love and well-being and my husband Jack for being such a good father to our children.

I was honored to have Dr. Henry Pollak as my sponsor. I thank Dr. Pollak for his insights were always helpful. His encouragement, patience and generosity of time are well appreciated.

I thank my committee Dr. Vogeli and Dr. Rosenbloom for their assistance and useful suggestions during the preparation of the thesis.

I thank the jury members for their giving of valuable time and their interest in good pedagogy: Professors Joseph Bergin, Michael Gargano, Marie Postner, Thomas Smith, Allen Stix, Patrice Tiffany and Carol Wolfe.

M. F. C.

TABLE OF CONTENTS

Chapter

I	BACKGROUND OF THE STUDY.....	1
	The Need for the Study.....	1
	The Purpose of the Study.....	7
	Procedures.....	9
II	SURVEY OF THE LITERATURE.....	11
	Computer Science Education Reports.....	11
	Data Structures and Algorithms Textbooks.....	15
	Discrete Mathematics Textbooks.....	20
	Theoretical Computer Science Textbooks.....	24
	Graph Theory Textbooks.....	25
III	DEVELOPMENT OF THE SOURCE BOOK.....	27
IV	EVALUATION OF THE SOURCE BOOK.....	35
	Report on the Responses of the Questionnaire...35	
	Responses to the Theoretical Questions.....35	
	Responses to the Pedagogical Questions.....39	
	Responses to the Technical Questions.....41	
	Discussion of Responses to the Questionnaire...42	
V	SUMMARY, CONCLUSIONS, AND RECOMMENDATIONS.....	45
	Summary.....	45
	Conclusions.....	52
	Recommendations and Implications.....	55
	BIBLIOGRAPHY.....	57
	APPENDIX A. TOPICS IN THE SOURCE BOOK.....	64
	APPENDIX B. SOURCE BOOK.....	65
	APPENDIX C. HOMEWORK SHEETS.....	129
	APPENDIX D. PROGRAM ASSIGNMENT.....	148
	APPENDIX E. GRAPH PROGRAM.....	149
	APPENDIX F. QUESTIONNAIRE.....	165

CHAPTER I

INTRODUCTION

The Need for the Study

The field of computer science has grown explosively, more rapidly than any other discipline in history. It is unique in that it evolved from researchers from diverse backgrounds instead of emerging from an existing discipline. Other fields, such as molecular biology, had the advantage of emerging from broader disciplines that could contribute researchers of all ages, along with resources and structures. Computer scientists came from many backgrounds and have not been able to bring the support structures of a mother discipline with them. (Hopcroft, 1987, p.202)

Hopcroft notices that the rapid growth of computer science causes difficulty for the discipline. With questions such as "What is Computer Science" still being answered, Computer Science(CS) educators confront many problems.

Less than half of the full time and only 30% of the part-time faculty of CS and CS/Mathematics departments have their terminal degree in Computer Science. (Albers, Anderson & Loftsgaarden, 1987) Forty-nine percent of the lower level courses, among which Data Structures and Algorithms would be considered, are taught by part-time faculty. Many of the faculty are limited in what they can teach and therefore may have a poor overview of Computer Science. "One-third of the full time and 3/5 of the part time CS faculty teach only lower level or specialty courses". (Albers et al., 1987, p.92) It is

doubtful that many of the faculty in CS or Mathematics Departments have studied graph theory in formal coursework. The 1985-1986 Survey of Conference Board of the Mathematical Sciences (CBMS) states that of the 637,000 students studying calculus level mathematics courses only 14,000 students were studying Discrete Mathematics. The discrete mathematics in the Computer Science Departments was offered to 12,000 of the 350,000 studying lower level courses. And only 4,000 of the 142,000 enrolled in upper level CS courses studied Discrete Structures. (Albers et al, 1987) Most of the people who have doctorates in Computer Science teach at universities. Universities use published research for faculty advancement, which encourages the professors to engage heavily in their research. How much involvement do they have in CS education? The National Science Foundation (NSF) workshop on Undergraduate Computer Science Education recommended that Presidential Young Teacher Awards be given to "provide incentives and rewards for creative and successful teaching of undergraduates, indicate to administrators, faculty, and students that both teaching and research are significant and bring national attention to the importance of educational excellence". (Foley, 1988, p.2) Good teaching cannot be assumed or ignored; it must be encouraged and rewarded.

Credit should be given to the professional

educational societies that have placed emphasis on education. Computing Sciences Accreditation Board (CSAB), the accrediting board that emerged from Association for Computing Machinery (ACM) and IEEE Computing Society, has set up standards for curriculum, material, and faculty. ACM's Special Interest Group on Computer Science Education (SIGCSE) publishes a journal quarterly; their annual symposium is well attended, though the educational issues addressed have centered around teaching CS1, CS2, and the use of group projects in software engineering. CS1 and CS2 are the first two computer science courses in the ACM Curriculum '78 (1981). Of the 78 topics in the advance program for '89 SIGCSE Symposium, only 6 deal in any manner with graph theory or algorithm complexity, as did 2 of the 98 topics in '87 SIGCSE. Graph theory and time complexity are difficult topics to teach. Why isn't there more being discussed?

Since 1979 the number and quality of programming textbooks in Pascal have increased dramatically. Elliot Koffman and Nell Dale are two authors of Pascal texts who are interested in educational quality and are very active in SIGCSE. The computer science algorithms textbooks have not been adequate in aiding the teacher or the student in the field of graph theory. Aho, Hopcroft, Ullman(1983) and Tarjan(1983) wrote classical books that are not appropriate for a sophomore level

course in algorithms. Sedgewick's(1986) and Baase's (1988) texts assume elementary material and undergraduate students have a difficult time understanding their texts. Data structures books such as Kruse(1987) and Wulf,Shaw, Helfinger, and Flor(1981) are well written, yet they do not cover graph theory sufficiently. Deo's(1974) and Goodman and Hedetniemi's (1977) texts should be updated with regard to code and flowcharting. Horowitz and Sahni (1978) have a different approach in their text. The graph theory algorithms are not together; algorithms are divided according to their method of solution, greedy, divide and conquer, and backtracking.

The CS curriculum requires the algorithms to be written in a pseudo-code format in preparation for writing a program. Frequently the graph theorists such as Even(1979), Berge(1973), Harary(1969), Melhorn(1984), Gondian and Minoux(1984), and Bondy and Murty(1976) are mathematicians and show no connection between mathematics and computer science. Tremblay and Manohar(1975) and Prather(1976) wrote discrete mathematics textbooks which cover graph theory but not from a CS curriculum viewpoint. Since a number of lower level courses are taught by underqualified faculty, the curriculum material must be particularly well organized. "There is a need to upgrade the skills of current computer science faculty particularly those who switched

into computer science from other disciplines... instructional materials need to be brought up to date". (Foley, 1988, p.5)

Are teaching faculty aware of the importance of graph theory in algorithm courses? The Carnegie Mellon Curriculum Committee places emphasis on a "set of courses that present algorithms and the mathematical foundations of CS with emphasis on integrating the practical aspects of the material with the presentation of the theory." (Shaw,ed., 1985, p.4) Tarjan in his Turing Award Lecture discusses the significance of designing algorithms.

But there is a more profound dimension to the design of efficient algorithms. Designing for theoretical efficiency requires a concentration on the important aspects of a problem so as to avoid redundant computations and to design data structures that exactly represent the information needed to solve the problem.... The result is not only an efficient algorithm, but a collection of insights and methods extracted from the design process that can be transferred to other problems.... it is these insights and general methods that are of most value to practitioners, since they provide tools that can be used to build solutions to real world problems. (Tarjan, 1987, p.205)

In preparing to teach an algorithm, a teacher must be concerned with motivating the student, eliciting an algorithm from the students, deciding on a good data structure, and helping the students to improve their ability to transfer problem solving methods.

Why study graph theory in CS? First, graph theorists need the computer to solve their larger

problems. Since graph theory answers many real world problems, the motivation for learning algorithms comes from the problems themselves. Telecommunications, biology, engineering, psychology, and sociology all use applications of graph theory. Secondly, CS students need graph theory for foundations of other areas in CS. "Networks and trees are used in the study of data structures, compiling, programming languages, operating systems, computational theory, sorting, searching, and AI" (Goodman and Hedetniemi, 1977, p.35). Lastly, the algorithms in graph theory provide great examples for teaching time complexity; it is much easier than using switching functions to teach time complexity. (Personal conversation, H.Pollak, Fall 1989)

Historical background and interesting problems are needed to increase the teachers' ability to motivate the students. What topics in graph theory should be taught in a CS algorithms course? The order of topics and the development must be considered. Is it beneficial for the students to understand isomorphic graphs, planar graphs, Euler and Hamilton circuits before they start coding depth first searches or minimum spanning trees? Appropriate programming problems must be devised to develop the students' understanding of graph theory. The CS curriculum at Pace University offers two courses in Data Structures and Algorithms and 35 to 50% of the second course is set aside for the study of graph

theory.

In addition to the question of qualified faculty, the present focus of Computer Science educators and the lack of appropriate textbooks all suggest a need for Computer Science educators to implement effective means for teaching graph theory and time complexity for undergraduate students.

The Purpose of the Study

The purpose of this study was to produce a sourcebook for college teachers of computer science, especially those with modest mathematical backgrounds, on the topics of graph theory and analysis of algorithms, and to carry out an evaluation of the source book utilizing reviewers who are active college teachers. In order to increase the teachers' opportunities to motivate the students, the mathematical development has been supplemented by historical background, interesting problems and application to programming.

The sourcebook was evaluated by submitting it for critical reading to a sample of college teachers. Some teachers had the opportunity to use the source book in their teaching. On the basis of questionnaires distributed to colleagues, the investigation attempted to answer the following questions.

Technical Questions

Is the material clear and precise?

Is the terminology consistent?

Was the numbering of lines of code useful for class lectures?

Pedagogical Questions

Is the material interesting and motivating?

Are the teaching techniques used helpful?

How important is the code in the teaching of the graph theory algorithms?

Do the homework sheets supplement the lesson properly?

Are there other pedagogical issues that should be addressed?

Theoretical Questions

Does the material help to teach the topic of time complexity?

Is it helpful in the teaching of algorithms?

Do you like the mixing of mathematics with computer science?

Are the topics developed in the correct order?

Are there topics that should be excluded?

Are there topics missing?

Since some responses were made after the sourcebook was used in teaching, interesting feedback was given. This new information, in turn provided additional information for future teachers of an algorithms course.

Procedures

After teaching the topic of algorithms and graph theory several times, the investigator wrote a sourcebook for teachers on the teaching of graph theory. The book ordered and developed topics that are listed in Appendix A. The investigator then used the first draft of the sourcebook in teaching an undergraduate algorithms course in the fall semester of 1989. This teaching led to a number of revisions.

Included in the sourcebook were motivation examples, suggested teaching techniques, homework problems, and programming problems that reinforce and supplement the teaching.

The evaluation of the sourcebook was done in two ways. First, Pace University faculty teaching an algorithms course in the spring semester of 1990 used the sourcebook in preparation for their lectures. In addition to the sourcebook, they received a disk with the solution to the program assignment in Turbo Pascal. At the end of the semester, they responded to a questionnaire prepared by the investigator and in two cases a meeting with the investigator was held. This group was referred to as Jury A. Secondly, a group of four college teachers with different backgrounds from the New York City metropolitan area read the sourcebook

and responded to appropriate questions of the same questionnaire. They are to be known as Jury B. The responses to the questionnaire were then summarized and analyzed.

CHAPTER II

SURVEY OF THE LITERATURE

The Computer Science core that makes up 40 to 60 percent of the computer science requirement "must provide reasonably even emphasis over the areas of theoretical foundations of computer science, algorithms, data structures... Within this portion of the program, analysis and design experiences with substantial laboratory work, including software development, should be stressed." (Computing Sciences Accreditation Board, 1987, p.9) The design and analysis of algorithms are in the core of the Computer Science Curriculum. (ACM Education Board, 1981) This survey of literature includes a review of the current literature on computer science education, a review of popular data structures and algorithms textbooks, a review of discrete mathematics textbooks, theoretical computer science textbooks, and graph theory textbooks.

Computer Science Education Reports

In 1988, a workshop was held at George Washington University, sponsored by the National Science Foundation, on Undergraduate Computer Science Education. The committee members produced a report (Foley, p.3) "identifying the major problems of undergraduate computer science education, and giving possible

solutions to these problems." Setting up quality curriculum in a fast changing field is overwhelming. Getting qualified faculty to teach in undergraduate departments is another hurdle. A new need exists for professionals in other disciplines to learn enough computer science to be competent in the two fields. The committee encourages joint majors where students satisfy the core in both disciplines, and then study additional courses that deal with computing in the second discipline. Good teaching and faculty retention are also discussed with solutions such as teacher awards and the establishment of teaching centers. Computer science must be regarded as a laboratory science with enough equipment, software and staff provided.

In discussing the weaknesses of ACM's Computer Science Curriculum '78, Tucker and Garnick in their paper "A Breadth-First Approach to the Introductory Curriculum in Computing" (1990, p.5) claim "No introductory course in the curriculum model directly relates the principles of logic, combinatorics, and graph theory to their many applications in computing." They also criticize the outdated programming methodology: the teaching of block structured languages such as Pascal rather than modular languages (Modula-2) which allow students to practice principles of abstraction and design. The authors refer to Ralston's (1984) proposal for a mathematics co-requisite for the first programming

course. Listed are the essential topics in mathematics and the topics in computing that require them. Graph theory is listed in the discrete mathematics area and analysis of algorithms, computational complexity, networking problems and compiler are listed in the computing topics requiring the mathematics.

Tucker and Garnick are proposing that the introductory curriculum include more theoretical topics. When theory is taught in a separate course as Discrete Mathematics, "students rarely draw the strong connections between theory and its place within the discipline." (Tucker and Garnick, 1990, p.20). With the teaching of

mathematical definitions of graphs, connectedness, directedness, traversals ... a general overview of networks can reasonably follow, in this context, and thus provide another example of the coherence of theory, abstraction, and design in the discipline of computing. (Tucker and Garnick, 1990, p.22)

In Shaw's paper "Information for a New Century Computing Education for the 1990s and Beyond", (1990) she informs computer science educators what is needed in their departments to be current and relevant. She sees four groups of computer users: computer scientists, computational specialists, light duty developers and casual users. Computer science educators in conjunction with other departments must provide for joint majors to satisfy the growing need of computational specialists in

other disciplines. In discussing flaws in the current software curriculum, Shaw states, "Moreover, students rarely read good programs; it's as if we asked students to write good English without reading good prose."

(Shaw, 1990, p.10)

In the final report of Computing as a Discipline (Denning, 1988), theory, abstraction and design as in the mathematical sciences, natural sciences and engineering are discussed as fundamental in the computing discipline. The committee struggled with defining computing but states,

The discipline of computing is the systematic study of algorithmic processes that describe and transform information: their theory, analysis, design efficiency, implementation and application. The fundamental question underlying all of computing is "What can be (efficiently) automated?" (Denning, 1988, p. 7)

Denning and his committee have set up guidelines for a new curriculum for the introductory course. The first course would include topics from 11 different modules of computing from algorithm concepts and data structures to operating systems and parallel computation. It is being assumed that students are proficient in programming before studying this course. Laboratory work should be supervised and closely aligned with theory.

In Appendix I included in the list of major elements of theory of algorithms are computational complexity theory, classes of problems in P (polynomial

bounded time) and NP (nondeterministic problems in polynomial bounded time) and supporting areas of graph theory and recursive functions. Graph color theory is listed under human computer communication. Appendix II describes in detail the topics for the introductory course.

The ACM Recommended Curriculum for Computer Science and Information Processing Programs in Colleges and Universities; 1968-1981 (1981) includes Curriculum '68 and Curriculum '78 of computer science. CS7 in Curriculum '78 includes graphs, algorithms for finding paths and spanning trees, basic techniques of design and analysis of efficient algorithms and intuitive notion of complexity (e.g.NP- hard problems).

Data Structures and Algorithms Textbooks

In Data Structures with Abstract Data Types and Pascal, Stubbs and Webre (1989) include one chapter of 30 pages on graph theory. The authors explain the data structures, algorithms, code for creating a graph, and traversing graphs in breadth first or depth first manner. Prim's method for determining Minimum Spanning Tree is developed yet Kruskal's method is not given. Introductory material on graphs, Euler paths, Hamilton paths, isomorphic graphs, planarity and NP problems are not mentioned.

Horowitz and Sahni have revised their Fundamentals

of Data Structures in Turbo Pascal (1989) and Fundamental of Data Structures in Pascal (1990). The chapter on graph theory is brief. While the authors claim that only an introductory programming course and modest mathematical backgrounds are prerequisites for the course, the code omitted in some algorithms is difficult. The code for Kruskal's algorithms asking whether an additional edge creates a cycle in minimum spanning tree is quite involved. "Union-find" algorithms are mentioned in a previous chapter on sets, yet the text fails to explain how to use these algorithms in determining whether a cycle has been found.

Isomorphism, planarity, NP problems are not discussed.

Kruse (1987) is the author of Data Structures and Program Design, a text used for teaching the first semester of a data structures and analysis course at Pace University. He devotes a good deal of space to software engineering, recursion and trees. In allowing only two sections for graph theory topics, Kruse (1987, p. 416) writes "we have hardly begun to scratch the surface of the broad and deep subject of graph algorithms."

Tennenbaum and Augenstein (1986) devote a fair amount of space to graph theory in their text, Data Structures Using Pascal. The code when given is complete and readable. Kruskal's method of determining a minimum spanning tree is left as an exercise. They

have not taken the opportunity to discuss classical problems, Konigsberg bridge problem, Hamilton paths and Travelling Salesman problem, graph coloring, and the relationship between these problems and the NP class of problems.

Sedgewick's Algorithms (1986) is the text used presently in this investigators's Data Structures and Analysis of Algorithms II course. John Remington the NY Territory Manager of Addison Wesley mentioned that college professors have a "love/hate relationships with this textbook" (personal conversation, Spring 1989). Sedgewick's presentation is compact; he has a mastery of the topics. The coding is concise, but sometimes difficult to follow. Many times the commentary does not give adequate explanations - it lacks development which is needed for a core course. His pictures can be confusing to the undergraduate student. Hamilton cycles are mentioned briefly as is the Travelling Salesman problem. Euler paths, isomorphic graphs, and graph coloring are missing. He does not explain how the "union find" algorithm is used in Kruskal's determination of minimum spanning tree. In his second edition, he refers to fast find which is incorrectly labelled in Chapter 30. If one has a substantial knowledge of the material, Sedgewick's text would provide an interesting commentary. It is just a difficult text for undergraduate students to read and

understand.

Sara Baase (1988) has also written a text on computer algorithms. In the preface to Computer Algorithms Introduction to Design and Analysis, Baase states that she intended the text for a one semester upper division course in algorithms, analysis of data structures and algorithms where the emphasis is on algorithms. Baase also states she taught the course to students with a strong mathematical background and she left details out in many algorithms. Baase includes graph theory in two chapters in the 2nd edition. For those students who have had a thorough introduction to graph theory in a discrete mathematics course or previous algorithms course, the text would not be an obstacle. Much of the introductory material has been omitted. Euler paths are left for an exercise. Hamilton paths and planarity are not mentioned until the discussion on NP Completeness. The only reference to the history of graph theory is in the Notes section at the end of the chapter. Kruskal's algorithm for minimum spanning tree is discussed in a later chapter since it needs the union find algorithm, but the discussion is mathematical and difficult for a sophomore CS major.

It is interesting that she would use parallel arrays rather than arrays of records in her graph algorithms. "Because the notation for a field in a record in an array is bulky (e.g. VertexData[V].status)

compared with the notation for individual arrays (e.g. status[V]), we will use separated arrays."(Baase, p.162) She does not mention the disadvantages of using parallel arrays.

Horowitz and Sahni's Fundamental of Computer Algorithms (1978) covers many of the important topics of graph theory. The authors state in their preface that they wrote the textbook because they felt what was missing in algorithms textbooks was an emphasis on design techniques. The chapters are organized according to these design techniques - divide and conquer, greedy method, dynamic programming, backtracking, branch and bound. It is a different pedagogical approach but one that can prove difficult for lower class levels. The graph theory is interspersed with other material (sorts, hashing, reliability design, code optimization) and may appear very confusing. The coding (SPARKS) is revised from the first edition of their data structures book, yet the comments and assignment statements prove a hindrance to the readability of the code. In their preface, they "view the material presented here as ideal for a one semester or two quarter course given to juniors, seniors or graduate students." (Horowitz & Sahni, 1978, p. ix) The mathematical notation and ideas presented in the chapter on NP completeness are difficult in nature.

Knuth in his Fundamental Algorithms (1968) was the

first to gather much of the early research in computer science. As a mathematician, Knuth gives much space to mathematical proofs. Ideas such as topological sorting and Prim's algorithm for minimum spanning trees are introduced in exercises. Adjacency matrices and lists, and graph traversals are not mentioned. It is beneficial though for students to be introduced to the reading of Knuth. Aho, Hopcroft and Ullman's text Data Structures and Algorithms (1983) is a popular textbook used for those students who have good backgrounds in mathematics, not only in discrete mathematics but also in logic. Many of the demanding exercises are proofs. Dijkstra's shortest path algorithm is proven. An analysis of many of the algorithms is given in big-oh notation.

Discrete Mathematics Textbooks

John Molluzzo and Fred Buckley in A First Course in Discrete Mathematics (1982) present a discrete mathematics textbook with computer applications. An entire chapter is devoted to graph theory including definitions, isomorphic graphs, minimum spanning trees, planar graphs, adjacency and distance matrices, Euler and Hamilton circuits and the critical path method. There is a special note listing items to check when showing that two graphs are not isomorphic. A number of exercises throughout the chapter include short method

type examples and mathematical proofs. A case study of the heapsort is included in this chapter. There are answers to selected odd-number problems.

Discrete and Combinatorial Mathematics by Ralph Grimaldi (1985) was written for the sophomore-junior level. Graph theory is distributed among three chapters. In addition to the usual topics in graph theory, isomorphism, Euler and Hamilton circuits, Grimaldi adds "real world" type examples. He uses the Instant Insanity problem as an example that can be solved using ideas from graph theory. Grimaldi also includes the definition of homeomorphic graphs and a demonstration of Kuratowski's theorem on planar graphs. Graph coloring and chromatic polynomials comprise an entire section. Breadth first and depth first searches are demonstrated in the chapter on trees. The optimization and matching chapter includes Kruskal's and Prim's algorithm for minimum spanning trees and the max flow min cut algorithm for transport networks. Matching theory and bipartite graphs for which the Stable Marriage problem is an example are also presented.

Skvarcius and Robinson in Discrete Mathematics with Computer Science Applications (1986) have divided graph theory into two chapters: undirected graphs and directed graphs. Algorithms such as nearest neighbor method for the Travelling Salesperson problem are written in pseudo code. The Backus-Naur Form, a language used to describe

the syntax of programming languages, is given as an application of a tree. The exercises at the end of the chapter include method type problems and a few short proofs. The chapter on directed graphs includes topological sort, Warshall's algorithm for reachability, and Dijkstra's algorithm for shortest paths. Graphs are always represented as adjacency matrices. An application of a computer network completes the chapter.

In his preface, Sahni in Concepts in Discrete Mathematics (1985) states that there is a bias toward computer science in this text. He feels that all science students with the traditional approach of learning mathematics through calculus miss out on many important discrete mathematics ideas. An entire chapter is devoted to analysis of algorithms. Intuitive understanding and mathematical analysis are given for big-oh notation. In graph theory, the definitions are given with few examples. Adjacency matrices, packed adjacency lists and adjacency lists are shown to store information from graphs. Pseudo-code is used to demonstrate breadth first and depth first traversals, and Kruskal's spanning tree algorithm. Big-oh notation is discussed throughout the entire chapter on graph theory. Euler's theorem for planar graphs $R = E - V + 2$, homeomorphic graphs, bipartite graphs, and cliques are included in the miscellaneous topics. The examples at the end of the chapter include method type problems and

short proofs.

Dierker and Voxman's text Discrete Mathematics (1986) is offered for courses at the freshman and sophomore level. Graph theory is introduced early in the text for the author finds graphs have interesting applications. In the section on connected graphs, a simple method for determining whether graphs are connected is given. Pseudo code for finding a minimum spanning tree using Kruskal's method is presented. Kruskal's algorithm is mathematically proven. Set notation is used throughout the chapter. Critical path method and its examples are demonstrated. Gray codes are explained using the reading of photoelectric cells on a disk as an example of the Hamilton circuit. The exercises include method type problems and short proofs.

Discrete Mathematics for Computer Science by Angela Shiflet (1987) is offered as a text for a freshman or sophomore level course. After the definitions and computer representation of graphs (only adjacency matrices) are given with a few examples, there is a section on trees with binary searches and preorder, inorder, postorder traversals. Finite state machines are given as an application of graphs. Kruskal's algorithm for minimum spanning tree is given as are Euler's circuits and Hamilton circuits, with no shortest path algorithms. Historical notes are interspersed throughout the chapter. Most of the exercises are

method type problems.

Fred Roberts in Discrete Mathematical Models (1976) discusses graph theory with real world applications, some of which are one-way streets, tournaments, food webs and garbage trucks and colorability.

Tucker's Applied Combinatorics (1984) is divided into two areas: graph theory and enumeration. He proposes it as a text for the discrete mathematics course in ACM's Curriculum'78 and states that it can be used by a wide range of students from sophomores to graduate students. The text uses many examples to teach theory and the problem sets are very extensive. The graph theory section includes the following topics: isomorphism, planar graphs, Euler and Hamilton circuits, graph coloring, minimum spanning trees, shortest path, and network flow algorithms.

Theoretical Computer Science Textbooks

Algorithmics, Theory and Practice by Brassard and Bratley (1988) contains elementary discussion on algorithms, efficiency notation and an introduction to NP completeness.

Garey and Johnson's Computers and Intractability (1979) is the uncontested authority on NP completeness according to Brassard and Bratley (1988). The introduction gives an intuitive understanding of NP

complete problems. Throughout the text are examples of NP complete problems, discussion of P vs. NP, Cook's Theorem and proof and an extensive list of problems that are classified as NP. Included in the list of open problems are graph isomorphism, composite number and linear programming(which is now known to be polynomial).

Kfoury, Moll and Arbib's A Programming Approach to Computability (1982) is a textbook used for theoretical computer science courses. The unsolvability of the Halting problem is discussed several times throughout the text.

Graph Theory Textbooks

In order to include more theorems than otherwise would be possible, Harary (1972, p.V) in Graph Theory has found it "pedagogically advantageous not to include proofs of all theorems." Harary includes the Konigsberg bridge problem, the Four Color Conjecture, cutpoints, connectivity, traversals, planarity, colorability and enumeration.

Shimon Even's Graph Algorithms (1979) is meant to be a textbook for an upper level undergraduate or graduate course. Most of the theorems and lemmas are proven in the text. Even includes Hopcroft and Tarjan's proof of testing planarity in linear time. An entire chapter is devoted to Ford and Fulkerson's maximum flow

in a network and another chapter to NP completeness. Many of the exercises are mathematical proofs.

Biggs, Lloyd and Wilson's Graph Theory 1736-1936 is an unusual mathematics book in that original papers are included in the text. Biographical information is included as the history of graph theory is developed throughout the book. The Koningsberg bridge problem, Knight's tour, trees, chemical graphs, Euler's formula for regions in a planar graph, four color theorem and graph coloring are the main topics of the book.

CHAPTER III

DEVELOPMENT OF THE SOURCEBOOK

One of the main tasks of curriculum planning is selecting and organizing content. (Taba,1962) The material in the source book (Appendix B) on graph theory should provide material for ten two-hour lectures. This would cover at least one-third of a four credit course. In addition to studying sorting and searching algorithms and their time complexity, this leaves little or no time for string processing and geometric algorithms. Taba (1962, p.12) states, "If reflective thinking is an important goal, a thorough study of fewer topics and greater opportunities to relate ideas would be more important than a complete coverage of facts." With the careful study of graph algorithms and trace tables, the students should be able to understand other algorithms on their own, if required in another course or assigned at work. Tracing through code is an exercise that should be developed early in the college career. In addition to "getting the gist" of an algorithm, it is worthwhile to know exactly what is happening. The technique used for tracing recursive code on p.27D of the sourcebook has been instrumental in helping the students understand recursion. Actually, the students gave evidence of the proficiency of understanding a new algorithm by presenting to the entire class an algorithm

that was researched on their own.

The first chapter of many graph theory books contains all the definitions. In contrast, the definitions appear in the sourcebook when they are needed. Problems exist with definitions in graph theory. First instinctive definitions may not be correct. For example in the first definition of graph, an edge connects two distinct vertices. Later on, in the discussion of Euler circuits, a graph with two edges between two adjacent vertices is allowed. It is defined as a multigraph, but is still a graph. The definition of graph has been extended.

The Denning report (1988) and Tucker and Garnick's paper (1990) both propose that the discrete mathematics be taught in computing courses when appropriate. The investigator found it necessary to cover topics like isomorphism, planar graphs, Euler and Hamilton circuits before teaching Kruskal's and Prim's minimum cost spanning tree algorithms and Dijkstra's shortest path algorithm. Covering the data structures for graphs early in the curriculum after a few basic concepts prepares the students for beginning their program assignment. Few deny that computer science is a mathematical science and integrating the two especially in the field of graph theory is rewarding. The level of mathematics needed to study graph theory is not particularly high. However, the more sophisticated the student is in mathematics,

the greater the understanding will be. Familiarity with the different types of proofs would be beneficial. In any case, the students should follow the proofs by contradiction in the sourcebook, the halting problem on page 60D and the lemma on page 35D proving that any connected planar graph has one vertex whose degree is at most 5. The opportunity to use a proof by induction exists with the Five Color Theorem on page 35D. Most students have satisfied the year requirement of calculus, but not all. Many students have not yet taken a course in discrete mathematics.

The curriculum was planned with the hope for much interaction between the teacher and the students. This investigator never agreed with this definition of a lecture - what passes from the notes of the professor to the notes of the students without passing through the minds of either. "A passive mind is still assumed in too large a part of teaching." (Taba, 1962, p.77) Most topics were introduced with a question or problem to stimulate students' thinking. The students should be given an opportunity to discover the algorithms themselves. While the source book has the explanations and code of many of the algorithms, it was never the intention of this investigator for the sourcebook to be lecture notes. Interaction among the students and the teacher should be the goal.

The difference between a tree and other graphs was

given on p.39D. The students are previously familiar with trees, especially binary search trees. This investigator agrees with the ACM Education Board (1981) that the more careful study of graph theory should follow the earlier use of trees in the computer science curriculum. While the data structures themselves are not complicated, the problems to be solved and the tracing of the code are more difficult than searching or traversing through a binary search tree.

The topics from discrete mathematics were chosen for several reasons. Hamilton paths (page 28D) are important for the Travelling Salesman problem - a popular problem in computer science circles. Euler circuits (page 17D) can be found in the Chinese Postman Problem. The ideas of isomorphism and planarity (pages 2D and 10D, respectively) should be learned, perhaps not as intensely in a computer science course as in a mathematics course. Planar graphs are needed in graph coloring. Graph coloring (page 33D) is an interesting problem in discrete mathematics or computer science. Many mathematicians still call the 4 Color Theorem a conjecture since the problem was solved only with the use of the computer.

Time constraints did not allow including algorithmic code for Euler paths or testing planarity. The brighter students may choose either one of them for their class presentation. (see Gibbons, 1985) The code

for Kruskal's algorithm for minimum spanning tree was not included; it assumes knowledge of the "union find" algorithm which has not been previously taught. Network flow algorithms including the Ford-Fulkerson method are quite interesting yet the time constraint does not allow for their discussion in this computer science course.

Homework problems were included on separate sheets. (Appendix C) The tracing of the code on their own will act as a self-test for the students. Being comfortable with a previous lesson also prepares students better for the new material. Faculty may wish to use the sheets in class or for homework; so, the sheets were labelled Sheet #1, etc.

This investigator has included many historical anecdotes she has come across through lectures or readings. Hopefully, they make the course exciting.

This investigator had taught graph theory to undergraduates studying an algorithms course and also to CS graduate students. The notes from the class were corrected and typed into the "A" edition of the sourcebook. This investigator then used the A edition to teach the course during the fall semester of 1989. Corrections were made and examples added and rewritten for the B edition. Also, for the B edition, most of the code was executed on the computer in Pascal. Final touches and revisions including homework sheets and the programming problem were completed for the C edition. A

disk with the graph program in Turbo Pascal (Appendix E) was also distributed. A jury of three computer science teachers at Pace University was selected to use the sourcebook in preparation for their lectures for the spring of 1990. A questionnaire, (Appendix F) was also distributed to this A jury. There was a second jury (Jury B) of teaching faculty from other colleges in the metropolitan area. While they had the opportunity to read the source book and answer the same questionnaire, they were not able to use it in preparation for a particular class. Any corrections and suggestions from both juries and readers were incorporated into the final D edition. Discussion of the evaluation by both juries is in chapter four of the dissertation.

The author discovered writing notes for one's own teaching is very different than writing for someone else's use. Nothing could be left to the imagination of the reader if a certain point had to be made. Having taught a course several times, teachers tend to have less information in their class notes. This is not a good practice since the teacher may forget the struggle of the first time learner.

Code had to be tested and executed to see if it were valid. The only code that was not executed before the B edition was written, was the topological sort on page 54D. Finally, during the C phase, when there was an attempt to execute this particular code, many

problems arose. The original data structure implementation for the graph was not suitable, since an extra field for the incount of each vertex was necessary. The problem was corrected by allowing count to be a parallel array and thereby keeping the same data structure for graph. Topological sort was the first code that used a stack and all the stack operations had to be included. This was a lesson learned for the author in the lack of good software engineering principles.

When the code was to be cleaned up for distribution to the jury, miscellaneous trace statements were deleted, and global variables were changed. Also the data structures for graph and table were changed to records allowing for graph to contain the number of edges and table to contain the number of elements. The investigator decided not to change the data structure in the sourcebook. In order for the students to avoid excessively difficult syntax, the code was to remain simple for class discussions. The students would be expected when writing their program to improve the implementation. This is an excellent teaching strategy since much of the code for their program assignment is given to them during class and changing the implementation will force the students to review and understand the code. In the real world, adapting and improving code is the beginner programmer's major task.

"Programming style should pervade the entire curriculum rather than be considered as a separate task." (Austing et al., 1981, p.121)

CHAPTER IV
EVALUATION OF THE SOURCEBOOK

Report on the Responses of the Questionnaire

Two different juries responded to the questionnaire (Appendix F). Jury A consisted of three faculty members using the sourcebook to prepare their lectures for an algorithms course they were teaching in the spring semester, 1990. The participants of Jury B were faculty members in mathematics and computer science departments who read the sourcebook but did not have the opportunity to use it for a class. This chapter first gives a report of their responses to the questionnaire followed by a discussion of the material that was or was not incorporated into the D and final version of the sourcebook.

The questionnaire was designed so that the participants of the jury would first respond to the short answer technical questions, and second to the pedagogical and theoretical questions. In this chapter, the responses to the theoretical questions from the survey are discussed before the responses to the pedagogical and technical questions.

Responses to the Theoretical Questions

Controversy existed on the teaching of introductory ideas of graph theory - isomorphism, planar graphs, and

graph coloring in a computer science course. A member of the B jury didn't think it was necessary but believes it gives the students a richer experience. An A juror stated, "it demonstrates to students that when things are most difficult the theory gives (up) the most help." Another A juror agreed totally with this teaching of graph theory in computer science and extended it much further than the material contained in the sourcebook. Since he eliminated proofs and tracing of code from his lectures, he added additional topics to his lectures: Heawood's Map Theorem, Heuristic coloring, Matchings, Flows/networks, and Max Flow Min Cut Theorem. A member of the B jury thought students like mathematics better when they see its applications. She thought the end of the source book was a bit terse and more text and examples should be added there.

Alternatively, an A juror stated that these topics belong in a discrete mathematics course. The curriculum for the algorithms course consisted of staple algorithms - depth first and breadth first traversals, finding the minimal spanning tree and Dijkstra's shortest path algorithm. Therefore, the topics of isomorphism, planarity and graph coloring are not necessary.

I feel that there is no better place than "in graph theory" for students to acquire generally transferrable insights and experiences in the fashioning of data structures and the design of algorithms; and I feel that by a fourth computer science course they are ready for these abstract lessons. This is what I aim to teach, and I feel it is much more important than any specific pieces of content (i.e. these lessons are primary; the graph theory itself is just the medium.)

The jury however agreed when discussing the mixing of mathematics and computer science. Faculty pro-mixing believed it would do wonders for mathematics, and allow students to get a more realistic picture. This also encourages the students to become more serious. The 5 color theorem and proof of the lemma concerning planar graphs on page 35D were good ways of showing different proof methods: proof by induction and proof by contradiction. A member of jury B said he introduced a variety of proofs in the introductory discrete mathematics course and would like the variety emphasized in later courses.

The faculty member opposing the combining of mathematics and computer science did not want to devote large amounts of class time explaining methods of proof by induction or proof by contradiction. "We have more than enough teaching algorithms and the issues associated with their implementation."

One B juror was cautious. He liked the "idea of incorporating as much mathematics as possible in the computer science curriculum because I think we do our students a serious disservice when we neglect it." He

was concerned that we don't have enough time to cover the analysis of algorithms with respect to the sorting algorithms. "The danger here, I think, is that it may be too easy to turn a computer science course using mathematical ideas into a mathematics course using computer programs."

The sourcebook neglected to mention what a greedy algorithm is and while stating that Prim's algorithm is greedy, it was never stated that Kruskal's algorithm is also greedy. The author added this statement to page 46D of the sourcebook.

Additional topics suggested for the course include critical path analysis as an extension of topological sort. Computing reachability by exponentiating the adjacency matrix is useful in determining whether adding an edge to a digraph will complete a cycle and this is useful in implementing Kruskal's MCST algorithm. Petri nets could be added since they enable concurrent processing to be modeled and studied. Another juror suggested the problem of finding the longest circuit in a graph. In teaching the different traversals of graphs, a juror stated the opportunity of discussing the merits of breadth first and depth first traversals should not be missed.

The jurors agreed with Dale that not much attention has been given to the CS 7 Algorithms course, and there should be. The students were very enthusiastic about

graph theory; it is so new to them and they see it as practical. A "B" juror stated, "My own experience in teaching graph theory is that students find the topic extremely interesting and entertaining until they realize how complex the problems become."

Responses to the Pedagogical Questions

All but one of the jury members enjoyed the historical anecdotes and were able to use them in their lectures. The background of the minimum cost spanning tree was especially interesting since it is recent history. An A jury member criticized that some anecdotes were not as germane. "That double and triple routes are disallowed on Mother's Day" (page 38D) is not directly pertinent to anything we're trying to teach." The incomplete anecdotes may misconvey the truth. The annals of mathematics contain more than one single erroneous proof of the four color problem. By mentioning just one proof (Kempe on page 34D), the nature of mathematics may be misapprehended. "It is harmful for students to suppose that a flawed proof is something that just doesn't/can't happen, or, at worst, the rare exception." How touch tone dialing is an example of the Gray code is unclear.

The question of tracing code provoked the most discussion. One B juror stated it is a very effective

tool; especially for recursion. She stated, "I don't think we can be anything but exact, we are looking for exact answers." However, she did not think code should be passed out to the students immediately; the class should develop the code together. Another B juror placed "quite a bit" of emphasis on tracing code. A third B juror stated that "unfortunately students still need a lot of programming experience." The jurors from the A group did not trace code in class and did not attempt to use the traces from the sourcebook. One professor used pseudo-code, basically because it is less time consuming and he can cover more algorithms. Another faculty member believed that presenting "a trace to the class relegates the students to bored bystanders." He concentrated on the higher level algorithmic process, which ends up in pseudo-code and believes this is all students need to know to grasp the basic idea. He elicited data structures and algorithmic steps from the students through pertinent questions, such as "How do you know which vertices are adjacent to the one just removed?" or "How do you mark a vertex as processed?" The homework sheets proved valuable for reinforcing ideas presented in class. One A juror who used them for his class would have preferred more examples. A B juror suggested there be an answer key to the homework problems.

Two A jurors commented that the chart method for

Dijkstra's SSSP problem (on p. 47D) was helpful. The use of parallel arrays and a simpler implementation of graphs for class lectures were considered acceptable as good teaching techniques. An A juror stated that students should see different implementation strategies. As an additional problem, converting a sparse matrix (for implementing a graph) to a reduced matrix was suggested. Another A juror suggested introducing trees earlier in the sourcebook on Day 5 and showing that a level by level traversal of a binary tree is a breadth first traversal. A member of the B jury requested a statement of output for each program. She also thought the use of the pitcher problem on page 20D was a good technique in explaining the difference between breadth first and depth first traversals.

Responses to the Technical Questions

The members of the juries agreed the technical aspect of the sourcebook was good. Keeping the notation consistent and the lines of code numbered were beneficial. For the most part diagrams were clear and informative. The faculty members saw the benefits of using the graph of the Southern states in class and the graph of the Western states for homework. One member of Jury A would have liked more examples for classwork and homework.

Two readers from Jury B did not like the use of a

term in the sourcebook previous to its definition. Also an index for the sourcebook was requested.

Discussion of the Responses to the Questionnaire

It is unfortunate that members of Jury A did not attempt to trace code in class. As a result, it is not known whether their students could have benefitted from it. In my teaching experience, students have always commented that tracing through code has helped them to understand the step by step process of an algorithm. It also shows students good code to model. Shaw (1990, p.10) states, "Moreover, students rarely read good programs; it's as if we asked students to write good English without reading good prose." It is noteworthy that four B jurors support the idea of tracing code in class as a good teaching technique.

Just "getting the gist" of an algorithm through pseudo code is not enough for the average underclassman. The suggestions, from the juries to elicit the algorithms from the class through pertinent questions, were excellent and have been incorporated into the sourcebook on page 42D and page 48D. It was never the intention of this investigator for the sourcebook to be the lecture notes for the professor. Undoubtedly, if the teacher traces code in class, less time is available for covering other material. Each professor must know the capabilities of his/her students and set the tone of

the class accordingly. A variety of teaching methods in one course proves valuable for the students.

Concerning the mixing of the mathematics and computer science, each teacher has his/her own ideas. One believes he should cover as much material as possible; he only expects students to comprehend 40% of what he teaches. Hoping that the students learn the introductory graph theory topics in their discrete mathematics course, another jury member wishes to emphasize just algorithms in the computer science course. Since a sourcebook is "all things to all people", it offers different opportunities for teaching. Tucker and Garnick (1990, p.20) state that when theory is taught in a separate course as Discrete Mathematics, "students rarely draw the strong connections between theory and the place within the discipline." With this new trend of teaching literature and history in Western Civilization courses, combining topics in mathematics and computer science seems appropriate.

One B juror was concerned that there was not enough time for the analysis of algorithms with respect to the sorting algorithms. This was not the case in this investigator's experience. The level of difficulty of the problems in graph theory was not high and therefore not time consuming.

As to the suggestion of introducing trees earlier in the sourcebook, trees are taught in the second course

of a computer science curriculum. A level by level traversal of a tree using a queue (without mentioning breadth first traversal) is usually a problem to be solved during that course. It is now fitting in the algorithms course that this type of traversal be called breadth first traversal.

The author agreed that the material on classification of algorithms and NP completeness was not clear. The material was revised for the D edition.

It was decided to leave the definition of a term following its use just as it was in the sourcebook. The investigator found this to be a valid teaching technique (spiral teaching) and it only bothered two jurors.

As it is necessary for computer science faculty to stay current in the field, very little time has been left for pedagogical discussion. The reviewing of the sourcebook by fellow faculty has led to pedagogical discussion with colleagues that had not been done previously. All who have participated are grateful for this. A member of the B jury (who has not yet taught an algorithms course) intends to use some of the examples in his introductory discrete mathematics course to whet the students' appetites for the types of problems that they will meet as they progress through the computer science major.

CHAPTER V
SUMMARY, CONCLUSIONS, RECOMMENDATIONS

Summary

The rapid growth of computer science has made developing curricula difficult for computer science educators. "What is computer science" is still being asked and answered. There is the question of qualified faculty. Less than half of the full time faculty have their terminal degree in computer science. Many of the faculty, full and part time, teach only lower level or specialty courses. The faculty who earn Ph.D.'s in Computer Science are usually heavily engaged in research. Do they have the time or inclination to be involved in computer science education? Results of surveys by the Conference Board of Mathematical Sciences state that few students are studying discrete mathematics courses and fewer are studying a higher level discrete structures course.

This is not to say there has been no interest in computer science education. The Computing Sciences Accreditation Board (CSAB), a joint committee of ACM and IEEE Computing Society, has set up criteria on curriculum, hardware, and faculty for accreditation procedures. SIGCSE, the educational special interest group of ACM, sponsors a yearly conference. However,

many of the topics have centered around software engineering and the first two programming courses. There has been very little discussion of graph theory and algorithm complexity. Dale at the '90 SIGCSE Symposium asked "Whatever happened to CS7 - the Algorithms Course?"

Textbooks for the first two programming courses have increased dramatically in number since 1979. This has not been true for data structures and algorithms textbooks. The texts that contain more theory are usually aimed at an upper level undergraduate or graduate student. Yet, Pace University's curriculum (as suggested by CSAB) requires us to teach an algorithms course with theory in the sophomore year. Designing efficient algorithms is important to the computer scientist. In teaching an algorithm, the teacher must be concerned with motivating the student, eliciting steps of the process of the algorithm, deciding on a good data structure and helping the students to improve their ability to transfer problem solving methods.

Graph theory is an area of mathematics and computer science that is rich with algorithms. Graph theory answers many real world problems and has applications in telecommunications, biology, engineering, psychology and sociology. Graph theory provides good examples for teaching time complexity and an introduction to NP

completeness. The history of graph theory is interesting; many of the theorems and algorithms have been recently developed. One of the questions discussed in chapter three of the dissertation is what should be included in the graph theory section of an algorithms course.

The purpose of the study was to produce a sourcebook for college teachers of computer science, especially those with modest mathematical backgrounds, on the topics of graph theory and analysis of algorithms, and to carry out an evaluation of the source book utilizing reviewers who are active college teachers.

The sourcebook was evaluated by two juries of college teachers. The first jury (A) had the opportunity to use the sourcebook in the preparation for their lectures for an algorithms course in the spring of 1990. The second jury (B), who consisted of computer science faculty in the New York metropolitan area, did a reading of the sourcebook. Both juries responded to a questionnaire (Appendix F) that included the following questions concerning the sourcebook:

Technical Questions

Is the material clear and precise?

Is the terminology consistent?

Is the numbering of lines of code useful?

Pedagogical Questions

Is the material interesting and motivating?

Are the teaching techniques used helpful?

How important is code in the teaching of graph theory algorithms?

Do the homework sheets supplement the lesson properly ?

Are there other pedagogical issues that should be addressed?

Theoretical Questions

Does the material help to teach the topic of time complexity?

Is it helpful in the teaching of algorithms?

Do you like the mixing of mathematics with computer science?

Are the topics developed in the correct order?

Are there topics that should be excluded?

Are there topics missing?

Included in the investigator's research were reviews of computer science education papers, popular data structures and algorithms textbooks, discrete mathematics textbooks, theoretical computer science textbooks, and graph theory textbooks. The more recent papers on computer science curriculum, Denning (1988) and Tucker and Garnick (1990), encourage the teaching of discrete mathematics in computer science courses. Data structures textbooks devote very little space to graph theory algorithms. Most of the algorithms textbooks were written for an upper level undergraduate

student with a good mathematics background. The discrete mathematics textbooks and graph theory textbooks do not show the connection between mathematics and computer science. The theoretical computer science textbooks offer insight into time complexity and NP completeness.

One of the members of Jury A offers a good description of a sourcebook.

A sourcebook is supposed to give help to teachers regardless of the textbook they use, the topics they treat, the order in which topics are covered, and their underlying instructional objectives.

A sourcebook ought to offer:

- * a "brush-up" for the instructor who is only marginally conversant with the ideas and algorithms that must be taught - a place to turn for down-to-earth explanation and help
- * excellent methods for teaching things the instructor has not had experience covering to a class (or has not had good success with)
- * good exercises for class use, problems for homework, programming assignment possibilities, and suggestions for larger projects
- * unusual anecdotes, sidelights, easy-to-grasp applications, and things the instructor can use to generate flashes of insight; maybe, even jokes, riddles, and puzzles
- * a table of contents and index for easy access of definitions, conceptual discussions, algorithms, and exercises; maybe even a glossary
- * a "key" to the literature; textbooks, popular articles, etcetera to which the instructor can turn for more extensive and alternate treatments of each idea and algorithm

The sourcebook was developed after the investigator had taught graph theory in algorithms courses several times. In contrast to many textbooks, the definitions appear in the sourcebook when needed. Topics, such as

isomorphism, planar graphs, Euler and Hamilton circuits, and graph coloring, were presented to the students before the teaching of Kruskal's and Prim's minimum cost spanning tree algorithms and Dijkstra's shortest path algorithms. It gave the students a more comprehensive view of graph theory. The data structures for graphs were presented early in the curriculum for the students to begin program assignments early. The union of mathematics and computer science enables the students to form a more realistic view of mathematics. The chance to use different types of proofs in the sourcebook was beneficial to the students' total education.

The curriculum was planned with the hope for interaction between the teacher and the students. The students should be given the opportunity to discover the algorithms themselves. Homework sheets were developed to give the students more practice; something that this investigator's previous students had requested. The programs for certain algorithms, breadth and depth first traversals, Prim's minimum cost spanning tree algorithm, Dijkstra's shortest path algorithm and topological sort were coded in Turbo Pascal. (Appendix E) The sourcebook went through several revisions. The A version was written before the investigator taught the course in the fall semester of 1989. The B version contained the corrections as the result of that teaching. The C version with homework sheets, program assignment,

questionnaire and disk with the solution to the program assignment in Turbo Pascal (Appendix E) were distributed to Jury A for preparing their lectures for the spring semester of 1990. With comments from both juries and readers, revisions were made for the final D version.

Juries A and B responded to the questionnaire.

(Appendix F) With regard to the theoretical questions, there was some discussion on teaching the introductory ideas of graph theory - isomorphism, planar graphs, and graph coloring - in a computer science course. All jury members except one supported the idea of teaching these topics in a computer science algorithms course. The one professor asserted these topics belonged in a discrete mathematics course since he needed the time to design algorithms and fashion data structures. The same philosophy held true among the jury members when questioned whether mathematics should be mixed with computer science. Also, there was the question whether time should be spent on different methods of proof such as the proof by induction for the 5 Color Theorem and the proof by contradiction for the lemma concerning planar graphs on page 35D. A member of jury B had introduced a variety of proofs in an introductory discrete mathematics course and appreciates the variety emphasized in later courses. Another professor skipped many of the proofs but was able to cover more topics in graph theory: Heuristic colorings, matching theory,

flows/networks, and Max Flow Min Cut Theorem. The material on NP completeness and classification of algorithms was not satisfactory to the juries, so with their suggestions, this section was revised for the final D edition of the sourcebook.

The responses to the pedagogical questions included the questioning of using incomplete anecdotes in the sourcebook. However, most jury members agreed the use of anecdotes was a good pedagogical device. The three members of jury A did not trace code during class time; they leave their algorithms in pseudo code all of the time. A member of Jury B thinks it is necessary to be exact with the students (that is, tracing code in class) only after the algorithm has been developed by the class. The four B jurors did support the idea of tracing code in class.

With regard to the technical aspect of the source book, most members of the juries agreed the notation was consistent and the material clear, and a B juror states consistency is always a plus. Two readers from Jury B objected to the use of a term before it was clearly defined. However, the other jury members agreed with the spiral development in the teaching of the definitions.

Conclusions

The results of this study have led the investigator to make several conclusions.

1. The sourcebook is beneficial for faculty to explore even though they may not use all of the ideas in their own algorithms course. It gives the opportunity for faculty to reflect and discuss pedagogy. The sourcebook acts as a brush up for the computer science instructor who is not as well versed in discrete mathematics. The examples are clear and easily understood. The proofs are easy to follow and at a level presentable for the undergraduate student. The supplementary homeworks give the students a chance to review and offer the opportunity to explore and develop ideas on their own.

2. Differences among faculty exist in the choice of content for the graph theory component of the computer science algorithms course. At least one jury member insisted that the introductory topics in graph theory belong in a discrete mathematics course. However, there were several jury members who enjoyed combining the mathematics and computer science aspects of graph theory. This is so because the computing offers real world applications for mathematics and the mathematics offers theoretical support in the teaching of computer science.

3. It is difficult for faculty to implement methods of teaching other than their own. The three members of jury A who used the sourcebook to prepare their lectures

did not trace code in class. While the members from jury A were self-assured about their teaching methods, a need may arise in the future for a different method of teaching when dealing with a particular group of students. The sourcebook offers them a solid pedagogical approach. The members of jury B saw the need for tracing code in class, as many of their students are not adept in programming. The guided approach of tracing code develops an essential characteristic of strong programmers.

4. Teaching faculty learned the topics of graph theory in different ways: mathematics programs, computer science programs, sociology programs, or self-taught. If the need arises, the sourcebook can be used as the "putty to fill some of the gaps" in the background of computer science faculty. The bibliography offers a wide range of references.

5. The teaching of efficiency of algorithms and NP completeness is difficult. It may be a topic similar to recursion that should be introduced in one course but reviewed in several other courses. Graph theory offers examples in NP completeness and classification of algorithms. Faculty should not waste the opportunity to introduce these topics at this time.

Recommendations and Implications

The investigator makes several recommendations after studying the current research, completing the source book, having the source book used and evaluated by teaching faculty, and discussing the topic with colleagues.

1. Sourcebooks are a rich source of information from which faculty can choose material for their lectures. Sourcebooks are an aid in clarifying curriculum for new and adjunct faculty. Since computer science curriculum is still in its infant stage, sourcebooks on other topics should be written.
2. Even though faculty in computer science departments are always trying to keep current with new discoveries and technology, it is necessary for CS faculty to set aside time for discussing pedagogy. ACM's SIGCSE should encourage presentations on the teaching of algorithms at its annual symposium. Faculty should employ various teaching methods in their lectures.
3. Curriculum topics for the earlier programming and mathematics courses should be more clearly established in order that students are better prepared and topics are not repeated without cause.

4. The question of whether the topics in discrete mathematics needed for graph theory should be taught in mathematics courses or computer science courses must be answered. In most schools this would mean cooperation between the mathematics and computer science departments.

5. Experiments testing different teaching techniques should be developed to test students' learning of algorithms. For example, is an algorithm better understood by writing executable code rather than leaving a solution in pseudo code?

6. Current computer science education literature has recommended a laboratory component of computer science courses. (Denning, 1988). Experimental type program assignments for graph theory should be developed, i.e. testing whether one data structure is more efficient than another or whether one algorithm is quicker.

BIBLIOGRAPHY

- ACM Education Board. (1981). ACM Recommended Curricula for Computer Science and Information Processing Programs in Colleges and Universities, 1968-1981. New York: Association for Computing Machinery.
- Aho, Alfred, Hopcroft, John & Ullman, Jeffrey. (1983). Data Structures and Algorithms, Reading, Massachusetts: Addison Wesley.
- Alavi, Y., Chartrand, G., Lesnick, L., Lick, D.R., & Wall, C.E., (Eds.). (1985). Graph Theory with Applications to Algorithms and Computer Science, Fifth International Conference, 1984. New York: John Wiley and Sons.
- Albers, Donald J., Anderson, Richard D., & Loftsgaarden, Don O. (1987) Undergraduate Programs in the Mathematical and Computer Sciences, The 1985-1986 Survey. Washington D.C.: Mathematical Association of American.
- Austing, Richard H., Barnes Bruce H., Bonnette, Della T., Engel, Gerald L., Stokes, Gordon, (eds.). Curriculum '78. (printed in ACM Recommended Curricula for Computer Science and Information Processing Programs in Colleges and Universities, 1968-1981. New York: Association for Computer Machinery).
- Baase, Sara. (1988). Computer Algorithms Introduction to Design and Analysis, (2nd ed.). Reading, Massachusetts: Addison-Wesley.
- Beane, James A., Toepfer, Conrad F., Allesi, Samuel J. (1986). Curriculum Planning and Development. Boston: Allyn and Bacon Inc..
- Berge, Claude. (1973) Graphs and Hypergraphs. Amsterdam: North Holland Publishing Company.
- Berlioux, Pierre and Bizarre, Philippe. (1986). Algorithms, The Construction, Proof and Analysis of Programs, New York: John Wiley and Sons.
- Bertztiss, A.T. (1971). Data Structure Theory and Practice. New York: Academic Press.

- Biggs, Norman L., Lloyd, E. Keith & Wilson, Robin J.. (1976). Graph Theory 1736-1936. Oxford: Clarendon Press.
- Boesch, Frank. (1982). Introduction to Basic Network Problems. The Mathematics of Networks. Proceedings of Symposia in Applied Mathematics, Providence, Rhode Island: American Mathematical Society.
- Bondy, J.A. & Murty, U.S.R. (1976) Graph Theory with Applications. New York: American Elsevier Publishing Company.
- Brassard, Gilles and Bratley, Paul. (1988) Algorithmic Theory and Practice. Englewood Cliffs, New Jersey: Prentice Hall.
- Computing Sciences Accreditation Board. (1987) Criteria for Accrediting Programs in Computer Science in the United States. (CSAB, 345 East 47th St. New York, NY 10017).
- Dale, Nell & Weems, Chip. (1987). Introduction to Pascal and Structured Design. Lexington, Massachusetts: D.C.Heath.
- Denning, Peter J. (1988). Computing as a Discipline, Appendix I, Appendix II. (Final report of the ACM Task Force on the Core of CS - chair Peter J. Denning NASA Ames Research Center Moffet, California, 94035).
- Deo, Narsingh. (1974). Graph Theory with Applications to Engineering and Computer Science. Englewood Cliffs, New Jersey: Prentice Hall.
- Dewdney, A.K. (1989). The Turing Omnibus. Rockville, Maryland: Computer Science Press.
- Dierker, Paul F. & Voxman, William L. (1986). Discrete Mathematics. New York: Harcourt Brace Jovanovich, Publishers.
- Dijkstra, Edsger. (1989). On the Cruelty of Really Teaching Computer Science. Communications of the ACM, 12, 1397-1414.
- Edmonds, Jack & Johnson, Ellis L. (1973). Matching, Euler Tours and the Chinese Postman Problem, Mathematical Programming 5, Amsterdam: North Holland Publishing.
- Even, Shimon. (1979). Graph Algorithms. Rockville, Maryland: Computer Science Press.

- Foley, James. (1988). (ed.) Undergraduate Computer Science Education. (Report of a workshop sponsored by National Science Foundation, Washington D.C.).
- Frenkel Karen A. (1987). Turing Award Interview, An Interview with the 1986 A.M. Turing Award Recipients - John E.Hopcroft and Robert E. Tarjan, Communications of the ACM, 3, 214-222.
- Gardner, Martin. (1984). Sixth Book of Mathematical Diversions from Scientific American. Chicago: University of Chicago Press.
- Garey, Michael & Johnson, David S. (1979) Computers and Intractability, A Guide to the Theory of NP-Completeness, San Francisco: W.H. Freeman and Company.
- Gibbons, Alan. (1985). Algorithmic Graph Theory. New York: Cambridge University Press.
- Gilbert, E.N. and Pollak, H.O. (1968) Steiner Minimum Trees, SIAM Journal on Applied Mathematics, 1, pp. 1-29.
- Gondian, Michel & Minoux, Michel. (1984). Graphs and Algorithms. New York: John Wiley and Sons.
- Goodman, S.E. & Hedetniemi, S.E. (1977). Introduction to the Design and Analysis of Algorithms. New York: McGraw Hill.
- Gotlieb, C.C. and Gotlieb, L.R. (1978). Data Types and Structures. Englewood Cliffs, New Jersey: Prentice Hall.
- Gries, David. (1987). 1985-1986 Taulbee Surbey. Communications of the ACM, 8, 688-694.
- Grimaldi, Ralph P. (1985). Discrete and Combinatorial Mathematics. Reading, Massachusetts: Addison-Wesley.
- Hansen, P. (ed.) (1982). Studies on Graphs and Discrete Programming. Amsterdam: North Holland Publishing Company.
- Harary, Frank. (1969). Graph Theory, Reading, Massachusetts: Addison Wesley.
- Harary, Frank. (1973). (ed.) New Directions in the Theory of Graphs, New York: Academic Press.

- Hopcroft, John E. (1987). Computer Science: The Emergence of a Discipline, Turing Award Lecture, Communications of the ACM, 3, 198-202.
- Horowitz, Ellis & Sahni, Sartaj. (1976). Fundamentals of Data Structures, Rockville, Maryland: Computer Science Press.
- Horowitz, Ellis & Sahni, Sartaj. (1978). Fundamentals of Computer Algorithms, Rockville, Maryland: Computer Science Press.
- Horowitz, Ellis & Sahni, Sartaj. (1989). Fundamentals of Data Structures in Turbo Pascal. New York: W.H. Freeman and Company.
- Horowitz, Ellis & Sahni, Sartaj. (1990). Fundamentals of Data Structures in Pascal. New York: W.H. Freeman and Company.
- Johnson, David, Nishizenki Takao, Nozaki Akihiro, & Wilf, Herbert S. (1987). (eds). Discrete Algorithms and Complexity, New York: Academic Press.
- Kerschenbaum, A. & Van Slyke, R.. (1972) Computing Minimum Spanning Trees Efficiently, Proceedings of the ACM Conference, August 1972.
- Kfoury, A.J., Moll, Robert N. & Arbib, Michael A. (1982). A Programming Approach to Computability, New York: Springer-Verlag.
- Koffman, Elliot B. (1989). Pascal Problem Solving and Program Design (3rd ed.) Reading, Massachusetts: Addison-Wesley.
- Knuth, Donald. (1968). The Art of Computer Programming, Fundamental Algorithms. (Vol.1) Reading, Massachusetts: Addison Wesley.
- Kronsjo, Lydia. (1979). Algorithms, Their Complexity and Efficiency. (2nd ed.). New York: John Wiley and Sons.
- Kruse, Robert L. (1987). Data Structures and Program Design. Englewood Cliffs, New Jersey: Prentice Hall.
- Lewis, Harry R. & Papadimitriou, Christos H. (1978). The Efficiency of Algorithms, Scientific America, 1, 96-109.

- Lin, S. & Kernighan B. (1973). An Effective Heuristic Algorithm for the Travelling Salesman Problem. Operations Research, 21, pp. 498-516.
- Melhorn, Kurt. (1984). Graph Algorithms and NP Completeness, New York: Springer-Verlag.
- Minsky, M.L. (1967) Computation: Finite and Infinite Machines, Englewood Cliffs, New Jersey: Prentice Hall.
- Molluzzo, John C. & Buckley, Fred. (1986) A First Course in Discrete Mathematics, Belmont, California: Wadsworth Publishing Company.
- Nijenhuis, Albert & Wilf, Herbert S. (1975) Combinatorial Algorithms, New York: Academic Press.
- Ore, Oystein. (1962). Theory of Graphs. Providence, Rhode Island: American Mathematical Society.
- Prather, Ronald. (1976). Discrete Mathematical Structures for Computer Science, Boston: Houghton Mufflin Company.
- Ralston, A. (1984) The First Course in Computer Science Needs a Mathematics Corequisite, Communications of the ACM, 10, pp. 1002-1005.
- Ralston, Anthony & Meek, Chester. (1976). (eds.) Encyclopedia of Computer Science. New York: Petrocelli/Charter.
- Roberts, Fred S. (1976). Discrete Mathematical Models. Englewood Cliffs, New Jersey: Prentice-Hall.
- Sahni, Sartaj. (1985) Concepts in Discrete Mathematics, North Oaks, Minnesota: The Camelot Publishing Company.
- Sedgewick, Robert. (1986). Algorithms. (2nd ed.) Reading, Massachusetts: Addison Wesley.
- Shaw, Mary. (Ed.) (1985). Carnegie Mellon Curriculum, New York: Springer-Verlag.
- Shaw, Mary. (1990). Informatics for a New Century Computing Education for the 1990s and Beyond (to appear as Carnegie Mellon University Technical Report CMU-CS-90-142).
- Shiflet, Angela B. (1987). Discrete Mathematics for Computer Science. New York: West Publishing Company.

- Skvarcius, Romualdas & Robinson, William B. (1986). Discrete Mathematics with Computer Science Applications. Menlo Park, California: The Benjamin/Cummings Publishing Company, Inc.
- Stanat, Donald F. & McAllister, David F. (1977). Discrete Mathematics in Computer Science. Englewood Cliffs, New Jersey: Prentice-Hall.
- Stewart, Ian. (1987) The Problems of Mathematics. Oxford, England: Oxford University Press.
- Stubbs, Daniel F. & Webre, Neil W. (1989). Data Structures with Abstract Data Types and Pascal. Pacific Grove, California: Brooks/Cole Publishing Company.
- Taba, Hilda. (1962). Curriculum Development Theory and Practice, New York: Harcourt Brace Jovanovich, Inc..
- Tarjan, Robert E. (1983). Data Structures and Network Algorithms. Philadelphia, Pennsylvania: Society for Industrial and Applied Math.
- Tarjan, Robert E. (1987). Algorithm Design, Turing Award Lecture. Communications of the ACM, 3, 205-212.
- Tenenbaum, Aaron M. & Augenstein, Moshe J.. (1986) Data Structures Using Pascal. Englewood Cliffs, New Jersey: Prentice-Hall Incorporated.
- The Report of the MAA/ACM/IEE Computer Science Task Force on Teaching Computer Science within the Mathematics Department. 1988. (David Bellew, committee member, Western Illinois University).
- Thesen, Arne (1972). Computer Methods in Operations Research, New York: Academic Press.
- Trees in Algebra and Programming, (1983) 8th Colloquium, CAAP '83. New York: Springer-Verlag.
- Tremblay, Jean-Paul & Sorenson, Paul G. (1984). An Introduction to Data Structures with Applications. New York: Mc Graw Hill.
- Tremblay, Jean-Paul & Manohar, R. (1975). Discrete Mathematical Structures with Application to Computer Science. New York: Mc Graw Hill.
- Tucker, Alan. (1984). Applied Combinatorics (2nd ed.). New York: John Wiley & Sons.

- Tucker, Allen B. & Garnick, David K. (1990). A Breadth-First Approach to the Introductory Curriculum in Computing. Unpublished manuscript, Bowdoin College, Brunswick, Maine.
- Wilf, Herbert S. (1986). Algorithms and Complexity, Englewood Cliffs, New Jersey: Prentice-Hall.
- Wulf, William A., Shaw, Mary, Helfinger, Paul N. & Flon, Lawrence. (1981). Fundamental Structures of Computer Science, Reading, Massachusetts: Addison Wesley Publishing Company.

APPENDIX A

TOPICS IN THE SOURCEBOOK

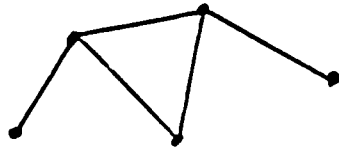
1.	Preliminary Definitions.....	1D
2.	Isomorphism.....	2D
3.	Computer representation of graphs.....	6D
	Adjacency matrix.....	7D
	Adjacency list.....	8D
	Edge adjacency.....	10D
4.	Planar graphs.....	10D
	Kuratowski's Theorem.....	12D
5.	Euler's Theorem on Regions	
	$R = E - V + 2$	15D
6.	Euler paths and circuits.....	17D
7.	Traversing graphs.....	20D
	Breadth first traversal.....	21D
	Depth first traversal.....	25D
8.	Hamilton circuit.....	28D
9.	Gray code.....	31D
10.	Graph coloring.....	33D
	Proof of 5 Color Theorem.....	35D
11.	Introduction to Network type problems.	37D
12.	Minimum spanning tree	
	Kruskal's method of solution.....	40D
	Prim's method of solution.....	41D
13.	Single Source Shortest Path	
	Dijkstra's Algorithm.....	47D
14.	All pairs shortest path	
	Floyd's Algorithm.....	52D
15.	Topological sort.....	54D
16.	Efficiency and classificaton of	
	algorithms.....	57D

APPENDIX B
SOURCE BOOK

DAY 1

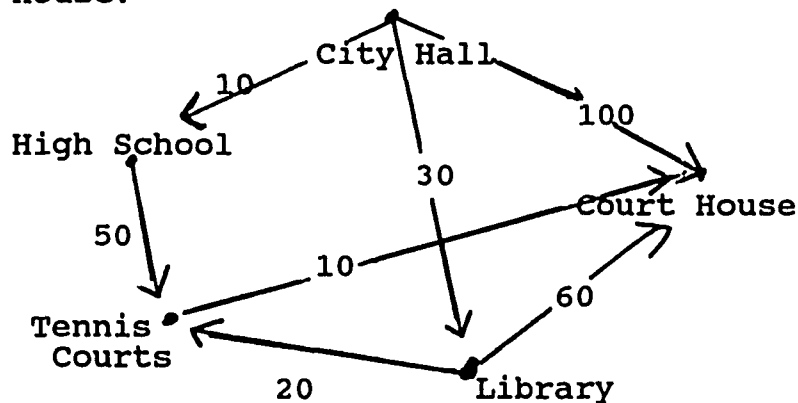
Introduce some problems in graph theory.

1. Is it possible in a group of 5 people that each person has exactly 3 friends? The picture drawn to determine whether there is a solution is a graph.



Use the trial and error method. What happens if there are 6 or 7 people in the group?

2. What is the shortest path from City Hall to the Court House?

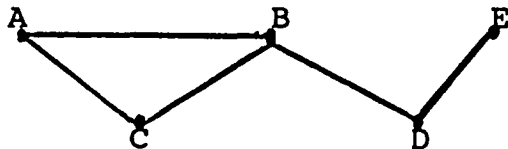


Graph theory will help us to answer such questions with the use of theorems and algorithms.

A graph $G = (V, E)$ consists of a set V of vertices and a set E of edges. Vertices are also called nodes and points while edges are also called arcs, lines, and branches. Each edge in E connects two distinct vertices in V . Harary states "We believe that uniformity in graphical terminology will never be attained, and is not necessarily desirable." (Harary, 1972, p.8)

The vertices in the graphs below are represented by dots and the edges by lines.

Graph 1

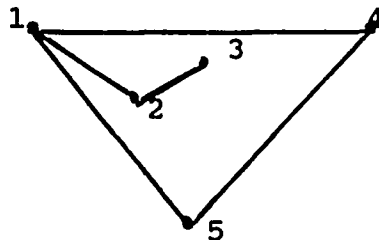


Graph 2



In graph 1, $V = \{A, B, C, D, E\}$ with $E = \{(A, B), (A, C), (B, C), (B, D), (D, E)\}$. The two vertices A and B are adjacent since there exists an edge (A, B) connecting them. The edge (B, D) is incident to vertex B since B is an endpoint of the edge (B, D) . Capital letters are used throughout this text for vertices. Graph 1 is an example of a connected graph while Graph 2 is not, since there is no path between A and C .

Does the drawing of graph 1 above differ in any way from Graph 3?



Graph 3

Two graphs can depict the same ideas if we can match up each edge and vertex preserving adjacency.

A	4		
B	1		
C	5	and	(A, B) corresponds with $(4, 1)$
D	2		(B, C) corresponds with $(1, 5)$
E	3		(B, D) corresponds with $(1, 2)$
			(D, E) corresponds with $(2, 3)$
			(A, C) corresponds with $(4, 5)$

What methods should be used to determine if the graphs are isomorphic? Two graphs G and H are isomorphic if there exists a one-to-one correspondence between their set of vertices which preserves adjacency.

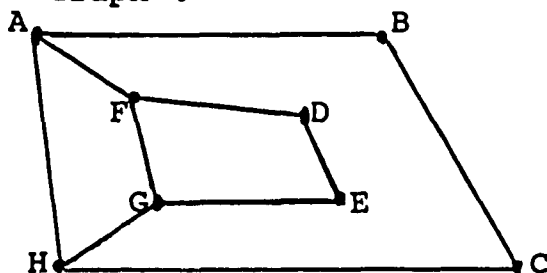
An example of one-to-one correspondence which does not preserve adjacency may need to be given.

Idea 1 In the graphs 1 and 3, the number of vertices are equal. Since vertices 1 and B each have 3 edges incident on them, those two vertices should correspond. The degree of a vertex is the number of edges in the graph that are incident to that vertex. Therefore in an isomorphism, vertices with the same degree must correspond.

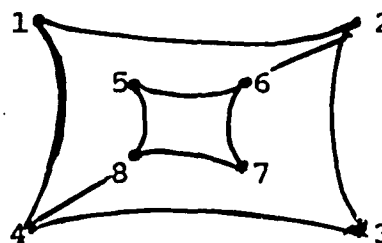
Why is it important to determine if two graphs are isomorphic? Chemical researchers keep a dictionary of graphs of all known molecular compounds. If the chemist has discovered a "new" compound, he/she must test the graph theoretic structure against the set of known compounds. Also, in designing efficient electrical circuits, engineers use isomorphic graphs. If a problem has been solved for an isomorphic network, the engineer will save time and money by using this solution. (Tucker, 1984)

Are these two graphs isomorphic?

Graph 4



Graph 5



Each graph has 8 vertices, 4 with degree 2, 4 with degree 3. But it is difficult to find a matching. In Graph 4, there exists a subgraph of 4 vertices each with degree 3, where there is no such subgraph in graph 5.

A subgraph G' of $G=(V,E)$ is a graph $G'=(V',E')$ where $V' \subseteq V$ and $E' \subseteq E$ where each edge in E' is incident with vertices in V' .

Theorem If two graphs are isomorphic, their corresponding collections of subgraphs are isomorphic.

Since a statement and its contrapositive always hold the same truth value, the contrapositive for the above statement is very helpful. Knowing this fact is useful in answering questions on SAT or GRE exams.

Review on truth of logic statements

Statement If $x = 4$, then $x^2 = 16$.

Converse If $x^2 = 16$, then $x = 4$.

Inverse If $x \neq 4$, then $x^2 \neq 16$.

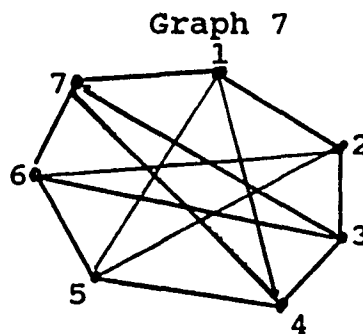
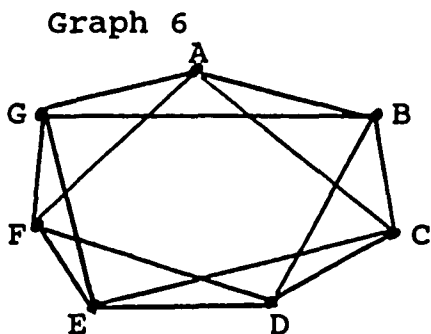
Contrapositive If $x^2 \neq 16$, then $x \neq 4$.

The converse and inverse are not always true, yet the statements and its contrapositive always have the same truth value.

Form the contrapositive of the theorem.

Idea 2 If a subgraph of graph A is not isomorphic to any subgraph of graph B, then graph A is not isomorphic to graph B.

Are these graphs isomorphic? (Example from Tucker, 1984). Inform the students that vertices occur only where points are labelled. Intersections of two edges do not necessarily constitute a vertex.



Since each graph has 14 edges, 7 vertices with degree 4, we try to construct the isomorphism. Matching vertex A with 1, let us look at the subgraphs formed by A and 1. One subgraph is the path A F G B C. (a set of vertices all adjacent to A) A path from 1 is 1 7 4 5 2 (a set of vertices all adjacent to 1) The matching must make the two path subgraphs isomorphic. F and C must match with either 7 or 2. Then G and B match with 4 and 5.

A	1
B	5
C	2
D	
E	
F	7
G	4

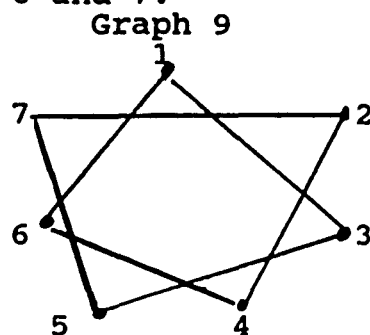
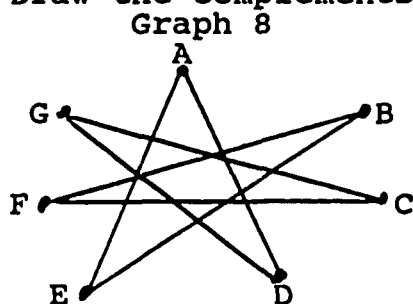
D, E and 3, 6 are the vertices not matched. Since D is adjacent to B and not A and 6 is adjacent to 5 but not to 1, match D with 6 and E with 3.

DAY 2

Idea 3 Another method for determining if two graphs are isomorphic is useful when the graph is dense rather than sparse. A dense graph has more pairs of vertices joined by edges than pairs not joined by edges. The complements of each graph will have fewer edges and be simpler to analyze.

A complement is a graph that joins every two vertices by an edge that does not exist in the original graph and removes the edges of the original graph.

Draw the complements of Graphs 6 and 7.



Is there an isomorphism between graphs 8 and 9?

Notice in graph 8, there exists a twisted circuit that passes through every vertex. A D G C F B E A. A circuit is a path whose point of origin is also its final point. There also exists a circuit in graph 9, 1 3 5 7 2 4 6 1. That correspondence gives an isomorphism between the 2 graphs.

What happens if we can't use any of these methods? (Ideas 1, 2 and 3) To prove that 2 graphs are not isomorphic, we must show that some vertex in graph A cannot correspond to any vertex in graph B. If each graph has n vertices, the exhaustive search would have n possibilities to match the first vertex in graph A, $n-1$ possibilities to match the second vertex in graph A, $n-2$ possibilities to match the third vertex, etc. or $n!$ Even if the number of vertices is only 25, $25! = 15,511,210,043,330,985,984,000,000$.

With our present theorems and definitions, we have enough information to prove a theorem necessary to answer the first question in the beginning of the chapter. Is it possible to have a group of 5, 6, or 7 people in a group such that each person knows exactly 3 others?

What is the relationship between the sum of the degrees of all vertices and the number of edges?

Sum of degrees	Edges	
10	5	Graph 1
10	5	Graph 2
28	14	Graph 6
28	14	Graph 7
14	7	Graph 8

$$\sum_{i=1}^n \text{degrees}(V_i) = 2 \cdot E$$

Theorem The sum of degrees of all the vertices is equal to twice the number of edges.

This is shown since each edge contributes a count of 2 units for the sum degrees.

What does this say about the number of vertices with odd degree in a graph?

Since the sum of degrees of all the vertices must be even ($2E$), and the sum of degrees of vertices with even degree is even (an even number times any number is even), what must be left is even.

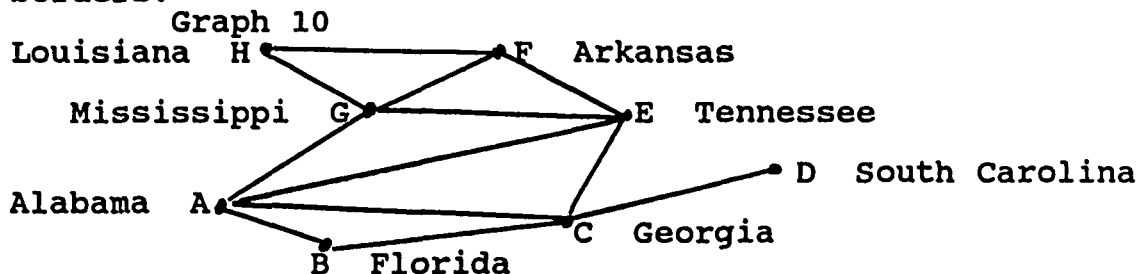
Corr. The number of vertices of odd degree must be even.

In finding a solution to our problem, if a graph has 5 vertices, the sum of the degrees must be even. So 5 vertices cannot each have degree of 3. The same would hold true for 7 vertices. This theorem leaves open the possibility for a group of 6 people to have exactly 3 friends.

Computer representation of graphs

What is necessary to represent a graph for the computer? it should be elicited from the students that the vertices and the edges are needed; the placement of vertices is not significant since any two graph drawings are isomorphic.

The following is a graph of the southeastern section of the United States showing which states have common borders.



If we represent every state by a letter, with which data structure are we familiar to show that two states have a common border? A two dimensional matrix called the adjacency matrix is appropriate. A pedagogical approach is to give this partially correct matrix and ask the students to check the validity. Should it be symmetric?

	A	B	C	D	E	F	G	H
A	0	1	0	0	0	0	1	0
B	1	0	1	0	0	0	0	0
C	0	1	0	1	0	0	0	0
D	0	0	0	0	1	1	1	0
E	1	0	1	1	0	1	1	0
F	0	0	0	0	1	0	1	1
G	1	0	0	0	1	1	0	1
H	0	0	0	0	0	1	1	0

The matrix has the property $\text{Graph}[i,j] = 1$ iff the edge (i,j) is in the graph and $\text{Graph}[i,j] = 0$ if there is no such edge. The space needed to represent the graph is V bits. The matrix is symmetrical around the diagonal, and for an undirected graph it is necessary to save only the upper triangle for large graphs. In undirected graphs, an edge i,j may also be referred to by j,i .

In Pascal, we are allowed to subscript the array by characters, we have chosen to do that to keep our picture graphs consistent with the type declarations.

What type do we need for the adjacency matrix?
`graphtype = array['A'..'H','A'..'H'] of boolean.` And if there is pertinent information to be kept about each vertex -
`vertextype = array[1..vertices] of vertexrecord.`

Disadvantages to this declaration must be discussed. What if the size of our graph varies? What if the size is greater than 26?

How do we read into the adjacency matrix?
 Each edge is read in by reading in the name of its endpoints. If we are reading in names of cities or states, they must be converted to a character subscript.

Aside: In setting up the graph for a program, how would one read in the name of a state and get a letter to correspond to it? Suggestion: Set up a table - an array of strings indexed by characters, 'A' to 'Z'. In function `Subscript`, each time a new state is read in (one that is not in the table), simply add it to the table subscripted by the next available character, and return that character. If the state is already in the table, have function `Subscript` return the character that is the index to that string.

```

Procedure Read( var graph: graphtype);
  Var i,j:char;
  vertexone, vertextwo:stringtype;{names of vertices
                                     e.g. cities, states, etc.}
begin
  For i:= 'A' to 'H' do
    for j:= 'A' to 'H' do
      Graph[i,j]:= false;

  for number:= 1 to edges do
    begin
      readln(vertexone, vertextwo);
      i:= subscript(vertexone);
      j:= subscript(vertextwo);
      graph[i,j] := true;
      graph[j,i]:= true;
    end;
end;

```

The nested for do loop requires $O(n^2)$ time where n is the number of vertices. Are there any disadvantages in using an adjacency matrix to represent a graph? For sparse graphs, it is not worthwhile to keep track of all the "zero" edges.

By using linked lists for the representation of the graph, operations such as finding adjacent vertices, adding a vertex, and determining the number of edges given a graph will take considerably less time. Notice, adding or deleting an edge between two existing vertices takes less time with the adjacency matrix. To keep the linked lists attached an array of pointers (a Comb) is sufficient.

```

Graph[A] → [B] → [C] → [E] → [G | nil]
Graph[B] → [A] → [C | nil]
Graph[C] → [A] → [B] → [D] → [E | nil]
Graph[D] → [C | nil]
Graph[E] → [A] → [C] → [F] → [G | nil]
Graph[F] → [E] → [G] → [H | nil]
Graph[G] → [A] → [E] → [F] → [H | nil]
Graph[H] → [F] → [G | nil]

```

The data structure for this in Pascal is an array of pointers

```
Type ptr = ^ node;
      node = record
          vertex:char;
          link:ptr;
      end;
      graphtype= array['A'..'Z'] of ptr;
Var   temp:ptr;
Reading into an adjacency list.
```

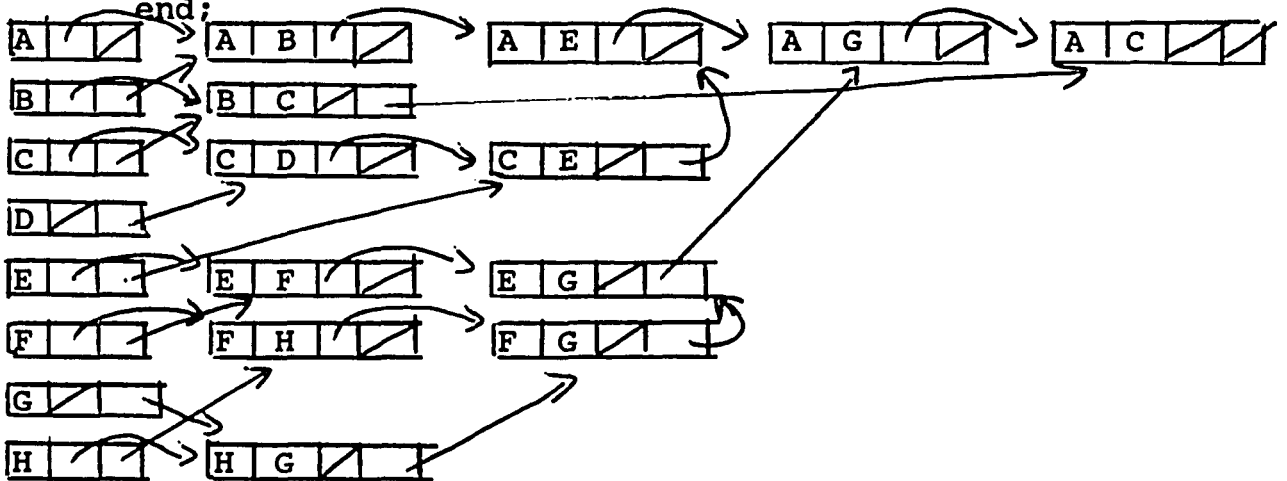
```
For i:= 'A' to 'H' do
    graph[i]:=nil;
For number := 1 to edges do (or while not eof do)
begin
    readln(vertexone, vertextwo);
    i:= subscript(vertexone);
    j:= subscript(vertextwo);
    new(temp);
    temp^.vertex:=j;
    temp^.link:= graph[i];
    graph[i]:=temp;
    new(temp);
    temp^.vertex:=i;
    temp^.link:=graph[j];
    graph[j]:= temp;
end
```

Determining the total number of edges in an adjacency matrix would be $O(n^2)$ where in an adjacency list, the time would be $O(n + e)$, n being the number of vertices and e the number of edges.

If there is information to keep about each vertex, the data structure may be changed to an array of nodes with several fields where the last field is a pointer to the first element in the linked list.

A third implementation for the graph is an Edge adjacency list where each edge is listed only once but there are two extra fields in each node, one keeping track of the second vertex in an edge and the other pointer is part of a linked list where the second field is always the second vertex.

```
Type ptr= ^node;
      node = record
        firstvertex,secondvertex:char;
        firstlink,secondlink:ptr;
      end;
```



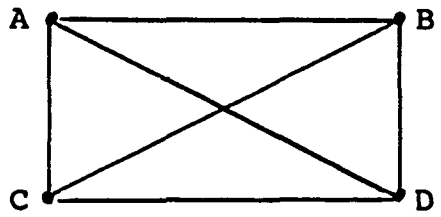
DAY 3

Planar graphs

It is cheaper to produce circuit boards if the circuits can be placed on a flat surface, with wires crossing only at connection points. There is a need to make graphs planar.

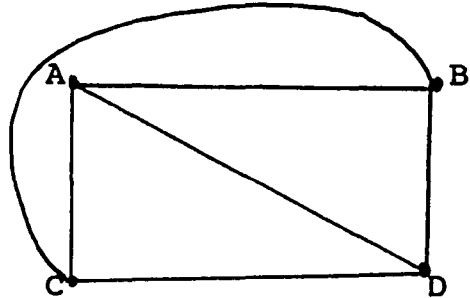
A former graduate student explained how circuits are made on the plastic circuit board with the copper etching process. Take a photographic image of the electrical network and project it onto the copper plate. Where there is no light the copper remains; the rest of the copper is etched away by acid. The copper remaining forms a graph. It is necessary for the electrical network to be planar in order for this to work and it is a cost effective method to produce circuit boards. It is fine electronically to use insulated wires to make the circuit in a non-planar fashion but it is very expensive to do so.

This graph has 2 edges crossing in the plane.
 Graph 11



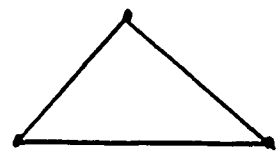
The graph is planar if it can be drawn on paper in such a way that no two edges cross.

Graph 12



We will be looking at systematic ways to draw graphs on a plane without the edges crossing.

Graph 13

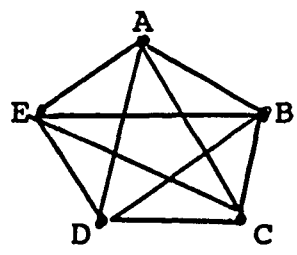


This graph is planar. It is called a complete graph since each vertex is adjacent to every other vertex. This is the K_3 graph.

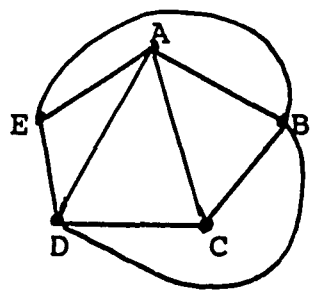
Is the K_4 graph planar? (See graph 12)

Is the K_5 complete graph planar?

Graph 14



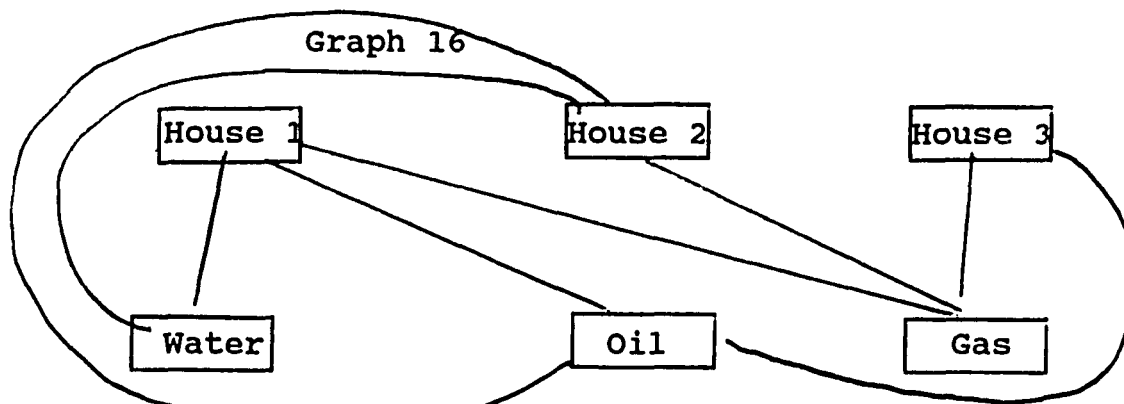
Graph 15



Which edge is missing? The best way to tell is to find the degree of each vertex.

It appears that it can not be drawn as a planar graph. Later discussion will prove such.

A problem. There are 3 houses to be built next to each other on a block, each house to be connected to a well, an oil tank, and a gas tank. Can the connections be made so that one pipe does not cross over the other? What would the graph look like?



No, House 3 cannot get any water.

This is a $K_{3,3}$ graph, an example of a bipartite graph. A bipartite graph consists of 2 distinct sets where edges of the graph are formed by connecting each point in the first set with a point in the second set. The English puzzle maker Henry Ernest Dudeney thought of this $K_{3,3}$ problem in 1917.

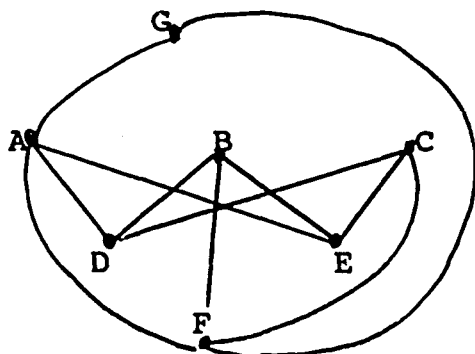
Gibbons (1985) states that there are a number of algorithms for testing planarity. Hopcroft and Tarjan developed an algorithm in $O(n)$ time. Gibbons shows a simpler yet efficient algorithm in his text due to Demoucron, Malgrange and Pertuiset.

In Frenkel's interview with Tarjan after the Turing Award Lecture, Tarjan responds "Yes the right representation of the input turns out to be very important in planarity testing because planar graphs are sparse." (Frenkel, 1987, p. 215) Papers in literature based graph algorithms on the adjacency matrix, but to check each entry, you are checking a quadratic size representation where many of the elements are zeroes. This drove computer scientists to use list based representation of graphs.

What are some ways of determining if a graph is planar?

Theorem (Kuratowski's) A graph is planar iff it does not contain a K_5 or $K_{3,3}$ configuration. (The word contain has an additional meaning besides the observation of a K_5 or $K_{3,3}$ configuration. There is a note to follow on homeomorphic graphs.)

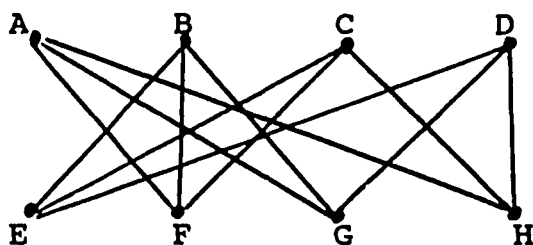
Graph 17



$K_{3,3}$ exists in it; so
it is not planar.
ABC - DEF

Problem Is this graph planar?

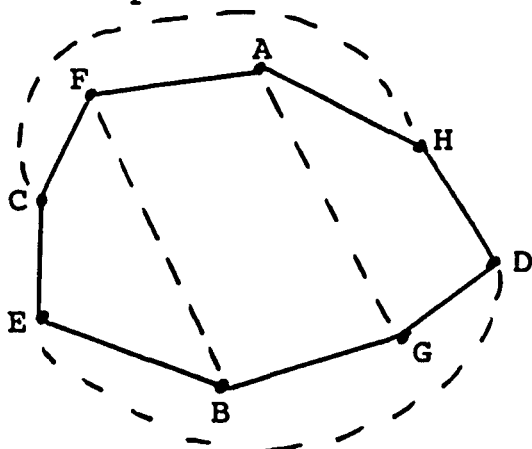
Graph 18



Try as you may, it
does not contain the
 $K_{3,3}$.

One method used to find if a graph is planar is to construct the longest circuit possible, A H D G B E C F on graph 19, and then draw as many edges inside. If it is not possible to draw an edge on the inside, then draw the edge on the outside.

Graph 19



Which edges are
missing?
(A,G), (F,B), (H,C),
(D,E)

The first two edges
may be placed on the
inside, while the
last two edges may be
drawn on the outside.

Conclusion: The graph is a planar graph.

The students may have guessed at Euler's (pronounced oiler) formula from the homework. It is surprising that the simple formula was not discovered by the Greeks. They had a great interest in mathematical properties of the regular polyhedra. Biggs, Lloyd and Wilson (1976) wonder if the discovery were lost or if the formula were missed because the Greeks concerned themselves with measurement, length, angles and areas. It wasn't until 1752 that Euler discusses $R = E - V + 2$ in a letter to his friend.

Euler's Theorem If G is connected planar graph, then $R = E - V + 2$. The proof is a nice example of induction on E .

1. Initial case $E = 1$

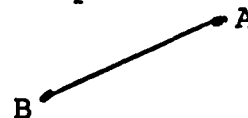
Graph 24



$$R=2, E=1, V=1$$

$$2 = 1 - 1 + 2$$

Graph 25



$$R=1, E=1, V=2$$

$$1 = 1 - 2 + 2$$

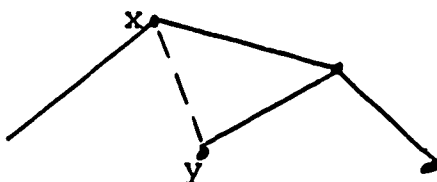
2. Assume the result is true for all connected planar graphs with k edges $R = E - V + 2$
3. Prove it true for $E = k + 1$ edges

Two cases

Case 1 Add one edge (x,y) between 2 existing vertices.

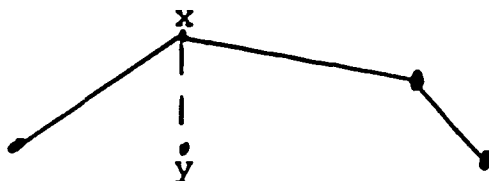
This adds one more edge to the graph and one more region. $R + 1 = E + 1 - V + 2$ and the equality still holds.

Graph 26



Case 2 Add an edge (x,y) where x is in the graph, and y was not in the graph previously. The number of regions remains the same, but the number of vertices and edges are increased by 1 so the equality is preserved.

Graph 27



Aside: For the astute student, it is necessary to point out that Graph 24 and any multigraphs (2 or more edges between 2 vertices) are extensions of the original definition of graph.

Problem How many regions would there be in a planar graph if there were 12 vertices each with degree 4? We need to know the number of edges. By a previous theorem, on page 6D, the sum of degrees $\sum V_i = 2 \cdot E$

$$\begin{aligned} \text{therefore, } 12 \cdot 4 &= 2 \cdot E \\ 24 &= E \end{aligned}$$

$$\begin{aligned} R &= 24 - 12 + 2 \\ &= 14 \end{aligned}$$

DAY 4

Euler's theorem has the following corollary that may be used to show a graph is non-planar.

Corollary In a connected planar graph, with $E \geq 3$, $E \leq 3 \cdot V - 6$.

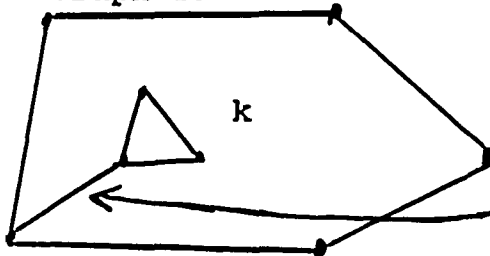
The Proof - First, define the degree of a region as the number of edges that bound the region.

Graph 28



inside and outside region
each have 4 edges

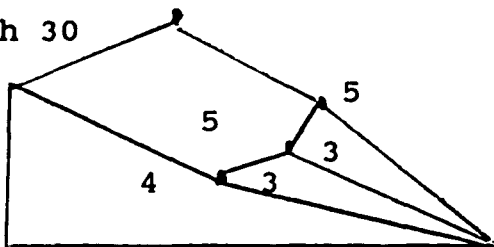
Graph 29



degree of region k is 10,
count that edge twice

Since $E \geq 3$, the smallest degree a region can have is 3. If there are R regions and each has a minimum degree of 3, then $3R \leq \sum \text{degrees of all regions}$.

Graph 30



How many regions are there? Is there any relationship between the regions and the number of edges? The sum of the degrees of the regions is 20 and the number of edges is 10. Therefore, the sum of the degrees of all regions is equal to $2E$, since the sum counts each edge twice (inside and outside). Hence, $3R \leq 2E$.

And from Euler's theorem $R = E - V + 2$.

$$\begin{aligned} 3(E-V+2) &\leq 2E \\ 3E - 3V + 6 &\leq 2E \\ E &\leq 3V - 6 \end{aligned}$$

We use this corollary to prove that K_5 is not planar. Since a statement and its contrapositive always hold the same truth value, the contrapositive of the corollary is also true. In a connected graph if $E > 3V - 6$, then the graph is not planar.

In the K_5 complete graph $E = 10$, $V = 5$.

$$10 > 3 \cdot 5 - 6$$

$10 > 9$, so the graph is not planar.

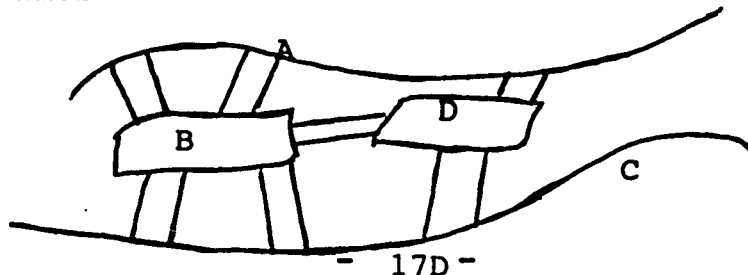
Note the converse of the corollary is not true. If $E \leq 3V - 6$, the connected graph is planar.

The $K_{3,3}$ graph has $E = 9$, $V = 6$. $9 \leq 3 \cdot 6 - 6$ which is true, but the $K_{3,3}$ graph is not planar.

Covering Graphs

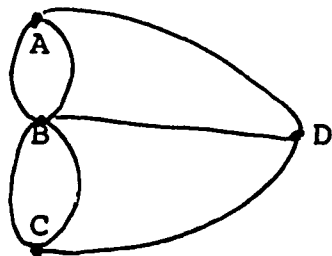
In 1736, a mathematician Leonhard Euler was presented with the following problem. In the city of Königsberg in East Prussia, there were seven bridges in the Pregel River connecting 2 islands to each other and mainland. The townspeople wondered if they could begin their Sunday walks from a starting point, cross each bridge exactly once and return to the point of origin. It didn't matter that you walked across an island several times.

Graph 31



A dual graph is formed by allowing each piece of land to be a vertex and a bridge to be an edge.

Graph 32



This graph is called a multigraph, since there are 2 edges between 2 adjacent vertices. Multigraphs are useful in the study of chemical bonding.

How would the implementation for our graph change for a multigraph since there may be two edges between 2 vertices? The adjacency matrix could be a two dimensional array of integers where the integer values could be the number of edges. For the adjacency list, an extra field could be held in each node.

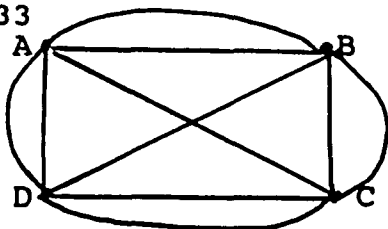
In the problem posed to Euler, no circuit can be formed that traverses all the edges in the graph exactly once and returns to the starting point.

What if anything added to the graph would give an Euler circuit? Today an eighth bridge between the 2 islands exist and forms an Euler path. The Euler path traverses each edge exactly once but does not necessarily return to the point of origin.

The difficult part about forming an Euler circuit in the Seven Bridges problem is the 3 edges on one vertex.

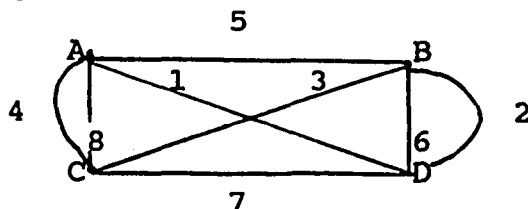
In graph 33, can an Euler circuit be formed? Can you start at a point and continue following a path without lifting your pencil and return to the point of origin? (As on the placemats of Howard Johnson's Restaurants)

Graph 33



Notice that 4 vertices have odd degree. **Students will have difficulty, since there is no Euler circuit.

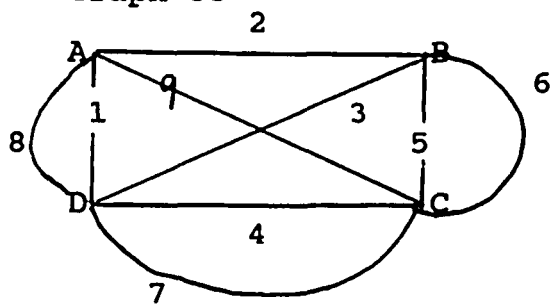
Can an Euler circuit be given for this graph?
Graph 34



The numeration (1,2,3,4,5,6,7) demonstrates a possible path.

Euler's Theorem: If G is an undirected or multigraph, then it has an Euler circuit iff G is connected and every vertex in G has even degree.

Can an Euler circuit be given for this graph?
Graph 35



With just 2 vertices of odd degree, you can have an Euler path.

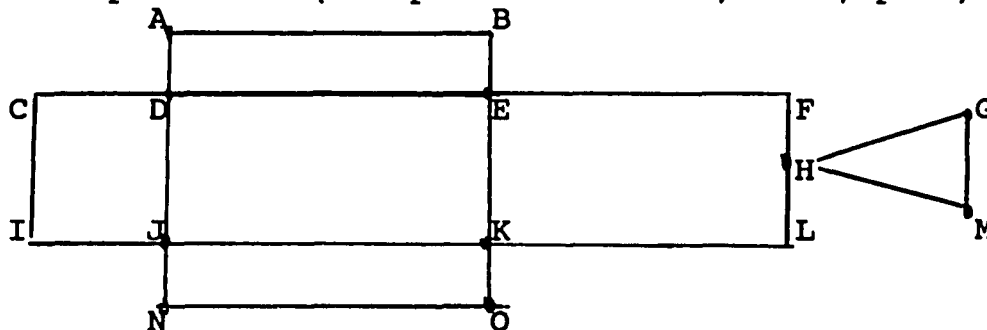
Corollary. If G is an undirected or multigraph, the graph has an Euler path iff it is connected and has 2 vertices of odd degree.

The Euler path begins at one of the vertices of odd degree and ends at the other.

Gibbons (1985) gives pseudo code for finding Euler circuits in $O(e)$ time.

Fleury designed an algorithm to determine Euler paths and circuits. In building an Euler path, never choose an edge whose erasure will disconnect the resulting graph of remaining edges.

Graph 36 (example from Tucker, 1984, p.53)



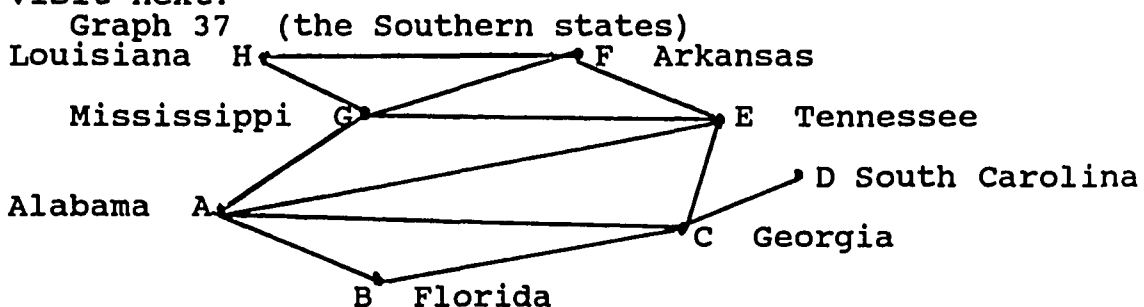
As you mark an edge erase it, AB(1), BE(2), ED(3), DA could not be done now, DC(4), CI(5), IJ(6), (Drawing JD and DA at this time would leave remaining edges,) JN(7), NO(8), OK(9), KL(10), LH(11), HM(12), MG(13), GH(14), HF(15), FE(16), EK(17), KJ(18), JD(19), DA(20).

The Chinese Postman Problem finds the shortest tour such that each edge of a graph is traversed at least once. (Euler circuits traverse every edge exactly once). It is the problem faced by a postman who must deliver mail along each edge of a graph and return to its starting point.

DAY 5

Traversing Graphs

Every time we learn a new data structure, we discuss which operations to use. We traversed binary trees using preorder, inorder and postorder methods. Now how will we traverse the undirected graph? Is there a starting point? If given a starting point, which state should we visit next?



The following is a nice example given for homework that aids in understanding the difference between breadth first and depth first traversal.

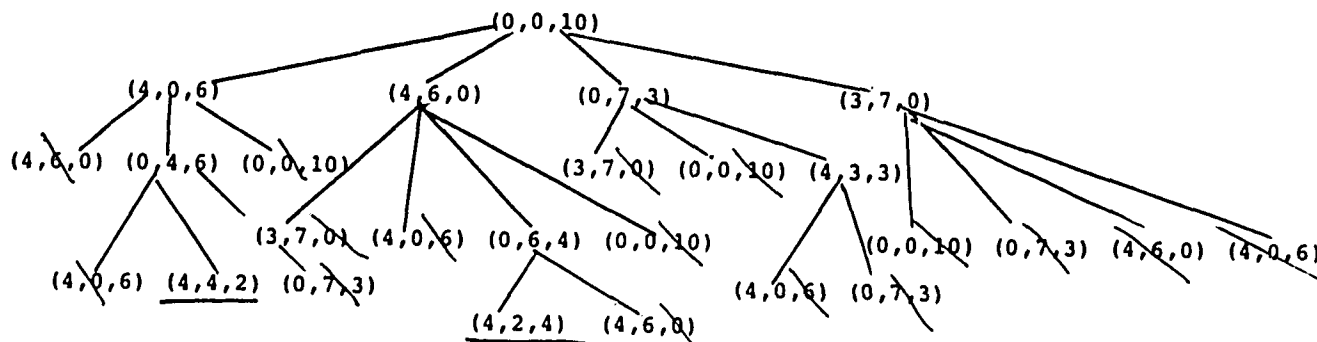
Suppose we are given 3 pitchers of size 10 quarts, 7 quarts, and 4 quarts. Initially the 10 quart pitcher is full and the other 2 are empty. We can pour from one pitcher into another pouring until the receiving pitcher is full or the pouring pitcher is empty. Is there a way to pour among pitchers to obtain exactly 2 quarts in the 7 or 4 quart pitcher. If so, find a minimum sequence of pourings to get two quarts. (Tucker, 1984)

A student's first approach is usually a depth first search. Use a three parameter point to show the amount in each pitcher. (4,7,10) are the possibilities

(0,0,10) (4,6,0) (4,0,6) (0,4,6) (0,7,3)
 (3,7,0) (4,6,0) (0,6,4)

The students may not realize the next step, and see a problem with using a depth first search.

But a breadth first approach guarantees an answer and also finds the minimum sequence.



A breadth first traversal visits all successors of a visible node before visiting any successors of those successors. (Tenenbaum and Augenstein, 1986)

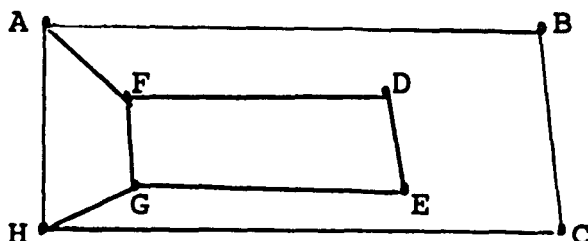
From graph 37, the breadth first search from Florida may be Florida, Georgia, Alabama, South Carolina, Tennessee, Mississippi, Arkansas, Louisiana.

There are different but correct breadth and depth first traversals. It will depend on how the adjacency list was created and how the code is written. Depending on the class, there may be a need to show the breadth first and depth first searches for several graphs.

Relative merits of depth-first search and breadth-first search should be mentioned. Depth-first is more efficient when there are many roughly equally long paths from the start state to goal state, and no path leads off into a tremendously long, futile search. Breadth-first search is always safer when there is a danger of long, futile pathways in the lower reaches of the exhaustive search tree; and it is more efficient when the shortest path from start state to goal state is sought. If you use our original adjacency list (p.8D)

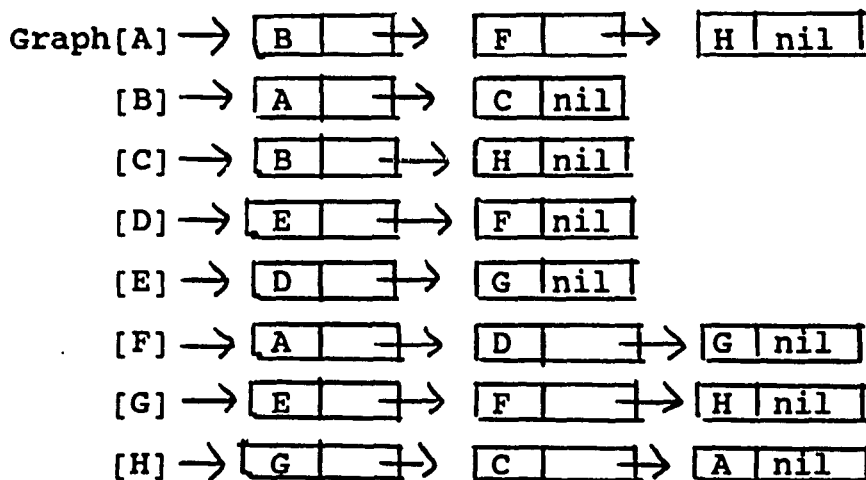
for the southern states, a stack will give you a correct breadth first traversal. This of course will not work all the time.

Use Graph 4 and its adjacency list to demonstrate that the FIFO queue is needed for breadth first traversals.



A breadth first traversal would be ABFHCDGE. How do we know what vertices to visit? (Answer: Use the adjacency list to know the adjacent vertices.) After we visit B, F, H how do we get to C? We have to go back to B and get any of its adjacent vertices. How do we get back to B? How shall we hold it in memory? If a student responds stack, demonstrate why the stack does not work. Which abstract data type is appropriate? We must go back to the first one in. FIFO - The fifo queue! Demonstrate how the queue works "manually".

We need the adjacency list for graph 4.



How will we remember which nodes we visited?

An extra array called marked is kept which will order the vertices when visited. Initially the marked array will be given the value of zero. When the element is put on the queue, it will be marked with a -1 and when visited the marked element will be given a positive integer.


```

1 Procedure Breadthfirstsearch(graph:graphtype;
                               point:char);
2 Type markedtype = array['A'..'Z'] of integer;
3 Var id:integer;
4     marked:markedtype;
5     queue:queuetype;
6     k:char;

7 Procedure visit(graph:graphtype;
                  var marked:markedtype; point:char);
8     Var temp:ptr;
9     begin
10        enqueue(queue, point);
11        while not emptyqueue(queue) do
12            begin
13                remove(queue, point);
14                id:= id + 1; marked[point]:= id;
15                temp:= graph[point];
16                while temp <> nil do
17                    begin
18                        if marked [temp^.vertex]= 0
19                            then begin
20                                enqueue(queue, temp^.vertex);
21                                marked[temp^.vertex]:= -1;
22                            end;
23                        temp:= temp^.link;
24                    end;
25                end;
26        end;

27 begin
28     id:=0;
29     clearqueue(queue);
30     for k:= 'A' to 'H' do
31         marked[k]:= 0;
32     visit (graph, marked, point);
33 end;

```

A trace table for Breadth first search from 'A'

Marked	[A]	=	0	1	ID	temp	point	Queue
	[B]	=	0	1	2	^B	'A'	
	[C]	=	0	1	5	^F		
	[D]	=	0	1	6	^H		
	[E]	=	0	1	8	nil		rear front
	[F]	=	0	1	3	^A	'B'	
	[G]	=	0	1	7	^C		E G D C H F B A
	[H]	=	0	1	4	nil		
					6	nil		
					7	^A	'F'	
					8	^D		
						^G		
						nil		
						^G	'H'	
						^C		
						nil		
						^B	'C'	
						^H		
						nil		
						^E	'D'	
						^F		
						nil		
						^E	'G'	
						^F		
						^H		
						nil		
						^D	'E'	
						^G		
						nil		

What changes in the code if the breadth first traversal begins from a vertex other than 'A'?

What if there are more than 26 vertices in the graph?

What if the graph isn't connected? As in graph 2?

```
Change line 32 to      For k:= 'A' to 'H' to
                        if marked [k] = 0
                        then visit(graph, marked,k);
```

Time complexity

In breadth first traversal, we are actually visiting every vertex and each of its edges. In our adjacency list implementation of graphs, we may be visiting each node twice though that doesn't have any effect on the order of the complexity. The time complexity is then $O(n + e)$, where n is the number of vertices and e , the number of edges. Since usually the number of edges is greater, we may say the efficiency of the bfs algorithms is $O(e)$.

If the adjacency matrix was used, we would have traversed the entire matrix and the algorithm would be $O(n^2)$.

DAY 6

Depth First Traversal

In graph 37, what is the depth first traversal from Florida?
from Louisiana?

Realize that there are a few.

If we take a certain path, we must backtrack to the last spot where the path was chosen. On the graph of the southern states, the depth first traversal could be A, E, D, C, B. Backtrack to the last point where you made a choice - E and take another path from there, F, G, H. What abstract data type allows for backtracking? The stack! Or allow recursion to do the backtracking for us. The students should be shown how the stack works before tracing the code.

```

1  Procedure depthfirst(graph:graphtype; point:char);
2  Var id:integer;
3     k:char;
4     marked:markedtype;

5  Procedure visit(graph:graphtype;
6     var marked:markedtype; point:char);
7  var temp:ptr;
8  begin
9     id:= id + 1; marked[point]:=id;
10    temp:= graph[point];
11    while temp <> nil do
12       begin
13          if marked [ temp^. vertex] = 0
14             then visit(graph, marked, temp^.vertex,);
15          temp:= temp^.link;
16       end;
17    end;

18  begin
19     id:= 0;
20     for k:= 'A' to 'H' do
21        marked[k]:=0;
22     for k:= 'A' to 'H' do
23        if marked[k] = 0 then visit(graph, marked, k);
24    end;

```

How would you change the code to output the vertex visited instead of numbering the vertices?

Add to line 8 writeln(table[point]) after passing in table.

Trace DFS using recursion on Graph 4 and same adjacency list.

THIS IS NOT A VERY INTERESTING EXAMPLE FOR RECURSION. SEE NEXT EXAMPLE. The examples chosen for class must be planned so that the algorithm is demonstrated well.

```
Marked[A] = 0 1
        [B] = 0 2
        [C] = 0 3
        [D] = 0
        [E] = 0
        [F] = 0
        [G] = 0 5
        [H] = 0 4
```

```
k='A'
visit(graph, marked, 'A')
  id = 1
  point = 'A'
  temp = ^B
  visit(graph, marked, 'B')
    id = 2
    point = 'B'
    temp = 'A'
    temp = 'C'
    visit(graph, marked, 'C')
      id = 3
      point = 'C'
      temp = 'B'
      temp = 'H'
      visit(graph, marked, 'H')
        id = 4
        point = 'H'
        temp = 'G'
        visit(graph, marked, 'G')
          id = 5
          point = 'G'
          temp = 'E'      and so on
```

The trace for recursive DFS on the Graph of the Southern States p.6D and its adjacency list p.8D.

```
Marked[A] = 0 1
        [B] = 0 2
        [C] = 0 3
        [D] = 0 4
        [E] = 0 5
        [F] = 0 6
        [G] = 0 7
        [H] = 0 8
```

Actually, on the blackboard it is good to show the completion of the recursive procedures by erasure.

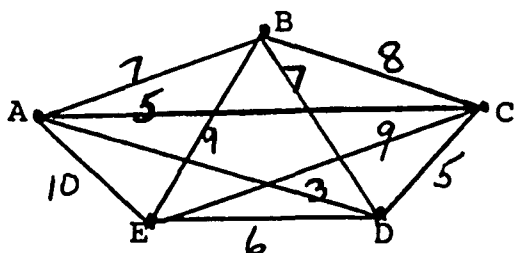
```
k='A'
visit(graph, marked, 'A')
  point = 'A'
  id = 1
  temp = ^B
  visit(graph, marked, 'B')
    point = 'B'
    id = 2
    temp = ^A
    temp = ^C
    visit (graph, marked, 'C')
      point = 'C'
      id = 3
      temp = 'A'
      temp = 'B'
      temp = 'D'
      visit(graph, marked, 'D')
        point = 'D'
        id = 4
        temp = 'C'
        temp = nil
        temp = 'E'
        visit(graph, marked, 'E')
          point = 'E'
          id = 5
          temp = 'A'
          temp = 'C'
          temp = 'F'
          visit(graph, marked, 'F')
            point = 'F'
            id = 6
            temp = 'E'
            temp = 'G'
            visit(graph, marked, 'G')
              point = 'G'
              id = 7
              temp = ^A
              temp = ^E
              temp = ^F
              temp = ^H
              visit(graph, marked, 'H')
                point = 'H'
                id = 8
                temp = ^F
                temp = ^G
                temp = nil
              temp = nil
            temp = ^h
            temp = nil
          temp = ^G
          temp = nil
        temp = nil
      temp = nil
    temp = nil
  temp = 'C'
  temp = 'E'
  temp = 'G'
  temp = nil
```

Time Complexity

In the depth first search how does the recursion affect the efficiency? We still only look at each vertex and its corresponding edge once. When the recursive procedure returns to a particular adjacency list, the recursion held the place of the last edge visited. This keeps the algorithm with a time complexity of $O(n+e)$, or $O(e)$ since the number of edges is usually greater. The trace table for the recursion demonstrates the recursion holding the place of the last edge visited on a particular adjacency list.

Hamilton Circuit

Graph 38



Notice it is a K_5 complete graph

If a salesman lives in City A and must travel to every other city exactly once and return home, what is the minimum cost of the trip?

The greedy algorithm starts at A and visits the next city that is the least expensive to visit.

$$\begin{array}{cccccc} \text{AD} & \text{CD} & \text{BC} & \text{BE} & \text{AE} & \text{ADCBEA} \\ 3 & + & 5 & + & 8 & + & 9 & + & 10 & = & 35 \end{array}$$

Is this the cheapest way? What about ABDECA?

$$7 + 7 + 6 + 9 + 5 = 34$$

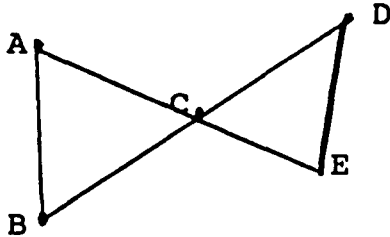
Have the students find a path with cost less than 34. [It is important to realize that this is a realistic question for cost, not for distance. Triangle AED does not satisfy the triangle inequality property that the sum of two sides of a triangle must be greater than the third side.]

Is there a method to find the cheapest way?

"Unfortunately, there is no efficient algorithm to solve the travelling salesman problem. To do so, a computer would have to examine all possible routes and there are $(n-1)!$ possible routes if n cities are involved. Thus if $n=30$ cities, the computer would have to check the costs of $29! \approx 8.8418 \times 10^{30}$ possibilities, an impossible task even for a computer." (Dierker, Voxman, 1986, p.31) Note: Since the Graph 38's distance from vertex V_1 to V_2 is the same and V_2 to V_1 , the number of routes is divided in half. $(n-1)!/2$

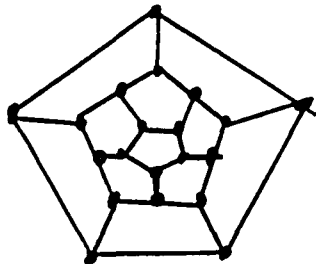
The Travelling Salesman problem is an example of the Hamilton Circuit. In 1859, the Irish mathematician Sir William Hamilton (1805-1865) developed a game that he sold to a Dublin toy manufacturer. The game consisted of a wooden regular dodecahedron with 20 vertices labelled with the names of prominent cities. The object of the game was to start at a city, visit every city only once and return to the point of origin. (You don't have to pass through every edge) (Biggs, Lloyd, Wilson, 1976)

Graph 39



This does not have a Hamilton circuit since you would have to visit C twice.

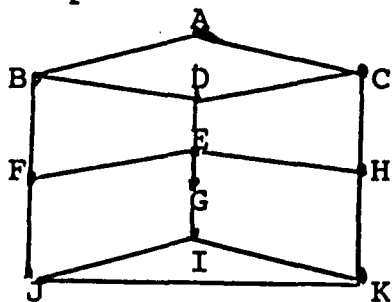
Hamilton's game "Around the World"
Graph 40 for homework



There are a few rules to determine if a graph has a Hamilton circuit. (Tucker, 1984, p.58)

- 1) If a vertex has degree 2, then both edges incident at x must be part of any Hamilton circuit.
Why? If one enters a city, one must also leave.
- 2) No proper subcircuit, that is, a circuit not containing all vertices, can be formed when building a Hamilton circuit.
Why? One would be visiting a city twice before visiting all other cities.
- 3) Once the Hamilton circuit we are building has passed through vertex x , all unused edges incident at x may be deleted.
Why? Since the vertices cannot be used later in the circuit - that would mean visiting a city twice.
- 4) If the graph is symmetric at a point, either edge may be chosen.

(Example from Tucker, 1984, p.59)
Graph 41

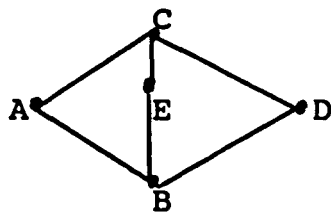


Find the Hamilton circuit from A.

Vertices In the circuit	The move	Rule
A, B, G I, E, C	1. AC, AB, GE, GI must be in the circuit.	#1
J	2. Since GI is in circuit, either JI or IK must be in the circuit. The graph is symmetric; choose JI. Delete IK	#1, #3, #4
K, H	3. KH and JK must be in the circuit	#1
	4. FJ must be deleted.	#3
F	5. F now has degree 2, so BF and FE must be in the circuit.	#1
	6. Since there are two edges incident to E, EF and EG must be in, eliminate EH and DE.	#3
	7. In order to visit D, DB and DC must be in the circuit, but that is impossible for it would form a subcircuit. ABCD	#2

All this work was not in vain, for a Hamilton path was formed. ABFEGIJKHCD. A Hamilton path visits every vertex once, but does not return to the point of origin.

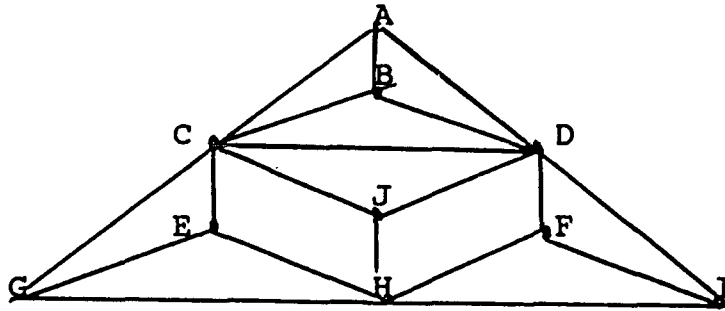
Graph 42



A Hamilton path is ABECD.

- Rules
1. BA, CA
CE, EB
DB, DC must be in the circuit for A, E, D all have degree 2 #1
 2. Subcircuit ABEC is formed #3

Graph 43 for Homework



Another real world example of a Hamilton circuit is the postal van that must pick up from n mail boxes each day and return to the post office. Minimizing the distance between the mailboxes is similar to the Travelling Salesman problem.

Also, consider a robot arm that must tighten the nuts on a piece of machinery. The position of the nuts can be represented by vertices of a graph. The arm must visit every nut exactly once and return to the starting point. The path of the arm is a Hamilton circuit.

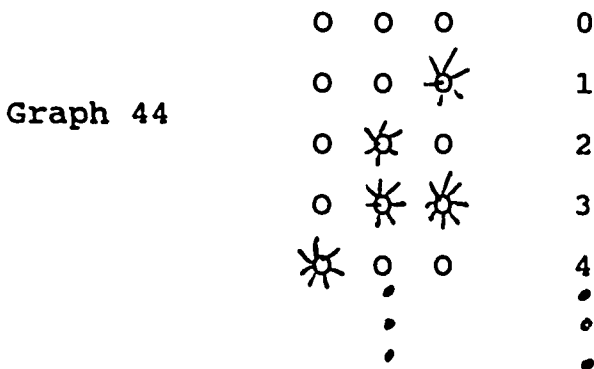
DAY 7

Gray Code

A 3 bit Gray code is a sequence of 2^3 3 bit binary strings such that the i th bit string differs from the $(i+1)$ st in exactly one bit, for $1 \leq i \leq 2^3$.

The binary numbers from 0 to 7 are not an example of Gray code, (000, 001, 010, ...) since 2 bits change in moving from 1 to 2. In the Gray code, one wants to switch one light off or one light on between numbers. Using the idea of light on and off is a clever way of demonstrating 1 and 0 in the binary number system.

Graph 44 is an example of binary code, not Gray code

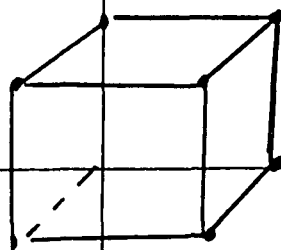


The following list of numbers satisfies the property of the Gray code, but realize there is no sequence to it.

001	1
011	2
111	7
101	5
100	4
110	6
010	2
000	0

One way to visualize the 3 bit Gray code is a cube in 3 space with a Hamilton circuit.

Graph 45



Find a Hamilton circuit starting from $(0,0,0)$

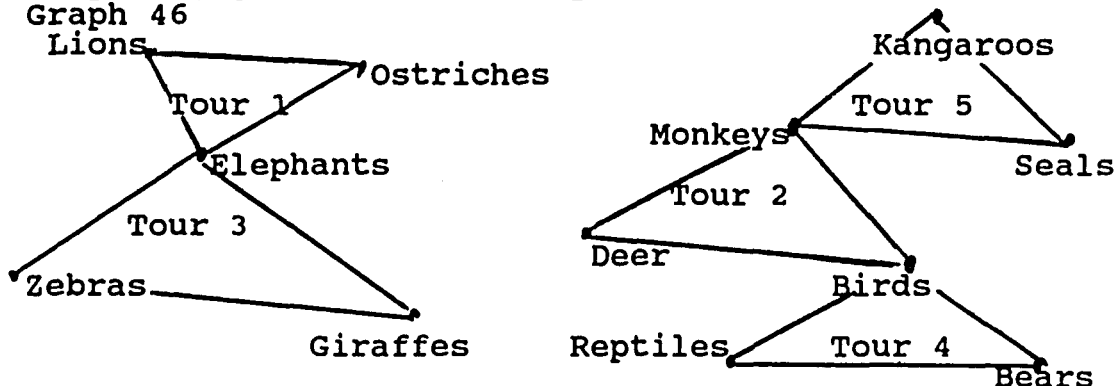
$(0,0,0)$	$(1,0,0)$	$(1,1,0)$	$(0,1,0)$
$(0,1,1)$	$(1,1,1)$	$(1,0,1)$	$(0,0,1)$
$(0,0,0)$			

Why is touch tone dialing based on Gray code? Why is it beneficial to go from value to value so that only 1 bit is switched? In order to count on the computer without mistakes, it is not possible to do two things exactly at the same time, as change two bit values. We need 12 tones for touch tone so that the Gray code must have 4 bits; increase it to 15 numbers and the picture is 4 space.

Graph Coloring

A local zoo wants to take visitors on animal feeding tours, and has hit upon the following tours. Tour 1 visits lions, elephants, and ostriches, Tour 2 the monkeys, birds, and deer, Tour 3, the elephants, zebras, and giraffes, Tour 4 the birds, reptiles and bears, Tour 5 the kangaroos, monkey and seals. If animals should not get fed more than once a day, can these tours be scheduled using only Monday, and Wednesday. (adapted from Roberts, 1976, p.167) Reminder: the tours cannot visit an animal twice in one day since they only get fed once a day.

Graph 46



Remember the elephants are in Tours 1 and 3 and they cannot be fed twice in one day.

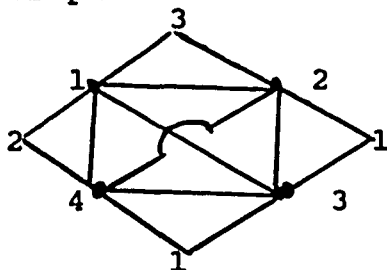
Answer: Yes, Tours 1, 4, and 5 may be scheduled on Monday, and tours 2 and 3 scheduled on Wednesday.

If Tour 4 were to include the seals, would the two days be sufficient for all the tours?

No, Tours 4 and 5 would have to be held on a separate day.

This problem is an example of chromatic numbering of a graph. The chromatic number is the minimum amount of colors you can use so that no two adjacent vertices have the same color. Start numbering the vertices so that no two adjacent vertices share the same number.

Graph 47

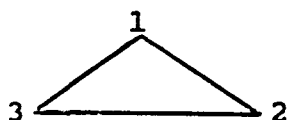


The chromatic number is 4.

A good method for determining the chromatic color is to find the largest complete graph and color each of the vertices a different color since the vertices are each connected to one another.

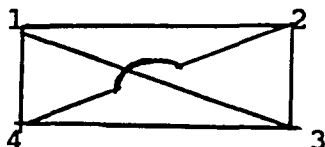
The triangle is the K_3 complete graph with chromatic color of 3.

Graph 48

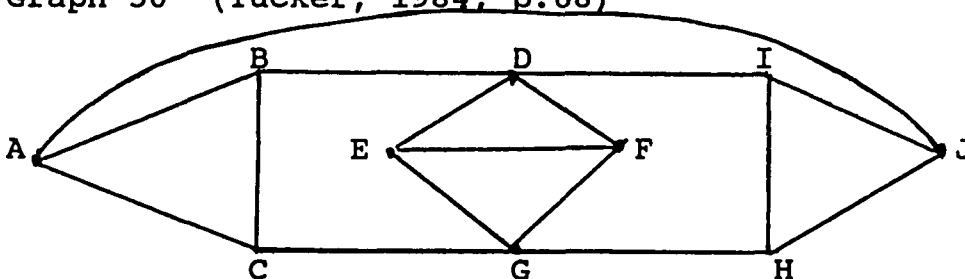


The K_4 complete graph has 4 for its chromatic color as shown in graph 49.

Graph 49



Graph 50 (Tucker, 1984, p.68)



The largest complete graph is K_3 . Triangle DEF forces a color on another vertex. G cannot be the same as E and F. Color D, 1; E, 2; F, 3; and this forces G to be 1. Since B and C are each adjacent to a vertex of color 1 and are adjacent to each other, they must have colors 2 and 3. Since the graph is symmetric, choose B to be 2 and C 3. This forces the color of A to be 1. The same reasoning could be applied to vertices H and I; allow H to be 2 and I to be 3. This also forces J to be 1, but that cannot be since J is adjacent to A which is one. There is a need for a fourth color.

A famous problem in graph theory questions whether one can color the different countries on a map so that two countries with a common border are assigned different colors. Mathematicians tried to solve this problem for over 100 years. Actually there was a proof given by Kempe in 1879, but it was shown to contain a fallacy. (Biggs, Lloyd and Wilson, 1976) Finally, a computer assisted proof was obtained by Appel and Haken in 1976. However, some mathematicians still consider it a conjecture.

Tucker gives an excellent proof of the 5 Color Theorem.
Theorem Every planar graph may be 5 colored.

We first need a lemma.

Lemma Any connected planar graph has one vertex of degree at most 5. (i.e. there must exist a vertex whose degree is 5 or less)

This is an example of a proof by contradiction.

What is the converse of this theorem? Assume it to be true.

Assume that every vertex has degree of at least 6. From a previous theorem (p.6D), we know that the sum of degrees of all vertices is equal to twice the number of edges. Therefore, $2E \geq 6V$

From the corollary on Euler's theorem, If G is a connected planar graph, then $E \leq 3V - 6$. (p.16D)

Observing these two statements

$$\begin{array}{rcl} E \geq 3V & E - 3V \geq 0 & \\ E \leq 3V - 6 & 3V - 6 - E \geq 0 & \\ & -6 \geq 0 & \end{array}$$

A contradiction exists, so what we assumed to be true is false.

The Lemma is true. Any connected planar graph has one vertex of degree at most 5. (Tucker, 1984)

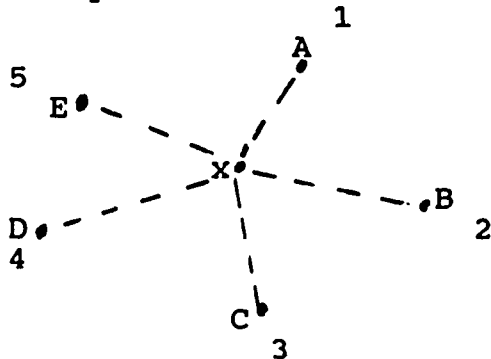
Another fascinating proof is the proof for the Five Color Theorem.

The proof is by induction.

1. Prove that a one vertex graph can be 5 colored.
It can be trivially one colored.
2. Assume that all connected planar graphs with (n-1) vertices can be 5 colored.
3. Prove that a connected planar graph G with N vertices can be 5 colored.

By the lemma above, since we have a connected planar graph there exists one vertex of G with degree at most 5. If we delete this vertex x from the graph, we know this graph with $(n-1)$ vertices can be 5 colored by the assumption.

Graph 51



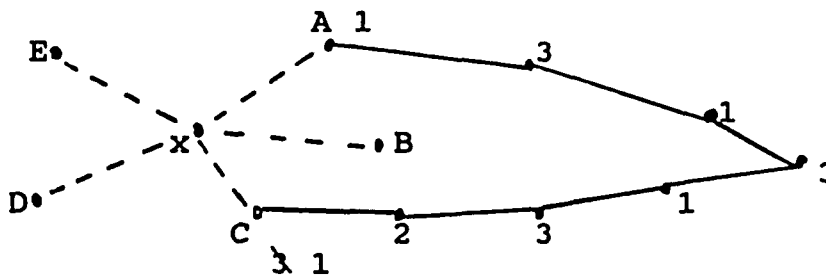
The vertices and their coloring

Attempt to place x back in the graph

If x has degree ≤ 4 , then we can simply assign to x a color different from the color of its neighbors. So the only real problem occurs when like the above picture, all 5 colors around x are used, and there is no free color available for x .

Two cases

1) Suppose there is no path using the color sequence 1-3 from A to C . We could make A and C the same color, 1, and free up color 3 for the vertex " x ".



2) If there is a 1-3 path from A to C , then it is impossible to free up one of those colors.

But look at B , in the interior of the 1-3 path. Since our graph is planar, there would be no 2-4 path between B and D . Therefore B and D can both be colored the same color 4, and appropriately change any of B 's adjacent vertices, and the color 2 is freed up for " x ".

Hence, we have colored a planar graph in 5 colors. The method has been used to try to prove the Four Color Theorem but to no success.

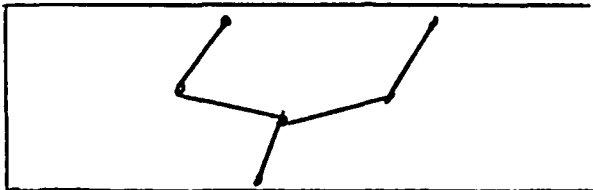
[Students may have difficulty with using the induction to prove the theorem. Why is it necessary to look at the vertex of degree 5? The inductive proof does not do that. If a graph has 22 vertices, by induction, after we remove a vertex of degree 5, we assume the graph of 21 vertices can be 5 colored. Then, what happens when we put the 22nd vertex back in the graph? The proof follows from there]

NETWORK TYPE PROBLEMS

Dr. Henry Pollak, former head of the mathematics department at Bell Labs, in his class on Mathematical Modelling gave a lecture on Graph Theory.

In 1956, accountants from ATT asked the mathematicians at Bell Labs to develop a formula for the following problem.

Graph 52



Given N points on a graph, develop a formula for the length of the line segments joining the points. Does a formula exist $L = c \cdot N$, i.e., is the length

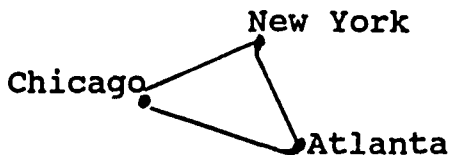
of the line segments directly proportional to the number of points in the graph. The mathematicians questioned the accountants about their need for such a formula. ATT was designing a method for charging for phone service such as a company desiring a special WATTS connection. If you distribute points randomly the result was $L \leq c\sqrt{N}$. There was no easy answer to this question.

ATT had to determine what is a fair way of charging for private line service. A list of such ways would have to include (1) a charge that was independent of actual calls made i.e any bulk service like an 800 number where you don't pay per call (2) a charge that was independent of actual lines used i.e. not where the physical telephone lines are placed (3) For two points, the distance between them is considered (4) As the size of a system increases, the charge should grow reasonably (5) The charge should be unique and easy to compute.

Is the following situation fair?

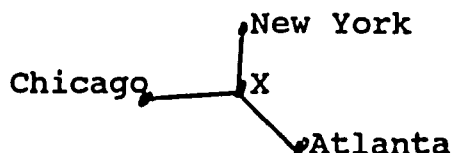
In 1960, Delta Airlines had branches in Chicago, New York and Atlanta.

Graph 53



A bright worker at Delta figured out if Delta established a branch at a strategic location, call it X, the sum of all the distances from that point X to all the cities is shorter than the original distance connecting all 3 cities.

Graph 54



This is a reference to Steiner points which was developed in 1770. (E.Gilbert and H.O.Pollak, 1968)

So an additional fair way of charging must be added.
(6) The charge must exclude a means of adding points to shorten length.

Another situation. When a phone call is to be made from NYC to Chicago at 10 A.M. , very often the lines are busy. In non-hierarchical routing, the call is switched to Los Angeles and then to Chicago. Why Los Angeles? It is only 7 A.M. and therefore phone lines are not busy. There are 10 nationwide switching centers, one of which is at White Plains New York. So for a long distance phone call, a direct route is preferred, but if it is not possible, two or three routes are pieced together. If many of these pieces are used, the telephone system may become overtapped and some of the calls are waiting for a 2nd or 3rd piece to come through.

What is the busiest day of the year for the phone company? Why, Mothers' Day, of course. What must Ma Bell do? All double and triple routes must be cut off, so there are no alternate routes for waiting purposes.

History of the Shortest Network Problem


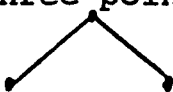
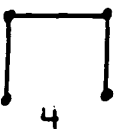
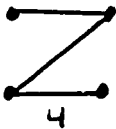

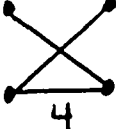
What should ATT charge for telephone service connection that gives a fair charging policy? At the old headquarters of ATT at 195 Broadway, Manhattan, a large map of the United States was placed on the floor and pins and strings were actually used to find the

"shortest network." A network is a graph with values (distance or cost) given to each edge. Long distance is charged by the geometric shortest physical distance -map service.

In 1956, the only property of shortest networks that was known was that the shortest network never includes a closed loop. Univac was asked to solve the problem of determining the shortest network for ATT.

DAY 8

How many different networks must be tested in order to find the shortest one? It may be necessary to put distances on each edge and show the students the total distance for each network, so they know what it means to find the shortest network.

		Points	# of networks	
1) Two points		2	1	2^{2-2}
2) Three points		3	3	3^{3-2}
3) Four points	 4	4	16	4^{4-2}
	 4	5	125	5^{5-2}
	 4			
	 4			

Using induction on the chart, for n points, there are n^{n-2} networks. This was first obtained by A. Cayley. Cayley wrote a paper, "On the theory of analytical forms called trees." (Biggs, Lloyd, Wilson, 1976) A tree is a connected acyclic graph, since trees do not have closed loops. The shortest network problem is renamed the minimum cost spanning tree problem.

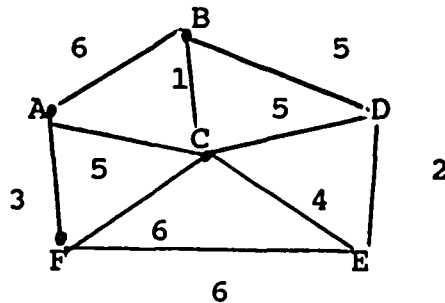
Is it necessary to test all of networks to see which is the shortest? (We hope not)

An algorithm must be developed so that all the spanning trees would not have to be found.

Given the opportunity, the students may devise ways of getting the shortest network. It was exciting to observe their methods.

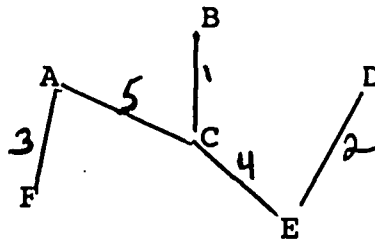
In 1956, J.B. Kruskal developed the following method.

For this graph (Aho, Hopcroft, Ullman, 1983, p.234) with the assigned weights, Graph 55



continually choose the smallest edge as long as no loops are formed.

Graph 56



BC - 1
DE - 2
AF - 3
CE - 4
AC - 5

cannot choose BD since it would form a loop

Now, every vertex is visited and the Minimum Cost Spanning Tree is formed.

The steps are simple: Examine edges by increasing cost
Maintain a set of components, that do not form a closed loop (THE COMPONENTS DO NOT HAVE TO BE CONNECTED)
Continue until every vertex is in the tree.

At that time, the longest network had 500 points. No, it was not a government agency; the service that checks your credit standing had over 500 different locations for its offices.

In 1958, Prim developed his greedy algorithm for determining the Minimum Cost Spanning Tree.

Starting with the smallest edge, continually add edges of minimum cost that are already attached to a branch on the tree, as long as it doesn't form a loop.

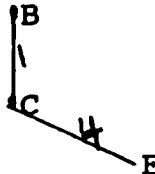
- 1) Choose BC



Add on to your possible list then, any touching edges

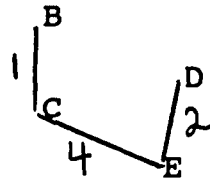
BD	CD	AC	AB	CE	CF
5	5	5	6	4	6

- 2) Choose CE



Add on to the possible list DE and FE
2 6

- 3) Choose DE



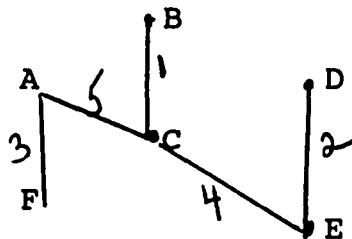
Nothing is added to the possible list.

- 4) Choose BD. No, it forms a loop. Actually B and D are already in the tree, so there is no need to join B and D.

- 5) Choose CD. No, for the same reason

- 6) Choose AC. That is fine.

Add AF on the list
3



- 7) Choose AF

Prim's greedy algorithm builds up a connected spanning tree.

What Abstract Data Type would be helpful in coding the possible list of edges? A queue? A stack? No, the priority queue is needed which could be implemented in several ways.

Some questions that would be helpful in developing the algorithm. You may want to develop a pseudo-code first.

1. How should we deal with the weight on each edge?
Put an extra field in the node.

2. How do you know which vertices are adjacent to the one just removed?

The adjacency list gives us that information.

3. How do you mark a vertex as "in the tree" or "on the queue"?

Use an extra field in the header node or use a parallel array to mark the vertex.

4. Does it matter if marked vertices are put on the waiting list? It is not done in the code, but it wouldn't matter if they were.

5. In our original diagrams, edges were put on the waiting list. This will not be the case in the code to be traced. If we just put a vertex on the waiting list, how will we know "its parent"? Realize, a "parent" may change while on the waiting list. A parallel array for parents will also be kept.

An explanation of the data structures for the following program.

```
Type markedtype = array['A'..'Z'] of integer;
   listtype= ptr;
   ptr= ^ node;
   node = record
       vertex:char;
       weight:integer;
       link:ptr;
   end;
   parenttype= array['A'..'Z'] of char;
   statustype = (intree, waiting, untouched);
   statusarray = array['A'..'Z'] of statustype;
```

Markedtype is similar to the array in breadth first traversal which shows the order of the visited vertices.

listtype is a linked list. THE CODE KEEPS THE WAITING LIST AS A PRIORITY QUEUE.

Parenttype array keeps track of an edge's parent - from where is it coming.

Statusarray is keeping track of a vertex being intree, waiting (on the priority queue) or untouched.

Baase (1988) suggests using parallel arrays rather than an array of records to avoid cumbersome code. I agree.

The code we will trace starts from A. Does it matter when finding the MCST? No, you may not get the same tree, but you will get the same minimum length.

```

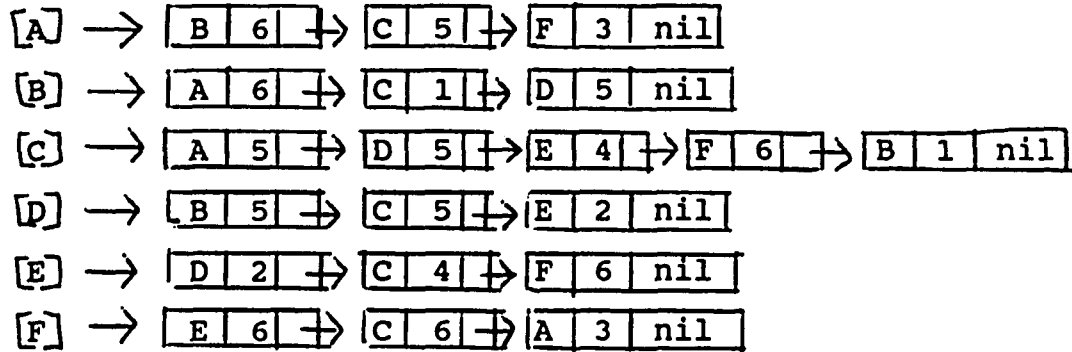
1 Procedure Minimumspanningtree(graph:graphtype);
2 var status:statusarray;
3   x,y:char;
4   edges:integer;
5   temp:ptr;
6   waitinglist:listtype;
7   parent:parenttype;
8   waitingweight:markedtype;
9   stuck:boolean;
10 begin
11   if trace then writeln('enter mst');
12   x:='A'; status['A']:=intree;
13   edges:=0;
14   clear(waitinglist);
15   for y:= 'B' to 'H' do
16     status[y]:= untouched;
17   stuck:=false;
18   while (edges < 10) and (not stuck) do
19     begin
20       temp:= graph[x];
21       while temp <> nil do
22         begin
23           y:= temp^.vertex;
24           if (status[y] = waiting) and
25             (temp^.weight < waitingweight[y])
26             then begin
27               parent[y]:= x;
28               waitingweight[y]:= temp^.weight;
29               updatewaitinglist(waitinglist,y,
30                 temp^.weight)
31             end;
32           if status[y] = untouched
33             then begin
34               status[y]:= waiting;
35               addon(waitinglist, y,
36                 temp^.weight);
37               print(waitinglist);
38               parent[y]:= x;
39               waitingweight[y]:=temp^.weight;
40             end;
41           temp:= temp^.link;
42         end; {while temp<> nil}
43       if empty (waitinglist)
44         then stuck:=true;
45       if not empty (waitinglist)
46         then begin
47           remove(waitinglist, x);
48           status[x]:=intree;
49           edges:=edges + 1
50         end;
51     end; {while edges < 10}
52   if trace then writeln('exit mst');
53 end;

```

The trace table for Prim's MCST from A

The ADJACENCY LIST

builds up
connected spanning
tree



	STATUS	PARENT	WAITINGWEIGHT
[A]	intree		
[B]	untouched waiting intree	'A' 'C'	6 1
[C]	untouched waiting intree	'A'	5
[D]	untouched waiting intree	'C' 'E'	5 2
[E]	untouched waiting intree	'F' 'C'	6 4
[F]	untouched waiting intree	'A'	3

Trace table for Prim's MCST

WAITING LIST	TEMP	STUCK	EDGES	X	Y
WL nil	^B	false	0	'A'	'B'
WL → B 6 \	^C				'C'
WL → C 5 → B 6 \	^F				'F'
WL → F 3 → C 5 → B 6 \	nil			'F'	
	^E				'E'
WL → C 5 → E 6 → B 6 \	^C				'C'
	^A				'A'
	nil			'C'	
WL → E 6 → B 6 \	^A				'A'
	^D				'D'
WL → D 5 → E 6 → B 6 \	^E				'E'
WL → E 4 → D 5 → B 6 \	^F				'F'
	^B				'B'
WL → B 1 → E 4 → D 5 \	nil			'B'	
WL → E 4 → D 5 \	^A				'A'
	^C				'C'
	^D				'D'
	nil			'E'	
WL → D 5 \	^D				'D'
WL → D 2 \	^C				'C'
	^F				'F'
	nil			'D'	
WL → nil	^B				'B'
	^C				'C'
	^E				'E'

true

Time complexity for MCST

Again we are traversing each vertex and all of its edges. Is it another $O(e)$ algorithm as depth first and breadth first searches? No, keeping track of the waiting list adds to the time complexity. In our implementation of the waiting list, the priority queue, we used an ordered single linked list. Removing an item from an ordered single linked list with highest priority is $O(1)$, but adding an element to any ordered linked list is $O(n)$. The operations for the waiting list are inside the while loops, so the time complexity is $O(e \cdot n)$.

Improving the efficiency depends on the implementation of the priority queue. If the priority queue were kept as a heap, removing an item is again $O(1)$ but inserting an element into a heap and retaining the heap properties is $O(\log n)$. Therefore the time complexity for MCST improves to $O(e \log n)$.

The code for Kruskal's algorithm was not mentioned. It involves knowing the union find algorithm which was not covered. Baase gives a pseudo-code for Kruskal's MCST algorithm in her text. (Baase, 1988) It is noteworthy to mention that Kruskal's algorithm runs faster for sparse graphs while Prim's algorithm is better for dense graphs.

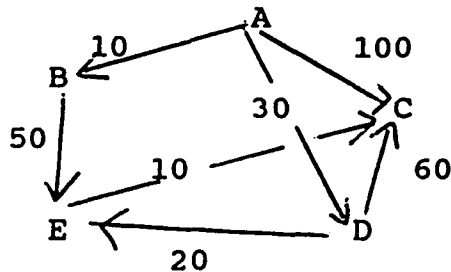
Kruskal's algorithm and Prim's algorithm for MCST are both greedy algorithms since at each step it bites off the most desirable piece.

Single Source Shortest Path

DAY 9

Graph 57 (from page 1D of sourcebook)

(Aho, Hopcroft, Ullman, 1983, p.205)



Notice this
is a directed
graph

There exists two different problems.

What is the shortest path from one vertex to another?

What is the shortest path from one vertex to every other vertex?

What are the costs from A to C?

An exhaustive search method

A, C	100
A, D, C	30 + 60 = 90
A, B, E, C	10 + 50 + 10 = 70
A, D, E, C	30 + 20 + 10 = 60

What is the difficulty with this method? It is time consuming to list all the possibilities.

Dijkstra's method for Single Source Shortest Path (SSSP) is very similar to Prim's MCST algorithm.

Begin by listing the distances from A to every other vertex.

S is the set of vertices already in the tree.

	Dist[B]	Dist[C]	Dist[D]	Dist[E]
S = {A}	10	100	30	∞

Choose the vertex with the minimum distance and add it to set

S = {A, B} Now, what is the minimum distance to every vertex using the vertex just added in to the set. This only helps us in getting to E.

	10	100	30	60
				A-B-E

Choose the vertex with the minimum distance

S = {A, B, D} Now use D to get to every other vertex.

	10	90	30	50
		A-D-C		A-D-E

Choose the vertex with minimum distance

S = {A, B, D, E}	10	60	30	50
		A-D-E-C		

The method terminates when all the vertices are in the set (or on the tree)

Call this method, the Chart method of Dijkstra for SSSP.

Dijkstra's Algorithm for SSSP.

S: set of vertices whose minimum cost are known
 Dist[x] minimum cost to get to vertex X passing only
 through vertices in S

originally

Dist[i] = cost['A',i]
 if there is no arc, cost = ∞

Algorithm

```

1 For V:= 1 to (n-1) do
2   find min Dist[w] where w is in V - S
3   Add w to S
4   for vertices in V-S update D
  
```

The code given for SSSP is very similar to Prim's MCST. Compare the algorithm to the code.

In addition to the data structures for minimum cost spanning tree, what new information is to be saved and how should we save it?

The distance of paths between 2 points will be updated in SSSP while the waitingweight in MCST was not. When is the distance updated? The distance value is updated for a vertex y when the distance from the point of origin to the vertex, x, whose adjacency list is presently being traversed plus the distance in the present node is less than the known distance from the point of origin to that particular vertex y.

No algorithm exists to determine minimum airflight cost. You travel agent doesn't necessarily give you the lowest fare, i.e. if you are willing to make several stops.

Time complexity

Updating the distances in the Single Source Shortest Path does not decrease the efficiency that was discussed with the Minimum Cost Spanning Tree. The time complexity is again $O(n^2)$ unless the priority queue is kept as a heap which will improve the time complexity to $O(e \log n)$ and the updated distances are just added to the heap, not searching and replacing the former larger distances.

```

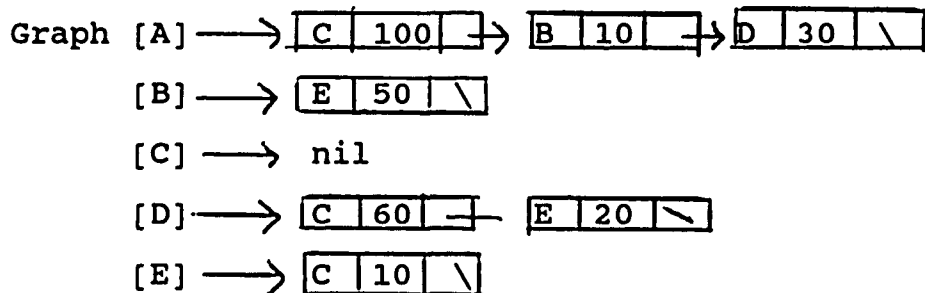
1 Procedure ShortestPath(graph:graphtype; v,w:char);
2 var status:statusarray;
3   parent:parenttype;
4   waitinglist:listtype; 5   distance:markedtype;
6   temp:ptr;              7   stuck:boolean;
8   x,y:char;
9 begin
10  if trace then writeln('entering shortest path');
11  distance[v]:=0;
12  parent[v]:=' ';
13  clear(waitinglist);
14  For y:='A' to 'H' do
15    status[y]:= untouched;
16  status[v]:=intree;
17  x:=v;
18  stuck:=false;
19  while (x<> w) and (not stuck) do
20    .begin
21      temp:=graph[x];
22      while temp <> nil do
23        begin
24          y:=temp^.vertex;
25          if (status[y] = waiting) and (distance[x] +
26            temp^.weight < distance[y])
27            then begin
28              parent[y]:= x;
29              distance[y]:= distance[x]+temp^.weight;
30              updatewaitinglist(waitinglist,y,temp^.weight);
31            end;
32            if status[y] = untouched
33              then begin
34                status[y]:=waiting;
35                addon(waitinglist, y, temp^.weight);
36                parent[y]:= x;
37                distance[y]:= distance[x] +
38                  temp^.weight;
39              end;
40              temp:= temp^.link;
41            end;
42            if empty(waitinglist)
43              then stuck:=true
44              else begin
45                remove(waitinglist,x);
46                status[x]:= intree
47              end;
48            end; ( while x<> w and not stuck do)
49            writeln('the shortest path in reverse order is');
50            while x<> ' ' do
51              begin
52                writeln(x);
53                x:= parent[x];
54              end;
55            end;

```

Single Source Shortest Path

Using this adjacency list, makes the code more interesting to trace.

Adjacency List



Aside: An interesting way to find the shortest path for an undirected graph is from A.K. Dewdney's The Turing Omnibus. (1989, p. 203)

Once again, we may solve this shortest-path problem by preparing a physical analog of the graph in question. Cut a long string into lengths which reflect the numbers assigned to edges, allowing a little extra for knotting the ends of the strings together. When the strings have thus been tied into a configuration identical to the graph, then take the "vertices" (knots) u and v in separate hands and pull them apart until the network of strings resists any further separation.

The shortest path (or all shortest paths, if there is more than one) will stand out clearly as a sequence of taut strings between the two hands. If knots have been labeled with the names of the vertices which they represent, then the shortest path can easily be identified in the original graph and the problem is solved.

Trace Table for Shortest Path from 'A' to 'C'

Dijkstra's algorithm

	STATUS	PARENT	DISTANCE
A	untouched intree	' '	0
B	untouched waiting intree	'A'	10
C	untouched waiting intree	'A' 'D'	100 90 60
D	untouched waiting intree	'A'	30
E	untouched waiting intree	'B' 'D'	10 + 50 = 60 30 + 20 = 50

WAITING LIST	TEMP	STUCK	V	W	X	Y
			'A'	'C'	'A'	
	^C					'C'
WL → [C 100 \]	^B					'B'
WL → [B 10] → [C 100 \]	^D					'D'
WL → [B 10] → [D 30] → [C 100 \]	nil					'B'
WL → [D 30] → [C 100 \]	^E					'E'
WL → [D 30] → [E 50] → [C 100 \]	nil					'D'
WL → [E 50] → [C 100 \]	^C					'C'
WL → [E 50] → [C 60 \]	^E					'E'
WL → [E 20] → [C 60 \]	nil					'E'
WL → [C 60 \]	^C					'C'
WL → [C 10 \]	nil					'C'
WL nil	nil					

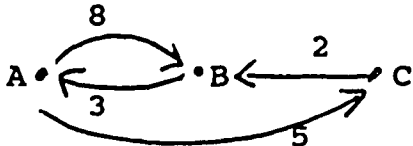
If you need to find the shortest paths from all n source nodes, you must apply Dijkstra's algorithm n times. The resulting algorithm is $O(n^3)$ since Dijkstra's algorithm for SSSP is $O(n^2)$. Again improvement can be made if the priority queue were kept as a heap.

Floyd developed an algorithm for finding the shortest paths from all n source nodes that is also $O(n^3)$ as can be easily seen by the nested for do loops.

Floyd's Algorithm for All Pairs Shortest Paths (APSP)

A two dimensional array is kept for the distance (cost) between 2 vertices. ∞ is used if there is no path.

From this small graph (Aho, Hopcroft, Ullman, 1983, p.207) Graph 57



$Dist_k[i,j]$ minimum distance for a path from i to j that does not pass through a vertex lettered higher than k .

$Dist_{\infty}, [i,j]$	A	B	C
A	0	8	5
B	3	0	∞
C	∞	2	0

$Dist_A [i,j]$ You are allowed to pass through vertex A.

	A	B	C
A	0	8	5
B	3	0	\leftarrow minimum(∞ , distance from B to C using A) $3 + 5$
C	∞	\leftarrow minimum(2, distance from C to B using A)	0

$Dist_A [i,j]$	A	B	C
A	0	8	5
B	3	0	8
C	∞	2	0

$Dist_B [i,j]$ allows the intermediate vertices to be A or B

	A	B	C
A	0	8	\leftarrow dist from A to C using B \leftarrow minimum(5, $8 + 8$) = 5
B	3	0	\leftarrow minimum(8, $0 + 8$)
C	\leftarrow minimum(∞ , distance from C to A using B) $3 + 2$	2	0

$$\text{Dist}_B [i,j]$$

	A	B	C
A	0	8	5
B	3	0	8
C	5	2	0

$$\text{Rule } \text{Dist}_K [i,j] = \text{minimum} \begin{cases} \text{Dist}_{K-1} [i,j] \\ \text{Dist}_{K-1} [i,k] + \text{Dist}_{K-1} [k,j] \end{cases}$$

Realize, since using letters, when $k='B'$, $k-1 = 'A'$
 Every entry in $\text{Dist}_K [i,j] \leq \text{Dist}_{K-1} [i,j]$

$$\text{Dist}_C [i,j]$$

	A	B	C
A	0	8	5
B	3	0	8
C	5	2	0

minimum(8, dist from A to B using C)
 $5 + 2$

All entries should be checked to see if there are any other minimum paths

$$\text{Dist}_C [i,j]$$

	A	B	C
A	0	7	5
B	3	0	8
C	5	2	0

Floyd's algorithm finds for us the shortest distance between every two vertices on the graph. Floyd's algorithm demonstrates if it is worth testing whether we should pass through an intermediate vertex or to move to a vertex directly. Express Air delivery has become a very lucrative business. Federal Express chose Memphis, Tennessee for its central delivery point because of Floyd's algorithm and the availability of airport space.

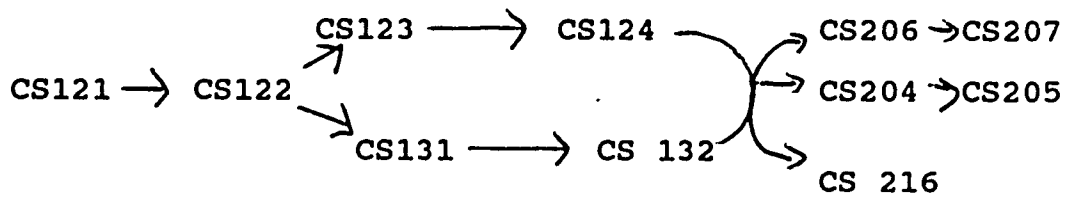
Floyd's Algorithm

```

For K:= 1 to N do
  For I:= 1 to N do
    For J:= 1 to N do
      if Dist[I,K] + Dist[K,J] < Dist[ I,J]
        then Dist[I,J] = Dist[I,K] + Dist[K,J]
  
```

Topological Sort

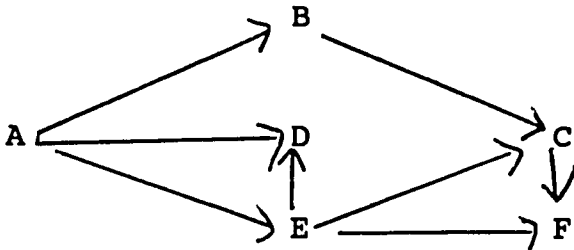
In our Computer Science Department, there are prerequisites for required and elective course.



There are a few different sequences to study courses.
 CS 121, CS 122, CS 123, CS 124, CS 131, CS 132, CS 204,
 CS 205 ...

or CS 121, CS 122, CS 131, CS 132, CS 123, CS 124, CS
 206,
 CS 204, CS 207 ...

A topological sort is a linear ordering that has the property that if i is a predecessor of j in the network than i precedes j in the linear ordering.



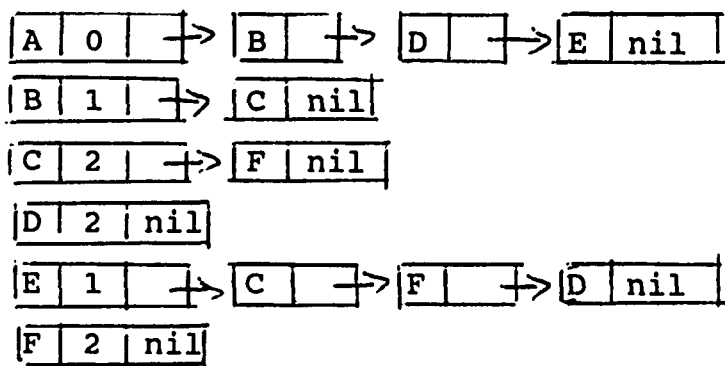
What are possible topological sorts for this graph?

Before each vertex is visited, all of its prerequisites must be traversed. In eliciting an Abstract Data Type to solve the problem, students may respond Stack or Queue. Putting all the elements on a stack as in depth first search will not work. But for above graph placing the elements on a queue in breadth first fashion will work. A counterexample is needed to show the problem with bfs. Construct an edge between A and C for the counterexample.

We must keep track of the indegree of a vertex, that is the number of edges coming into a vertex.

A counter field in the vertex node is used for the indegree of a vertex or in the code we use a parallel array with the incunts that is passed into procedure topological order, so that the original implementation of graph is not changed. Countarray = array['A'..'Z'] of integer.

The adjacency list for the graph.



```

1 Procedure topologicalorder(var graph:graphtype;
2   n:integer, count: countarray);
3 Var j,k:integer;
4   temp:ptr;
5   done:boolean;
6   stack:stacktype;
7   I:char;
8 begin
9   clearstack(stack);
10  For I:= 'A' to lastvertex do
11    if count[I] = 0
12      then push(stack, I);
13    J:=1;
14    done:=false;
15    while(J<= N) and (not done) do
16      begin
17        if empty(stack) do
18          then begin
19            writeln('loop formed');
20            done:=true;
21          end
22        else begin
23          pop(stack,I);
24          writeln(I);
25          temp:= graph[I];
26          while(temp<> nil) do
27            begin
28              k:= temp^.vertex;
29              count[k]:=count[k] - 1;
30              if count[k] = 0
31                then push(stack,k);
32              temp:= temp^.link;
33            end; {of while temp <> nil do}
34          end;
35          J:= J + 1;
36        end;
37 end;

```

The trace for topological sort. Notice the use of the stack.

bottom Stack	→ top	J	Done	N	I	temp	k
A →		1	false	6	'A'	^B	B
B →						^D	D
						^E	E
B, E →						nil	
B →		2			'E'	^C	C
						^F	F
						^D	D
B, D →						nil	
B →		3			'D'	nil	
		4			'B'	^C	C
C →						nil	
		5			'C'	^F	F
F →						nil	
		6			'F'	nil	

true

```

count[A]  0
count[B]  1  0
count[C]  2  1  0
count[D]  2  1  0
count[E]  1  0
count[F]  2  1  0

```

Classification and Efficiency of Algorithms

In the graph theory component of the algorithms course, we have studied many different type problems and some solutions.

Determine whether two graphs are isomorphic.

Is this graph planar?

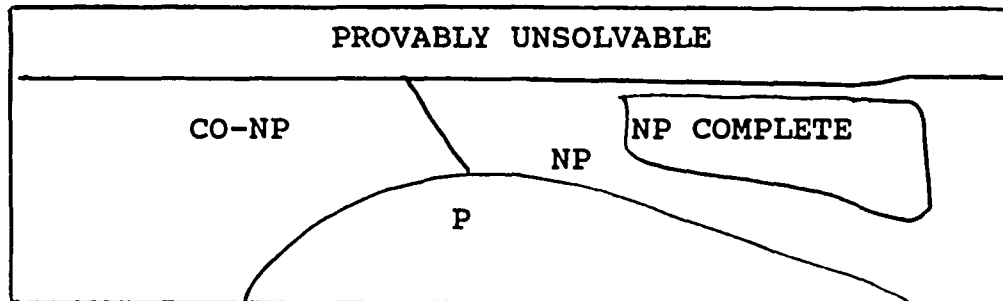
Does this graph have an Euler path or Hamiltonian path?

Can this graph be colored using only five colors? using only four colors?

Given a graph, what is the minimum cost spanning tree?

Find the shortest path between two points on a graph.

Mathematicians and computer scientists have set about classifying algorithms depending upon their efficiency.



Let's look at some of these classes.

There are some problems in mathematics for which efficient algorithms exist: square root, long division, Minimum Cost Spanning Tree and Single Source Shortest Path. These are placed into the class P for which polynomial time algorithms exist. There are other problems for which no algorithms exist and still another group of problems which can only be solved by inefficient and therefore largely unusable algorithms.

Garey and Johnson's introduction in Computers and Intractability gives an intuitive understanding of an NP complete type problem.

The boss calls you into his office. "He needs a good method to determine whether or not any given set of specifications for a new "bandersnatch" component can be met, and if so, for constructing a design that meets them."

After a considerable amount of work, you cannot find an algorithm better than searching through all possible designs. This is not the desired method.

How should you approach the boss?

1. I cannot find an efficient algorithm. I guess I am just too dumb.
2. I cannot find an efficient algorithm because no algorithm is possible.
3. I cannot find an efficient algorithm but neither can all of these famous people.

The boss would prefer the second option if there were a proof that no algorithm is possible. But such proofs are just as hard as finding the algorithms themselves. The third response should make the boss just as satisfied. (Garey and Johnson, 1979, pp.1-3)

Class NP consists of problems that can be solved by a non-deterministic machine in polynomial time. An explanation of non-deterministic machine is given on page 61D of the sourcebook.

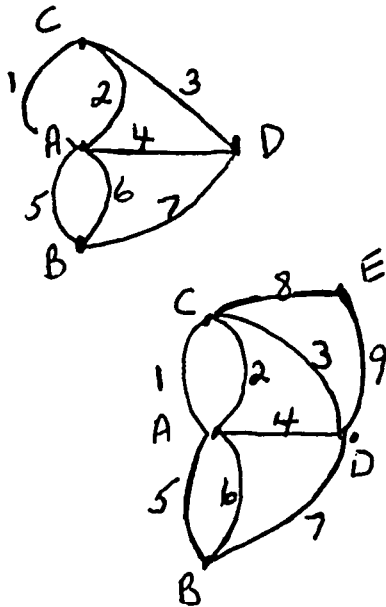
Why should we be concerned about efficient or inefficient algorithms? Hopcroft states in his Turing Award Lecture, "People argued that faster computers would remove the need for asymptotic efficiency. Just the opposite is true with faster computers, the size of attempted problems becomes larger." (Frenkel, 1987, p.200)

Lewis and Papadimitriou in "The Efficiency of Algorithms" return to Euler's Konigsberg bridge problem to explain efficiency. (Source Book, p.17D) Without knowing Euler's theorem, one could list all possible paths and observe if any satisfy the requirement of visiting all bridges. This exhaustive search approach is too time consuming when there are a fair number of vertices.

Another approach is to use the Depth First Search to see if the graph is connected and then determine if the degree of every vertex except two is even. This is a very efficient way of testing to see if there is an Euler path.

How much better is the second approach?

With the original graph of 4 points and 7 edges both Euler's techniques and exhaustive search are fast enough to be considered practical. Yet every time we add one more point and a few edges, the size of the list of possible paths for the exhaustive search doubles.



The exhaustive search from A
 125647
 125643
 126475

There are at least 43 different paths just starting from A. It can be tediously shown that by adding a point E and two edges, much more work is required using this method.
 124983765
 124983756
 124389765
 124389756

Exhaustive search

Using the rules for geometric sequence $l = ar^{n-1}$, where l = last term, a = first term, r is the ratio of the second term divided by the first, and n is the number of terms you wish in your sequence, the number of test cases is 25 for a graph of 10 vertices for some unknown algorithm. One can see how quickly the number of test cases increases, as the number of vertices increases by 1.

number of vertices	number of test cases
10	25
11	50
12	100
13	200
⋮	⋮
20	256,000

Exhaustive search not only is a slower method... in general it is too slow to be of any value." (Lewis and Papadimitriou, 1978, p.99) If n is the number of vertices then the growth of this kind, can be described by a mathematical function such as 2^n . There are other functions n^n and $n!$ which have similar or higher rates of growth. On p.39D, there is a reference to Cayley's result that the number of networks given n points is

$$n^{n-2}$$

Many texts give tables showing values of n and the comparison of time for different type algorithms. (Horowitz, Sahni, 1989, p.124; Baase, 1988, p.32; Sedgewick, 1988, p.74)

Polynomial functions characterize the class of problems that are guaranteed to run in a reasonable amount of time. In a polynomial function, n , the size of the list would never appear as an exponent. Euler's method for finding Euler paths increases as a linear function of the size of the graph. Dijkstra's method for SSSP is displayed as a quadratic function. For small values of n , the polynomial function may have a greater value than an exponential function, yet there is always a value of n beyond which the exponential function is greater.

This discussion of efficiency is independent of the machine. "For sufficiently large problems a polynomial time algorithm executed on even the slowest machine will find an answer sooner than an exponential time algorithm on the fastest computer." (Lewis and Papadimitriou, 1978, p.101)

Notice the provably unsolvable problems. These are not the inefficient problems we have been discussing. A classic example of a provably unsolvable problem is the Turing Machine Halting Problem.

"Is it possible to construct a program or algorithm to decide if an arbitrary program halts on an arbitrary input?"

While this question will be discussed in a later course, Theoretical Computer Science, a brief explanation will be given. This problem was presented at an AP Pascal Conference at Barnard in December 1986, and is also discussed in "A Programming Approach to Computability." [Kfoury, Moil, Arbib, 1982]

This proof is another example of proof by contradiction.

Assume there is a function called $\text{halt}(x)$ that upon reading a program determines if it halts.

$\text{halt}(x) = 1$ if the program halts on input x
 $\text{halt}(x) = 0$ otherwise

Given another program called Confuse.

```
Program Confuse;
begin
  y:=1;
  read(N);
  if halt(N) = 1
    then while y=1 do
      (nothing)
    end;
```

If halt is a legal program, then so is Confuse. Hence, the program Confuse can be operated on by halt . Confuse will only terminate when $\text{halt}(N) = 0$, but that is a contradiction. Confuse does not halt when $\text{halt}(N) = 1$,

but that is when it was supposed to terminate. The contradiction tells us that confuse and halt cannot exist together. It is unsolvable whether an arbitrary computer program halts on its own index.

"We have only the deepest sympathy for those readers who have not encountered this type of simple yet mind-boggling argument before. It resembles the argument in "Russell's paradox", which forces us to discard arguments concerning the class of all classes, a notion whose absurdity is not otherwise particularly evident." (Minsky, 1967, p.149)

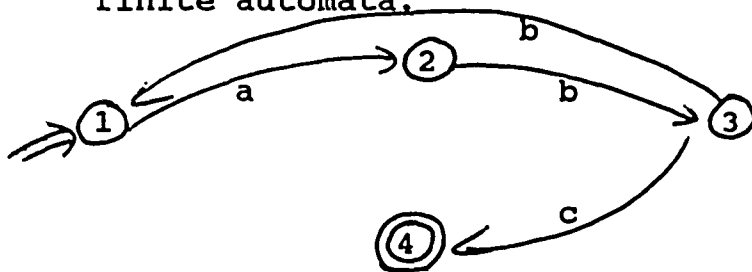
Ian Stewart in The Problems of Mathematics (1987, p.218) shares with us an idea that " has a similar logical structure to a card, on one side of which is written:

The statement on the other side of the card is true.
and on the other:

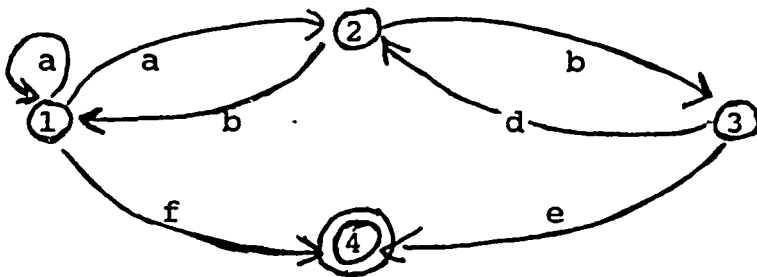
The statement on the other side of the card is false."

Finding an Euler path is considered a polynomial time problem, using Euler's method. We have already discussed there is no efficient solution to the finding of a Hamilton path or in answering the Travelling Salesman problem. (p.28D) But mathematicians have not been able to prove there is no efficient method. The class NP or non-deterministic polynomials are a set of problems that can be solved by a non-deterministic algorithm in polynomial time.

The explanation of non-determinism is understood by brief discussion of deterministic and non-deterministic finite automata.



What are suitable words in this language if the initial state is 1 and the ending state is 4?
abc very definite! The next state is determined
abbabc at the reading of a certain character.



In this non-deterministic finite automaton, the words consist of

abe
aabe
aabbabe
abdbf
... etc.

At certain states in this multigraph, you have 2 choices for the next state depending on a character read in. That is the non-determinism.

In a deterministic algorithm, each time we run an algorithm with the same input, we get the same output. This does not happen with non-deterministic algorithms. Class NP consists of problems that can be solved by a non deterministic machine in polynomial time.

Many problems in the class NP can now be solved by exponential time algorithms. Perhaps in the future, efficient algorithms may be discovered in which case NP will be the same as P. At this time, we think that is highly unlikely.

How do we distinguish between P and NP?

For a problem to be in the class P, the efficient algorithm must be given.

"Problems in the class NP ask a yes-or-no question that often can be answered only through a time-consuming, inefficient procedure, but the answer is known to be yes." (Lewis and Papadimitriou, 1978, p.103) A demonstration of the solution can be shown in polynomial time. For example, given a certain graph a Hamilton path can be shown even though there were no efficient algorithm to find it. The demonstration of the Hamilton path is the "Yes" answer to the question, "Is there a Hamilton path?" It may have been very difficult to find a Hamilton path for there is no known efficient algorithm.

The method we used to find Hamilton paths on page 29D is called a heuristic since it does not work in all cases.

Another problem in the class NP asks whether a number is composite. Can it be written as the product of two other numbers? Even with the computer it takes a long time to find factors of a very large number, but once they are found the two factors multiplied together

In 1640 Pierre de Fermat proposed that 4,294,967,297, which is equal to $2^{32} + 1$, is a prime number, and he was not proved wrong until Euler discovered the factors of the number in 1732. (Lewis and Papadimitriou, 1978, p. 103) The two factors are 6700417 and 641.

Whenever the answer is yes for asking whether there is a solution to an NP problem, there must be a short and convincing argument for proving it.

The complements of NP problems may not be in the class NP. The complement of the Hamilton path problem asks to show that given a graph there is no path passing through each vertex. The only solution known is to show all the paths (Exhaustive search) and this does not qualify for a short argument for the proof. So the complement of the Hamilton path problem may not be in the class NP. Mathematicians have put this sort of problem into a class of its own. co-NP. Co-NP consists of the complements of problems that are NP and are not NP themselves.

What about the complement of the composite number problem? Is this number prime? Vaughan Pratt of MIT has demonstrated short proofs for determining that a number is prime, yet no one has discovered an efficient algorithm. Testing for a prime number is an example of an intractable problem. It can be proved that there is no fast algorithm for it. A problem is tractable if there is a fast algorithm that will solve all instances of the problem. The prime number problem is an NP problem.

To summarize there are two definitions for NP problems. First, NP consists of problems that can be solved by a non-deterministic machine in polynomial time. Second, NP problems are decision type problems, which cannot be solved by a deterministic algorithm, yet for which a solution for a given input can be checked quickly.

Mathematicians have proven that if one efficient solution is discovered for some problems in NP then many problems in NP will also have a solution. How remarkable! They are reducible to each other. We call these problems NP Complete. This is a profound discovery, that is, if one solution is found, more problems will get solutions. In 1971, Stephen Cook was able to prove this is so by using propositional calculus. (More on this in the Theoretical Computer Science Course). Since a proof exists that there is no fast algorithm to determine if a number is prime, the prime number problem is NP but it is not NP complete.

Is $P = NP$ or is P a proper subset of NP ? "In other words," Ian Stewart (1987, p.210) states, "if you can check a solution in polynomial time, can you find it in polynomial time?" Baase (1988) states that it is believed that NP is a much larger set than P . But no one has been able to prove that any NP problem is not in P . Garey and Johnson (1979) have stated that Graph Isomorphism (determining whether two graphs are isomorphic to each other) is an open problem and have not classified it in any of the groups just studied.

Why is it good to know whether problems are in the class P , NP , or NP complete? Since NP complete algorithms probably have no efficient algorithms, it is not worth looking for one. Approximate heuristic solutions that are close to optimum solutions should be used.

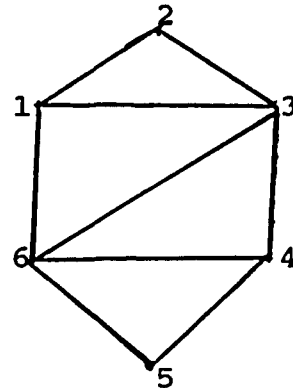
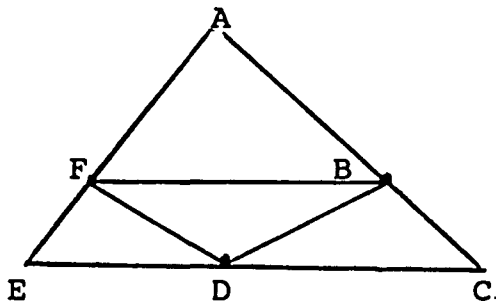
In teaching the algorithms course, we require students to research an algorithm we have not discussed in class, write a short paper explaining the algorithm and then make a presentation to the entire class (after discussing it with the professor). Some of the interesting graph theory problems that have been chosen were the Stable Marriage Problem, Knapsack or Bin Packing problems (Sedgewick, 1986) and Instant Insanity (Tucker, 1984 or Grimaldi, 1985).

APPENDIX C
HOMEWORK SHEETS

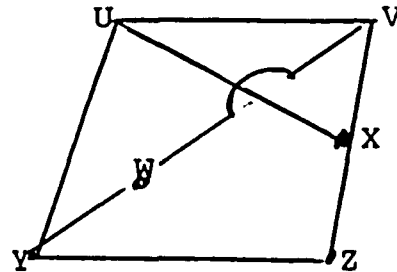
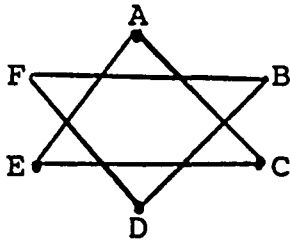
Sheet #1 Definitions and Isomorphism

1. Are the following graphs isomorphic?
If yes, give the matching of the vertices.
If not, explain why not.

a)



- b) (from Grimaldi, 1985, p.437)
State your method.

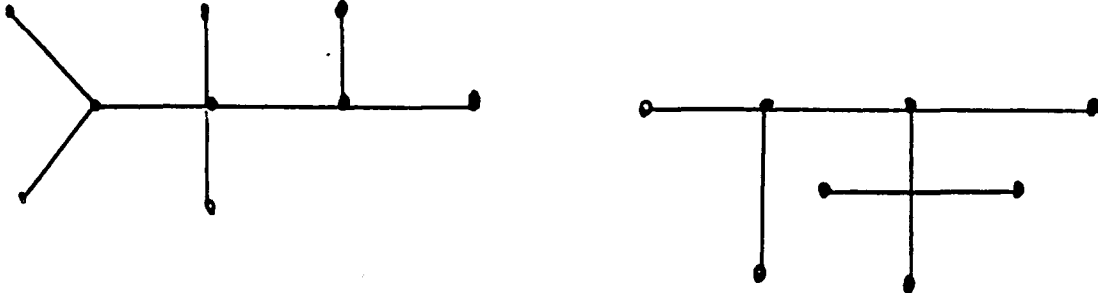


2. Coach Courtney is having a tournament for the six in-house basketball league. Each team must play two other teams. Draw a model (graph) for a possible schedule for this. There is no time limit to the games. Are all the answers isomorphic to each other?

Sheet #2

More on isomorphism, adjacency matrices and adjacency lists

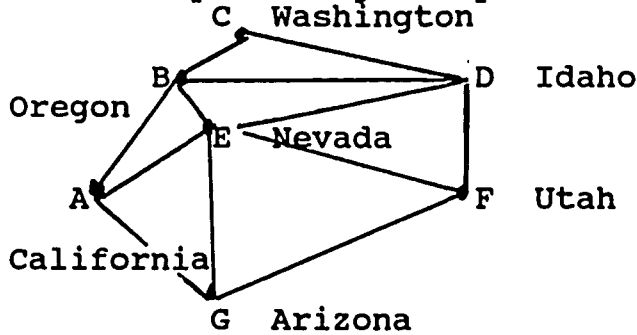
1. Are the following graphs isomorphic? If yes, give the matching? If not, why not? (from Molluzzo, 1986, p.412)



2. Given this adjacency matrix, draw its corresponding graph. Are all the answers isomorphic to each other?

	A	B	C	D	E
A	0	1	0	1	0
B	1	0	1	0	1
C	0	1	0	1	0
D	1	0	1	0	1
E	0	1	0	1	0

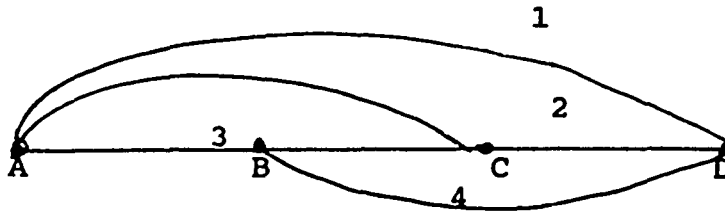
3. Set up the adjacency list for the following graph.



4. (from Tucker, 1984, p.28) There used to be 26 football teams in the NFL with 13 teams in each of 2 conferences. An NFL guideline said that each team's 14 game schedule should include exactly 11 games against teams in its own conference and 3 games against teams in the other conference. By considering the right part of a graph model of this scheduling problem, show that this guideline could or could not be satisfied!

Sheet #2 continued

5. A graph is partitioned into regions(faces). A region is an area surrounded by edges. No matter how many different isomorphic ways a planar graph is drawn the number of regions remains the same.



This graph has
4 regions.

Look at some of our graphs and fill in the chart.

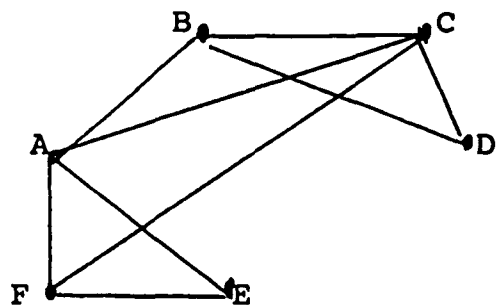
Regions	Edges	Vertices	
_____	_____	_____	Graph 21,23
_____	_____	_____	Graph 19
_____	_____	_____	Graph 3
_____	_____	_____	Graph 4

Can you find a relationship among the regions, edges and vertices?

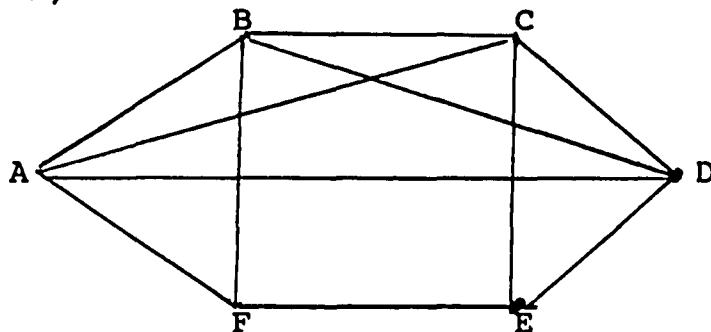
Sheet #3 Planar graphs

1. Redraw the graphs with as few edge crossings as possible. If the graph turns out to be planar, verify Euler's formula $R = E - V + 2$. If it isn't planar, can you prove it isn't?

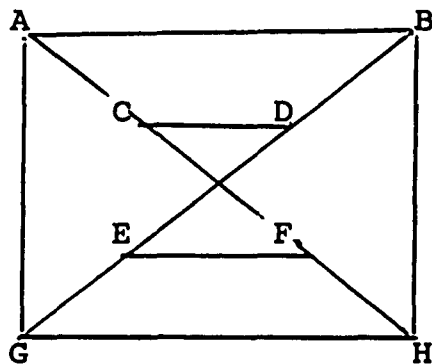
a)



b)

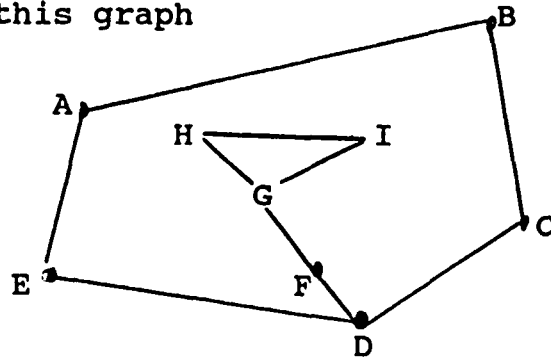


c)



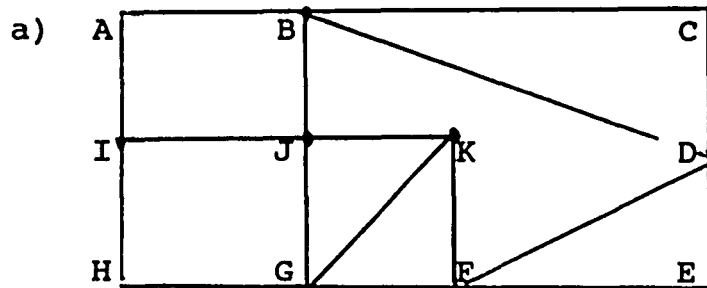
Sheet #4 Euler paths and circuits.

1. For this graph

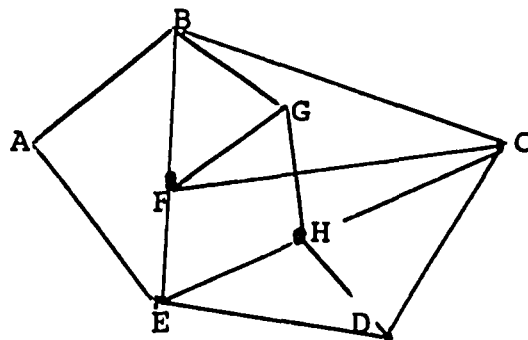


How many regions are there? _____
 What is the degree of each region? _____
 Verify that the sum of the degrees of all regions is twice the number of edges.

2. Find an Euler circuit or path for:



b)



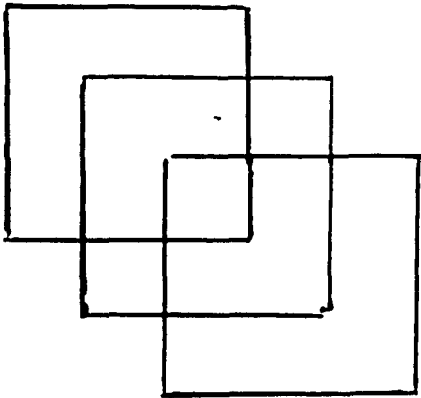
c) When would you be looking for an Euler circuit?

d) When would you be looking for an Euler path?

Sheet #4 continued

3. (from Martin Gardner's Sixth Book of Mathematical Games from Scientific American, 1971, p.96)

Lewis Carroll first proposed the problem.
Find an Euler path without intersecting lines. You must traverse each edge exactly once.



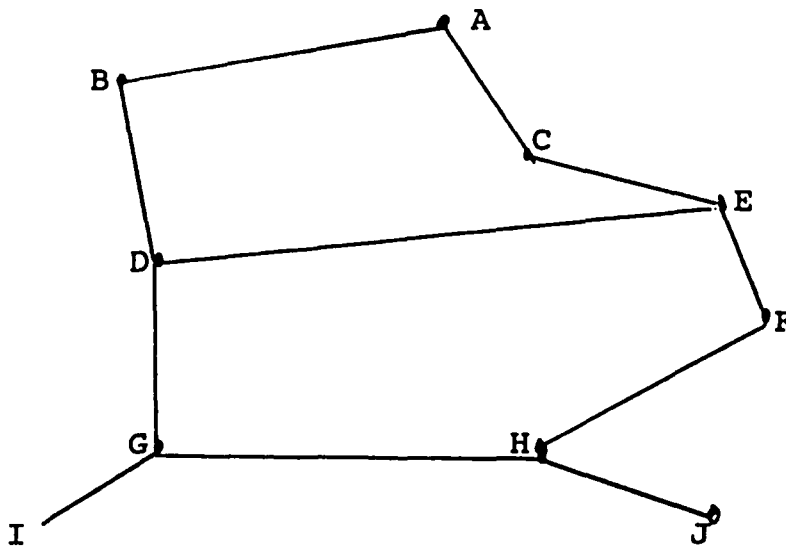
4. (from Tucker, 1984, p. 92) Suppose we are given 3 pitchers of size 10 quarts, 7 quarts, and 4 quarts. Initially the 10 quart pitcher is full and the other 2 are empty. We can pour from 1 pitcher into another pouring until the receiving pitcher is full or the pouring pitcher is empty. Is there a way to pour among pitchers to obtain exactly 2 quarts in the 7 or 4 quart pitcher? If so, find a minimum sequence of pourings to get 2 quarts.

Sheet # 5 Breadth first traversals

1. Consider the tic-tac-toe game after a first move by X and then a move by O as shown. Build a tree for the successive plays of the game and show how X can always win. Refer to the boxes as numbered

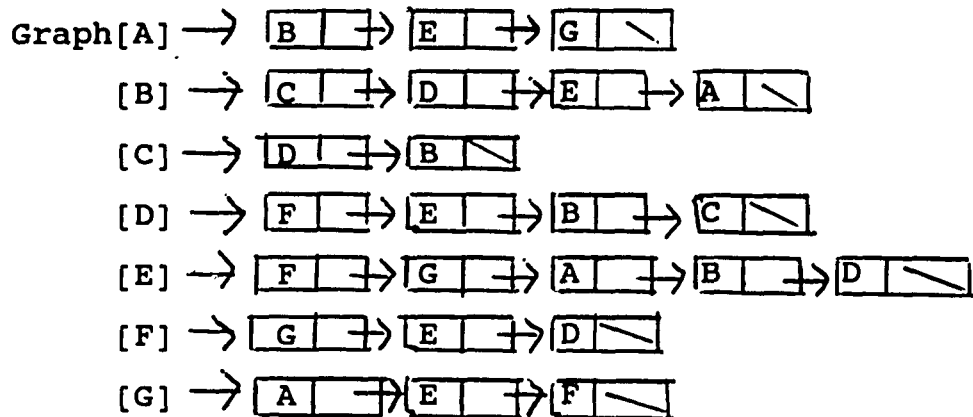
1	2	3
-----	O	-----
4	5	6
-----	X	-----
7	8	9

2. From this graph give a breadth first traversal from A.



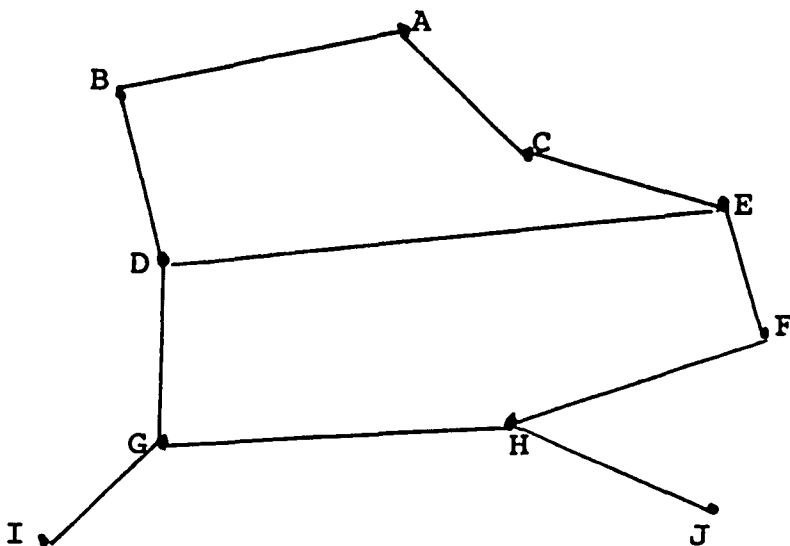
Sheet #5 continued

3. Practice the code used today for the Breadth first traversal. With this adjacency list of the graph of Western states from sheet 2, give a trace table.



Sheet # 6 Depth first traversals, Hamilton circuits

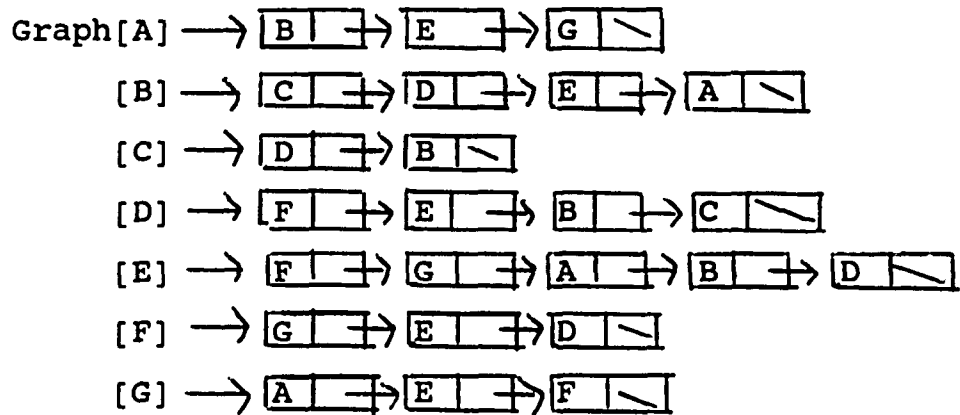
1. For this graph, show a depth first traversal.



2. A graph is connected if for every pair of distinct vertices A, B there is a path between A and B . What could we use to tell if our graph is connected given an adjacency list?

Sheet #6 continued

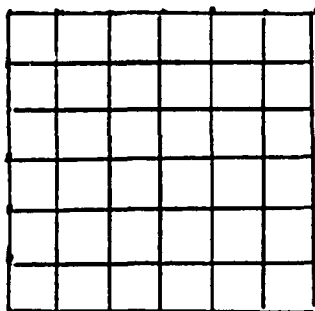
3. From this adjacency list for the graph of the western states from sheet 2, show the trace table for the recursive DFS.



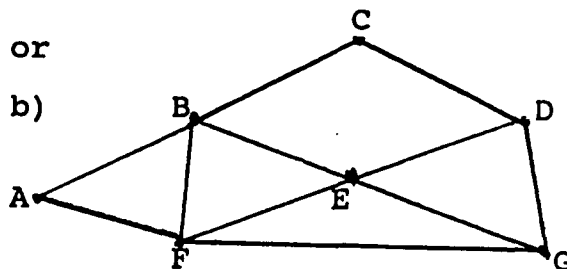
Sheet #6 continued

4. Find a Hamilton circuit or path, if one exists.

a)

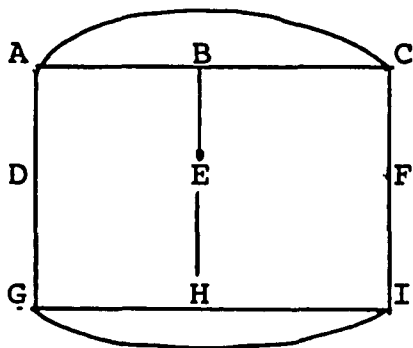


b)

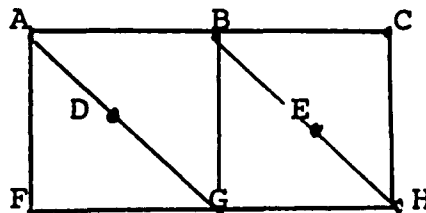


5. Find a Hamilton path, if it exists. Otherwise, if possible, prove no Hamilton circuit exists.

a)



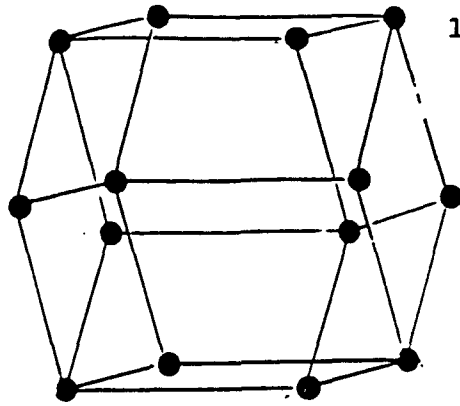
b)



6. What is the difference between an Euler circuit and a Hamilton circuit? Which one is easier to find? Is there an algorithm for each?

Sheet #6 continued

7. (From Gardner, 1971, p.98)



Why can't a Hamilton path be found?

Clue: Count the degree of each vertex; color the vertices of degree 3 black; color the vertices of degree 4 red. (This solution is from a proof by H.S.M. Coxeter)

8. (from Gardner, 1971, p.98)

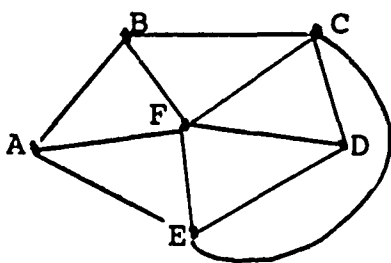
Do you remember studying the Knight's tour problem from recursion? Placing the Knight on a square of the chessboard, can you find a path of continuous Knight's moves that will visit every square once and return the knight to the original square. The line segments connecting consecutive moves of the knight, will form a graph. What does this have to do with a Hamilton circuit? Try it on a 6 by 6 chessboard.

9. Why is it true that every complete graph contains a Hamilton circuit?

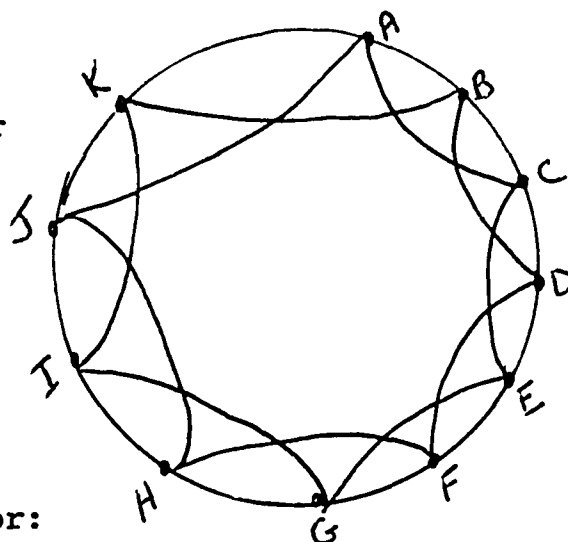
Sheet # 7 Graph coloring

1. Find the chromatic number for

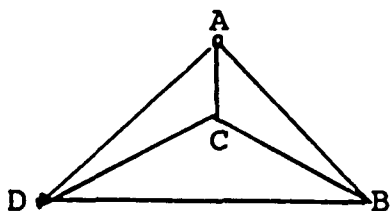
a)



b)



2. Find minimal edge coloring for:



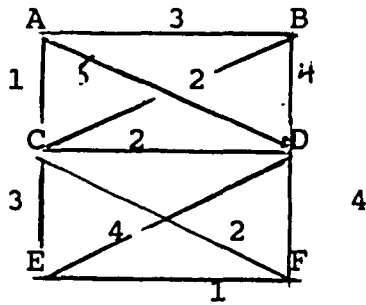
3. How many colors are needed to color the 15 billiard balls in this triangular array with only touching balls of different colors?



4. The ACM SIGCSE has 18 different committees. Each committee is supposed to meet one hour during the convention week. The one constraint is that no member should be scheduled in two different committee meeting at the same time. The program chairperson has a list of committees and its members. She must use as few as hours as possible to schedule the meetings. Help her by modelling this as a graph color problem. Two or more committees may meet at the same time as long as they don't have a common member.

Sheet #8 Minimum Cost Spanning Trees

1. (Graph is from Stubbs and Webre, 1989, p.370)

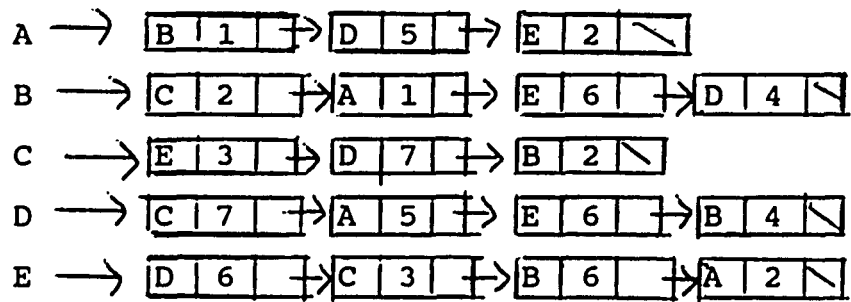


- a) Using the step by step process show Kruskal's method for finding Minimum Cost Spanning Tree.

- b) Using the step by step process, show Prim's method for finding Minimum Cost Spanning Tree.

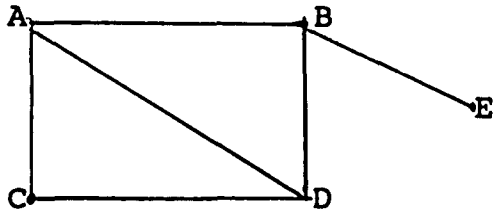
Sheet # 8 continued

2. Using this adjacency list, draw the graph and trace through the code for Prim's algorithm for MCST.



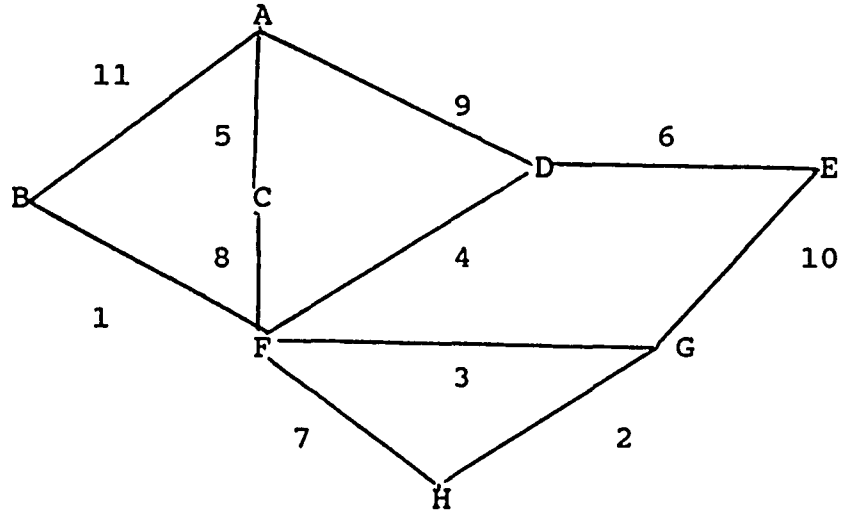
Sheet # 8 continued

3. Find all the spanning trees for this graph.

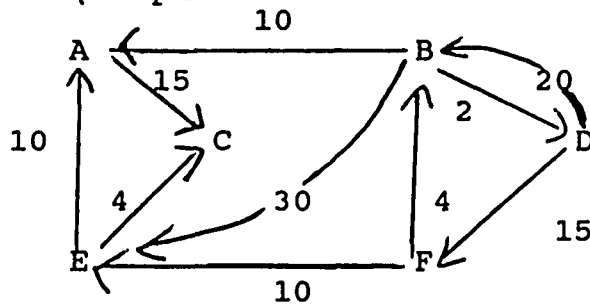


Sheet # 9 Single Source Shortest Path (Dijkstra's algorithm)

1. Use the chart method for finding the single source shortest path from A to every other vertex.



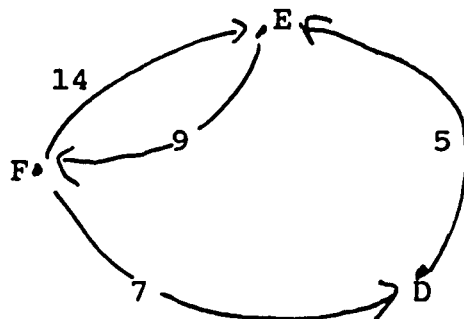
2. (Graph is from Horowitz and Sahni, 1989, p.407)



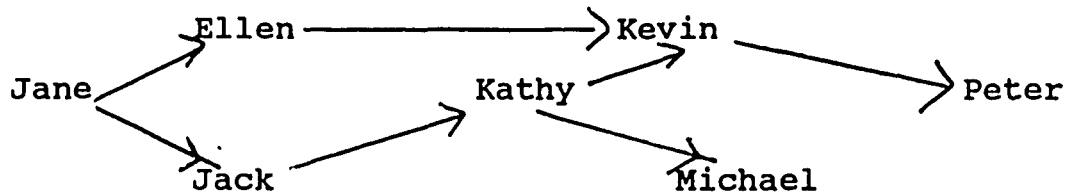
- a) Give the adjacency list for this directed graph. Have each adjacency list in alphabetical order.
- b) Trace the code for Dijkstra's SSSP algorithm.

Sheet # 10 Floyd's Algorithm, Topological Sort,
Efficiency of Algorithms

1. Use the chart method for Floyd's algorithm to find All Pairs Shortest Paths

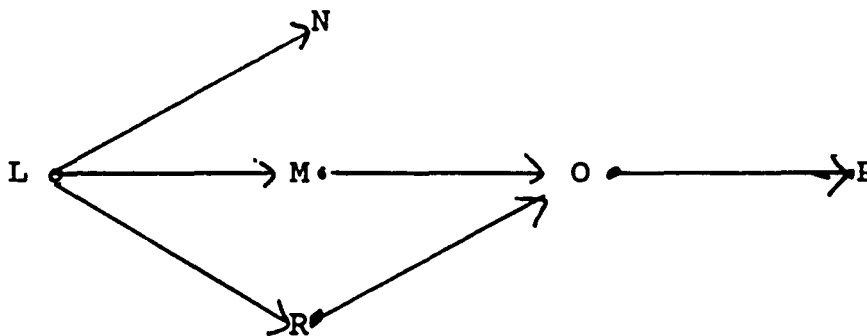


2. Professor Courtney neglected to tell her students they are to have an exam on Wednesday. Aware of who are friends in the class, she set up this graph.



If students are willing to make 1 or more phone calls, give two possible orderings of who gets the message first. Kevin won't believe the message until he gets two phone calls.

3. For this graph, trace the code for topological sort.



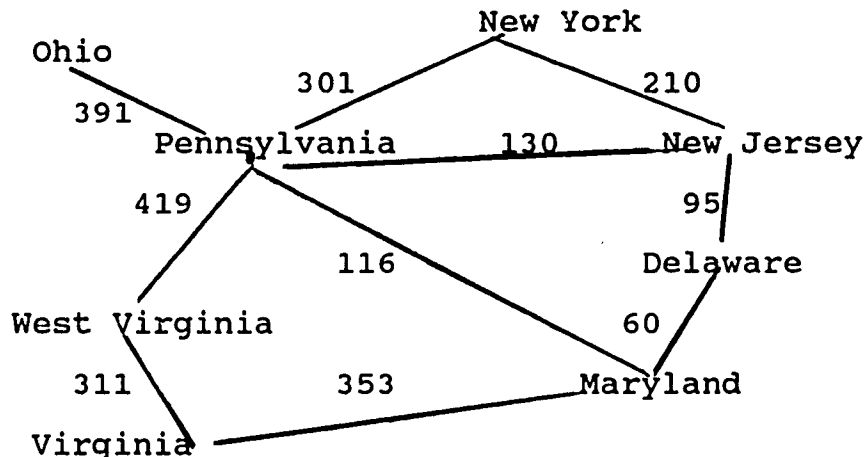
4. Which choice would a 12 year old girl like to have for an allowance?
 - a) To be given the square of the number of days of the year i.e. $1¢, 2¢, 3¢, 4¢, \dots, 365¢$
 - b) To be given 1c on January 1, and double the amount each day of the month, and stop the allowance on January 31st.

5. Would you prefer an algorithm that requires N^5 steps or 2^N steps? Explain.

6. What does it mean that two NP complete problems are reducible to each other?

APPENDIX D

PROGRAM ASSIGNMENT ON GRAPH THEORY



Given the graph, with the highway distances between the capital cities of the given states, write and test a well structured program that

1. Reads in the vertices of each edge and its corresponding weight to an adjacency list.
2. Prints out the adjacency list for the graph.
3. Does a breadth first search from any vertex inputed.
4. Does a depth first search from any vertex inputed.
5. Finds the minimum spanning tree using Prim's algorithm and prints out the edges of this tree.
6. Finds the shortest path between any two vertices inputed using Dijkstra's SSSP algorithm.

You may use the code we have traced in class for each algorithm. Realize that in class all the vertices were referred to by a letter. You will have to set up a look up table whose subscripts are letters and whose values are the names of the vertices. This table will be sent to most procedures. Table should be declared as a record where one of its fields concerns itself with the size of the table and allows the program to work for any size table.

Your code should work for any size graph. Therefore, the graph should be declared as a record with one of its fields holding the number of edges.

APPENDIX E

```

Program graphtraversal(input,output);
{the code has not been tested for many cases;
* if a state is requested that is not in the table, an
* error message is given and
* the operation is not executed'}
uses Dos, Printer;
Const
    blank='          ';
type
    ptr = ^ node;
    node = record
        vertex:char;
        weight:integer;
        link:ptr;
    end;

    graphtype = record
        edges:integer;
        list:array['A'..'Z'] of ptr; { every
        vertex must be labeled by a letter}
    end;

    stringtype= string[15];
    markedtype= array['A'..'Z'] of integer;

    tabletype = record
        elements:array['A'..'Z'] of stringtype;
        lastvertex:char;
    end;

    queuelink = ^ queuenode;
    queuenode = record
        info:char;
        link:queuelink;
    end;
    queuetype= record
        front, rear:queuelink;
    end;

    statustype= (intree, waiting, untouched);
    parenttype= array['A'..'Z'] of char;
    statusarray= array['A'..'Z'] of statustype;
    listtype= ptr;

    stacktype = ^ stackrecord;
    stackrecord = record
        info: char;
        next:stacktype;
    end;

```

```

Var    temp:ptr;
        graph:graphtype;
        edges, I:integer;
        state:stringtype;
        point,pointtwo:char;
        table:tabletype;
        ch:char;
        trace:boolean;
        code:integer;

```

```

Procedure Initializetable(var intable:tabletype );
Var    I:integer;
        character:char;
begin
  if trace then writeln('Enter initializetable');
  for character:= 'A' to 'Z' do
    intable.elements[character]:= blank;
  if trace then writeln('Exit initializetable');
end;

```

```

Function subscript(stringvalue:stringtype;
                  var intable:tabletype): char;
{creates a table of strings and corresponding
* characters - it first searches the
* table to see if the string is there, if it is it
* returns the character, if not it adds it onto the
* table. The table is in a linked list since we don't
* know how many items will be on the list}

var ch:char;
    found:boolean;
begin
  if trace then writeln('enter subscript');
  ch:= 'A';
  found:=false;
  while (intable.elements[ch] <> blank) and
        (not found) do
    begin
      if intable.elements[ch]= stringvalue
      then found:=true
      else ch:= succ(ch);
    end;
  if found
  then subscript:= ch
  else begin  intable.elements[ch]:=stringvalue;
            subscript:= ch;
            intable.lastvertex:=ch;
          end;
  if trace then writeln('exit subscript returning ',
                        ch);
end;

```



```

Function search (stringvalue:stringtype;
                table:tabletype):char;
(This is used after the table has been created to return
* the letter name of the vertex)
var ch:char; found:boolean;
begin
    found:=false;
    ch:='A';
    while (not found) and (ch<=table.lastvertex) do
        if table.elements[ch] = stringvalue
            then found:=true
            else ch:=succ(ch);
    if found
        then search:=ch
        else search:=' ';
end;

(code for reading in the graph and creating the
adjacency list)

Procedure Readinggraph( var graph:graphtype;
                        var intable:tabletype);
Var vertexone, vertextwo: string[15];
    i,j,ch:char;
    temp:ptr;
    number, t:integer;
    infile:text;
    newweight:integer;
    filename:string[20];
    undirected:boolean;
    response:string[5];
begin
    writeln( ' From which file are you reading?');
    readln(filename);
    assign(infile, filename);
    reset(infile);
    for I:= 'A' to 'Z' do
        graph.list[I]:=nil;
    initializetable(intable);
    writeln(' How many edges are there in your
                                graph?');
    readln(infile, graph.edges);
    writeln('There are ', graph.edges, ' edges');
    writeln('Is your graph undirected? Yes/No and
                                press enter');
    readln(response);
    if (response ='Yes') or (response ='yes')
        then undirected:= true
        else undirected:= false;

```

```

for number := 1 to graph.edges do
  begin
    vertexone:=''; vertextwo:='';
    while not eoln(infile) do
      begin
        read(infile, ch);
        vertexone:= vertexone + ch;
      end; readln(infile);
    while not eoln(infile) do
      begin
        read(infile, ch);
        vertextwo:=vertextwo + ch;
      end;
      readln(infile);
      write(vertexone, ' to ', vertextwo);
      i:= subscript(vertexone,intable);
      j:= subscript(vertextwo,intable);
      if trace then writeln(vertexone, '=', i);
      if trace then writeln(vertextwo, '=', j);
      readln(infile,newweight);
      writeln('    with weight ', newweight);
      new(temp);
      temp^.vertex:=j;
      temp^.weight:=newweight;
      temp^.link:= graph.list[i];
      graph.list[i]:= temp;
      if undirected
        then begin
          new(temp);
          temp^.vertex:=i;
          temp^.weight:=newweight;
          temp^.link:=graph.list[j];
          graph.list[j]:=temp;
        end;
      end;
    close(infile);
  end;

Procedure Printgraph(graph:graphtype; table:tabletype);
  var temp:ptr; ch:char;
begin
  writeln; writeln(' The graph is ');
  For ch:= 'A' to table.lastvertex do
    begin
      writeln; write(' **', table.elements[ch]:13);
      temp:=graph.list[ch];
      while temp <> nil do
        begin
          write(' -> ', table.elements[temp^.vertex]:13,
            ', ', temp^.weight);
          temp:= temp^.link;
        end;
      end;
    end;
end;

```

```

Procedure Depthfirst(graph:graphtype; point:char;
                    var table:tabletype);
  Var  id:integer;
        k:char;
        marked:markedtype;

Procedure Visit(graph:graphtype;var marked:markedtype;
               point:char);
  Var  temp:ptr;
  begin
    if trace then writeln('enter visit');
    id:=id + 1; marked[point]:= id;
    writeln(table.elements[point]);
    temp:=graph.list[point];
    if trace then writeln('traversing the list');
    while temp <> nil do
      begin
        if trace then write
          (table.elements[ temp^.vertex]);
          if marked[ temp^.vertex] = 0
            then visit(graph, marked, temp^.vertex);
            temp:= temp^.link;
        end;
      end;

  begin
    id:= 0;
    for k:= 'A' to table.lastvertex do
      marked[k]:=0;
      writeln('A depth first search is ');
      visit (graph, marked, point);
      ( for k:= 'A' to 'H' do
        writeln(table[k], '-', marked[k]))
    end;

Procedure enqueue ( var inqueue:queuetype;
                   inpoint:char);
  var temp:queuelink;
  begin
    new(temp);
    temp^.info:= inpoint;
    temp^.link:= nil;
    if inqueue.front = nil
      then inqueue.front:= temp
      else inqueue.rear^.link:=temp;
    inqueue.rear:=temp;
  end;

Procedure Clearqueue(var inqueue:queuetype);
begin
  inqueue.front:=nil;
  inqueue.rear:=nil;
end;

```

```

Procedure Removequeue(var inqueue:queuetype;
                      var outpoint:char);
begin
  if inqueue.front = nil
  then writeln('cannot remove from queue - empty')
  else if inqueue.front = inqueue.rear
  then begin
        outpoint:= inqueue.front^.info;
        inqueue.front:=nil;
        inqueue.rear:=nil;
      end
  else begin
        outpoint:= inqueue.front^.info;
        inqueue.front:= inqueue.front ^.link;
      end;
end;

```

```

Function Emptyqueue(inqueue:queuetype):boolean;
begin
  if inqueue.front = nil
  then emptyqueue:= true
  else emptyqueue:=false;
end;

```

```

Procedure Breadthfirstsearch(graph:graphtype;
                             point:char; table:tabletype);
Var
  id:integer;
  marked:markedtype;
  queue:queuetype;
  k:char;

  Procedure printqueue(queue:queuetype);
  Var t:queuelink;
  begin
    t:= queue.front;
    if trace then writeln('on the queue now');
    while t <> nil do
      begin
        if trace then write(' ',
                             table.elements[t^.info]);
        t:= t^.link;
      end;
    end;
  end;

```

```

Procedure visit(graph:graphtype;
               var marked:markedtype; point:char);
var temp:ptr;
begin
  if trace then writeln('entering visit of
                        breadthfirst search');
  enqueue(queue,point);
  while not emptyqueue(queue) do
    begin
      printqueue(queue);
      removequeue(queue,point);
      id:= id + 1; marked[point]:=id;
      writeln( table.elements[point]);
      temp:= graph.list[point];
      while temp<> nil do
        begin
          if marked[temp^.vertex] = 0
            then begin
              enqueue(queue, temp^.vertex);
              marked[temp^.vertex]:= -1;
            end;
          temp:= temp^.link;
        end;
      end;
    if trace then writeln('exiting visit of
                          breadthfirstsearch');
  end;

begin
  if trace then writeln('entering
                        breadthfirstsearch');
  id:=0;
  clearqueue(queue);
  for k:= 'A' to table.lastvertex do
    marked[k]:=0;
  writeln ('a breadth first search is ');
  visit(graph,marked, point);
  if trace then writeln('exiting breadthfirstsearch');
end;

Procedure Clear(var list:listtype);
begin
  if trace then writeln('enter clear');
  list:=nil;
  if trace then writeln('exit clear');
end;

```

```

Procedure Addon(var list:listtype; y:char;
                yweight:integer);
Var  traverse, followup, p:ptr;
     found:boolean;
begin
  if trace then writeln('enter addon');
  new(p);
  p^.vertex:=y;
  if trace then writeln('inserting ',y , 'onto the
waiting list with a weight of', yweight);
  p^.weight:=yweight;
  found:= false;
  traverse:=list;
  followup:=list;
  while (not found) and (traverse <> nil) do
    if yweight < traverse^.weight
      then found:= true
      else begin
        followup:=traverse;
        traverse:=traverse^.link;
      end;
  if (list= nil)
    then begin
      p^.link:=nil;
      list:=p;
    end
    else if followup = traverse {it is the first in
                                list}
      then begin
        p^.link:=traverse;
        list:=p;
      end
      else {if found elsewhere in the list or not
            found at all}
        begin
          p^.link:=traverse;
          followup^.link:=p;
        end;
  if trace then writeln('exit addon');
end; {procedure addon}

Procedure Remove (var list:listtype; var x:char);
{removing from a priority queue}
begin
  x:= list^.vertex;
  {xweight:=list^.weight;}
  list:=list^.link;
end;

```

```

Function empty(list:listtype):boolean;
begin
  if trace then writeln('enter empty');
  if list = nil
    then empty:= true
    else empty:=false;
  if trace then writeln('exit empty');
end;

Procedure print(list:listtype);
var t:ptr;
begin
  writeln('what is on the waiting list');
  t:=list;
  while t<> nil do
    begin
      writeln(t^.vertex, 'and the weight is',
              t^.weight);
      t:=t^.link;
    end;
end;

```

```

Procedure Minimumspanningtree(graph:graphtype;
                               table:tabletype);
var  status:statusarray;
     x,y:char;
     edges:integer;
     temp:ptr;
     waitinglist:listtype;
     parent:parenttype;
     waitingweight:markedtype;
     stuck:boolean;

```

```

begin
  if trace then writeln('enter mst');
  x:='A'; status['A']:=intree;
  edges:=0;
  clear(waitinglist);
  for y:= 'B' to table.lastvertex do
    status[y]:= untouched;
  stuck:=false;
  while (edges < graph.edges) and (not stuck) do
    begin
      temp:= graph.list[x];
      while temp <> nil do
        begin
          y:= temp^.vertex;
          if (status[y] = waiting) and (temp ^.
            weight < waitingweight[y])
            then begin
              parent[y]:= x;
              waitingweight[y]:= temp^.weight;
            end;
          if status[y] = untouched
            then begin
              status[y]:= waiting;
              addon(waitinglist, y,
                temp^.weight);
              if trace
                then begin
                  writeln('This is the
                    waiting list in MST');
                  print(waitinglist);
                end;
              parent[y]:= x;
              waitingweight[y]:=temp^.weight;
            end;
          temp:= temp^.link;
        end; (while temp<> nil)
      if empty(waitinglist) then stuck:=true;
      if not empty(waitinglist)
        then begin
          remove(waitinglist, x);
          status[x]:=intree;
          edges:=edges + 1
        end;
    end; (while edges < graph.edges)
    writeln('These are the edges included in the
      Minimum Spanning Tree');
    For y:='B' to table.lastvertex do
      writeln(table.elements[y], ' is connected
to ', table.elements[parent[y]], ' with distance ',
waitingweight[y]);
    if trace then writeln('exit mst');
  end;
end;

```



```

Procedure Updatewaitinglist(var inwaitinglist:listtype;
                           y:char;newdistance:integer);
var  front, trail:ptr;
     found:boolean;

begin
  found:=false;
  front:= inwaitinglist;
  trail:=front;
  while (front <> nil) and (not found ) do
    if front^.vertex = y
      then found:=true
      else begin
           trail:=front;
           front:=front^.link;
         end;
    if (found) and (trail = front)
      then begin
           inwaitinglist:=front^.link;
           addon(inwaitinglist, y, newdistance);
         end
      else if found
         then begin
              trail^.link := front^.link;
              addon(inwaitinglist, y,
                  newdistance);
            end;
    if trace then writeln('exit update waitinglist');
  end;

```

```

Procedure ShortestPath(graph:graphtype; v,w:char;
                      table:tabletype);
var
  status:statusarray;
  parent:parenttype;
  waitinglist:listtype;
  distance:markedtype;
  temp:ptr;
  stuck:boolean;
  x,y:char;

```

```

begin
  if trace then writeln('entering shortest path');
  readln;
  distance[v]:=0;
  parent[v]:=' ';
  clear(waitinglist);
  For y:='A' to table.lastvertex do
    status[y]:= untouched;
  status[v]:=intree;
  x:=v;
  stuck:=false;
  while (x<> w) and (not stuck) do
    begin
      temp:=graph.list[x];
      while temp <> nil do
        begin
          y:=temp^.vertex;
          if (status[y] = waiting) and (distance[x] +
            temp^.weight < distance[y])
            then begin
              parent[y]:= x;
              distance[y]:= distance[x] +
                temp^.weight;
              updatewaitinglist(waitinglist, y,
                temp^.weight);
            end;
          if status[y] = untouched
            then begin
              status[y]:=waiting;
              addon(waitinglist, y,
                temp^.weight);
              parent[y]:= x;
              distance[y]:= distance[x] +
                temp^.weight;
            end;
          temp:= temp^.link;
        end;
      if empty(waitinglist)
        then stuck:=true
        else begin
          if trace
            then begin
              writeln(' This is the waiting
                list in shortest path');
              print(waitinglist);
            end;
          remove(waitinglist,x);
          status[x]:= intree
        end;
    end; { while x<> w and not stuck do}

```

```

(output the path)
writeln('the shortest path in reverse order is');
while x<> ' ' do
  begin
    writeln(table.elements[x]);
    x:= parent[x];
  end;

if trace then writeln('exit shortest path');
end;

Procedure Printvertices (table:tabletype);
  Var ch:char;
  begin
    For ch:= 'A' to table.lastvertex do
      write(table.elements[ch] ,', ');
    writeln;
  end;

Function emptystack(stack:stacktype):boolean;
begin
  if stack = nil
    then emptystack:=true
    else emptystack:=false;
end;

Procedure clearstack(var stack:stacktype);
  begin
    stack:=nil;
  end;

Procedure Push(var stack:stacktype; newelement:char);
var temp:stacktype;
begin
  if trace then writeln('entering push stack');
  new(temp);
  temp^.info:= newelement;
  temp^.next:= stack;
  stack:=temp;
  if trace then writeln('exiting push stack');
end;

Procedure Pop(var stack: stacktype;
              var oldelement:char);
begin
  if trace then writeln('entering pop stack');
  if emptystack(stack)
    then writeln('stack empty - data cannot be
                 returned')
    else begin
      oldelement:= stack^.info;
      stack:= stack^.next;  end;
  if trace then writeln('exiting pop stack');
end;

```

```

Procedure topologicalorder(var graph:graphtype;
                           table:tabletype);
Var temp:ptr;done:boolean;
    stack:stacktype;
    I,j,k:char;
    count:array['A'..'Z'] of integer; {parallel array to
    each list in the graph to hold incount}
begin
  writeln(' Give the values for count, the number of
          edges coming in to a vertex');
  For I:= 'A' to table.lastvertex do
    begin
      writeln ('How many edges come in to ',
              table.elements[i]);
      readln(Count[i]);
    end;
  if trace
    then begin
      for I:='A' to table.lastvertex do
        writeln( 'I=', I, 'count is ', count[i]);
      end;
  clearstack(stack);
  For I:= 'A' to table.lastvertex do
    if count[I]= 0
      then begin
        push(stack,I);
      end;
  J:='A';
  done:=false;
  writeln( 'A topological ordering of the graph is ');
  while (J <= (table.lastvertex)) and (not done) do
    begin
      if emptystack(stack)
        then begin
          writeln('loop formed,no topological
                  sort ');
          done:=true;
        end
      else begin
        pop(stack,I);
        writeln(table.elements[I]);
        temp:= graph.list[I];
        while(temp<> nil) do
          begin
            k:= temp^.vertex;
            count[k]:= count[k] - 1;
            if count[k] = 0
              then push (stack, k);
            temp:= temp^.link;
          end; {of while temp <> nil do}
        end; {else}
        J:= succ(J);
      end;
    end;
end;
end;

```

```

begin (main program)
  trace:= false;
  readingraph(graph, table);
  printgraph(graph, table);writeln; writeln;
  writeln('please choose 1,2,3,4,5,6 and press enter');
  writeln('1  for  depthfirst search');
  writeln('2  for  breadthfirst search');
  writeln('3  for  minimum spanning tree');
  writeln('4  for  shortest path between two vertices');
  writeln('5  for  topological sort');
  writeln('6  for  exiting');
  readln(code);
  case code of
    1: begin
      writeln('where would you like to start your depth
              first search from?');
      Printvertices(table);
      state:='';
      while not eoln do
        begin
          read(ch);
          state:= state + ch;
        end;
      point:= search(state, table);
      if point = ' '
        then writeln('Your vertex name is not a choice')
        else  depthfirst(graph, point, table);
          end;

      2: begin
      writeln('where would you like to start your breadth
              first search from?');
      Printvertices(table);
      state:='';
      while not eoln do
        begin
          read(ch);
          state:= state + ch;
        end;
      writeln('starting breadth first search from ',
              state);
      point:= search(state,table);
      if point = ' '
        then writeln('Your vertex name is not a choice')
        else  breadthfirstsearch(graph, point,table);
          end;

      3: begin
      if trace then writeln('before mst');
      Minimumspanningtree(graph,table);
      if trace then  writeln('after mst');
        end;

```

```
    4: begin
Printvertices(table);
writeln('Name the state of origin');
state:='';
while not eoln do
    begin
        read(ch);
        state:= state + ch;
    end;
readln;
point:= search(state,table);
if point = ' '
    then writeln('Your vertex name is not a choice');
writeln('Name the state of destination');
state:='';
while not eoln do
    begin
        read(ch);
        state:= state + ch;
    end;
pointtwo:= search(state,table);
if pointtwo = ' '
    then writeln('Your vertex name is not a choice');
if (point <> ' ') and (pointtwo <> ' ')
    then Shortestpath(graph,point,pointtwo, table)
    else writeln('Shortestpath will not work');
    end;

5: topologicalorder(graph, table);

6:
end;
end.
```

APPENDIX F
QUESTIONNAIRE ON THE USE OF THE SOURCE BOOK

Please answer as many questions as possible from your teaching. It is realized that you were not able to use the source book in its entirety, for each teacher has his/her own style of teaching.

TECHNICAL

1. There was an attempt to keep notation consistent, such as vertices always named by capital letters. Was this helpful? Did you find any inconsistencies?

2. The lines of code were numbered. Was this helpful for class lectures and discussions?

3. Were the diagrams clear and informative?

4. Was it beneficial using the graph of the Southern states throughout the class notes and the graph of the Western states for homework? Or would you have liked using different graphs for more practice?

5. Do you have any other comments on the technical aspect?

PEDAGOGICAL

6. Were you able to share the historical tidbits with the students? How did they enjoy this? Do you know of any other background information?

7. The tracing of code was very methodical throughout the source book.

a) How much emphasis do you place on tracing code?

b) Did you use any of the traces I gave in the source book? Which ones?

c) Which traces did you find worthwhile?

d) Which traces were unnecessary?

e) Do you use any other methods for tracing? Explain.

f) What do you think of tracing the recursive code for depth first search?

8. a) Were the homework sheets beneficial?

b) Which examples were "great"? Why?

c) Which examples were weak? Why?

d) Do you have any other problems which would benefit the students' learning process?

9. Comment on the use of the Chart Method for Dijkstra's SSSP problem.

10. Notice the difference between the implementation for graph in the source book and in the program. The graph in the source book was left as simple as possible to avoid cumbersome code. In the program, graph is a record with the number of edges as a field. And in topological sort, the incount is kept as a separate parallel array instead of an extra field just to be consistent with the original implementation of graph. How exact should we be in our teaching? Please comment.

11. Do you have any other comments on the pedagogy of the source book?

THEORETICAL

12. Do you think it is necessary to teach the introductory ideas of graph theory - isomorphism, planar graphs, graph coloring - to master the algorithms that are usually taught in computer science courses -depth first, minimum cost spanning trees, single source shortest path?

13. Is there a topic in the source book you would omit?

14. Is there a topic that should be added to the source book?

15. Nell Dale, a writer of Pascal textbooks and influential in Undergraduate Computer Science Education, questioned at the 1990 SIGCSE conference, "Whatever happened to CS7 - the Algorithms course?" Would you agree that much of the attention in CS education has been centered on CS1, CS2, and Software Engineering?

16. How do you like mixing mathematics with computer science? How about your students?

17. Comment on the different types of proof, i.e. induction in the 5 Color Theorem, and the proof by contradiction in the lemma on p.32C.

18. Are your students enthusiastic about graph theory?

19. Assuming the students have already learned the mathematical definition of Big Oh, how was the discussion of time complexity in the source book? Do you have any suggestions for improving it?

20. How does the learning of graph theory help in other courses in Computer Science?

21. Do you have any other comments on the theoretical aspect of the source book?

22. Do you have any other comments on the source book in general?
