



Knowledge Representation in Mathematics a Case Study in Graph Theory

Epstein, Susan Lynn

<https://scholarship.libraries.rutgers.edu/esploro/outputs/technicalDocumentation/Knowledge-Representation-in-Mathematics-a-Case/991031549863904646/filesAndLinks?index=0>

Epstein, S. L. (1983). Knowledge Representation in Mathematics a Case Study in Graph Theory. Rutgers University. <https://doi.org/10.7282/T3WH2THR>
Document Version: Version of Record (VoR)

This work is protected by copyright. You are free to use this resource, with proper attribution, for research and educational purposes. Other uses, such as reproduction or publication, may require the permission of the copyright holder.

Downloaded On 2024/03/18 18:12:56 -0400

April, 1983

**KNOWLEDGE REPRESENTATION IN
MATHEMATICS A CASE STUDY IN
GRAPH THEORY**

by

Susan Lynn Epstein

DCS-TR-134

Laboratory for Computer Science Research
Hill Center for the Mathematical Sciences
Busch Campus
Rutgers University
New Brunswick, NJ 08903

Copyright (C) 1983
Susan L. Epstein
ALL RIGHTS RESERVED

ACKNOWLEDGMENTS

Sincere thanks are due to my advisor, Professor N. S. Sridharan, for his support and encouragement in the development and construction of this work. His breadth and depth of knowledge serve as a perpetual inspiration, and his incisive humor as a welcome leavening. He taught me the pleasures of research and the power of perspective.

Professor Ann Yasuhara has been a substantial influence in my intellectual development. Her insistence on clarity, in writing and in thought, has been invaluable. Her efforts during the preparation of my proposal and this document were heroic, and I am truly grateful.

Many others deserve thanks for their contributions, direct or indirect, to this document. Professor Tom Mitchell introduced me to artificial intelligence and persisted in challenging my own. His attention to this work has been sincerely appreciated. Professor Marv Paul is to be thanked for his inquiring study as a member of my committee. Professor Saul Amarel has been a source of inspiration and encouragement, within and beyond the context of this work. Professor Patricia Kenschaft of Montclair State College has provided determination and perspective when my own supply ran low. Don Watrous and Professors Barbara Ryder and Don Smith have provided technical support and patience beyond any reasonable bounds. My thanks go, too, to professors of mathematics Neal McCoy and Alice Dickinson of Smith College, who long ago taught me the elegance of mathematics and the joys of constructing one's own knowledge representation.

My children, Allison, Brian and Gillian, have been remarkably patient and supportive throughout my intensive graduate school career. I thank them and my husband, Wally, whose cooperation and encouragement made it all possible.

ABSTRACT OF THE THESIS

KNOWLEDGE REPRESENTATION IN MATHEMATICS A CASE STUDY IN GRAPH THEORY

by Susan Lynn Epstein

Thesis Director: Professor N. S. Sridharan

In this dissertation we present our work on representational languages for graph theory. We have shown that a knowledge representation can be structured to provide both expressive and procedural power.

Our major research contributions are three. First, we have defined representations of infinite sets and recommended that mathematical concepts be considered as sets of objects with relations among them. Second, we have demonstrated how a carefully controlled hierarchy of representations is available through formal languages. Third, we have employed a recursive formulation of concepts which enables their application to many of the behaviors of a research mathematician.

Two major families of representations are described: edge-set languages and recursive languages. The edge-set languages have finite expressive power and an interesting potential for hashing digraphs, characterizing classes of graphs and detecting differences among them. The recursive languages have extensible expressive power and impressive procedural power. Recursive languages appear to be an excellent implementation technique for artificial intelligence programs in mathematical research.

Our results enable us to compare the complexity of mathematical concepts

(via floors). Concepts represented in our languages can be inverted (to test for the presence of a property) and merged (to combine properties). Conjectures are available through simple search, and most theorems easily proved under the representation.

TABLE OF CONTENTS

1. Artificial Intelligence and Graph Theory	1
1.1. Overview	1
1.2. Background	4
1.3. Graph Theory and Its Representation	7
1.4. Languages for Graph Theory	8
1.5. Questions and Answers	10
1.6. Some Fundamental Definitions	11
1.6.1. Basic Graph Terminology	12
1.6.2. Graph Properties, Characteristics and Descriptions	13
1.6.3. Graph Terminology and Graph Grammars	14
2. Edge-Set Graph Languages	17
2.1. General Overview	17
2.1.1. Language L_1 Summary	18
2.1.2. Language L_2 Summary	20
2.1.3. Language L_3 Summary	20
2.1.4. Summary of Languages L_{1n} , L_{2n} and L_{3n}	21
2.1.5. Language L_1^* Summary	21
2.2. Language L_1	22
2.2.1. A Grammar for Language L_1	22
2.2.2. L_1 for Undirected Graphs	26
2.2.3. L_1 for Directed Graphs	29
2.2.4. An L_1 Graph Generator	32
2.2.5. An L_1 Testing Algorithm	34
2.2.6. Transition from L_1 to L_2	35
2.3. Language L_2	35
2.3.1. A Grammar for Language L_2	35
2.3.2. L_2 for Undirected Graphs	36
2.3.3. L_2 for Directed Graphs	41

2.3.4. Algorithms for Generating and Testing in L_2	41
2.3.5. A Comparison of L_1 and L_2	44
2.4. Language L_3	45
2.4.1. A Grammar for Language L_3	45
2.4.2. L_3 for Undirected Graphs	46
2.4.3. L_3 for Directed Graphs	49
2.4.4. Algorithms for Generating and Testing in L_3	50
2.4.5. A Comparison of L_3 with L_2	50
2.5. The Language L_1^*	51
2.5.1. A Grammar for Language L_1^*	51
2.5.2. L_1^* for Undirected Graphs	52
2.5.3. Evaluation of L_1^*	57
2.6. The Edge-Set Languages: a Review	58
3. Recursive Languages	60
3.1. Graph Construction	60
3.2. Recursive Graph Grammars	64
3.3. The Components of a Recursive Language	66
3.4. The Floor of a Graph Property	72
3.5. Inversion	74
3.6. Automated Inversion	75
3.7. Readily Invertible Graph Properties	81
3.7.1. Acyclic Graphs	82
3.7.2. Trees	85
3.7.3. Loopfree Graphs	88
3.7.4. Chains	90
3.7.5. Cycles	94
3.7.6. Stars	96
3.7.7. Wheels	98
3.7.8. Complete Graphs	101
3.7.9. Graphs with an Even Number of Vertices	102
3.7.10. Graphs with an Odd Number of Vertices	105
3.7.11. Graphs with an Even Number of Edges	106
3.7.12. Graphs with an Odd Number of Edges	108
3.7.13. Eulerian Graphs	111

3.7.14. Graphs with K Vertices	114
3.7.15. Graphs with K Edges	115
3.7.16. Graphs of Minimum Degree K	119
3.7.17. Graphs of Maximum Degree K	120
3.7.18. Pinwheels on Hubs of Size h	123
3.7.19. Graphs with K Components	126
3.7.20. Regular Graphs	129
3.7.21. Connected Graphs	138
3.7.22. Biconnected Graphs	139
3.7.23. k-Connected Graphs	142
4. Advanced Topics in Recursive Languages	147
4.1. Extended Recursive Languages	147
4.1.1. Calculating the Number of Vertices and Edges in a Graph	148
4.1.2. Calculating the Degree of a Vertex	151
4.2. The Loop as Marker	153
4.2.1. Calculating the Maximum Vertex Degree in a Graph	153
4.3. The Loop as Label	156
4.3.1. Bipartite Graphs	156
4.3.2. Complete Bipartite Graphs	158
4.3.3. K-Vertex-Covered Graphs	161
4.3.4. Graphs with K Independent Vertices	164
4.4. Labelling/Coloring Graphs	165
4.4.1. K-Colored Graphs	168
4.4.2. K-Chromatic Graphs	170
4.4.3. Graphs with Vertex Covering Number K	176
4.4.4. Graphs with Independence Number K	182
4.4.5. Graphs with Labelled Edges	184
4.4.6. Graphs with Circumference K	185
4.4.7. Graphs with Edge Covering Number K	189
4.4.8. Graphs with a k-Factor	193
4.4.9. K-Factorable Graphs	196
4.5. Subsumption	201
4.6. Merger	204
4.7. NP-Completeness and R-Properties	213
4.7.1. Subgraph Properties and Two-Stage Algorithms	215

4.7.2. Graph Properties with Elaborate Seed Sets	217
4.7.3. NP-Completeness and the Recursive Formulation	218
5. Conclusions	219
5.1. Languages for Graph Properties	219
5.2. Edge-Set Language Results	222
5.3. R-Language Results	223
5.3.1. Expressive Power	223
5.3.2. The $\langle P, L, \Sigma \rangle$ Formulation	226
5.3.3. Floors	226
5.3.4. Inversion, Subsumption and Merger	230
5.3.5. Complexity and Redundancy	231
5.3.6. Boolean Properties	233
5.4. Applications	238
5.5. Open Questions	242
5.6. Implications of This Work	243
Appendix A. Key to Notation	245
Appendix B. Investigation of the Language L_2 for Undirected	247
B.1. The Program L2	247
B.2. L2 Output	250
Appendix C. Investigation of the Language L_2 for Directed	252
C.1. The Program L2DI	252
C.2. L2DI Output	257
Appendix D. Investigation of the Language L_3 for Undirected	259
D.1. The Program L3	259
D.2. L3 Output	262
Appendix E. Investigation of the Language L_3 for Directed	265
E.1. The Program L3DI	265
E.2. L3DI Output	269
REFERENCES	271
INDEX	275

LIST OF FIGURES

Figure 1-1:	Forbidden Subgraphs for a Line Graph	2
Figure 2-1:	A Venn Diagram for Undirected Graphs	26
Figure 2-2:	A Venn Diagram for Directed Graphs	29
Figure 2-3:	A Venn Diagram for Undirected Graphs in L_1^*	53
Figure 2-4:	A Preliminary Venn Diagram for Directed Graphs in L_1^*	58
Figure 3-1:	An Algorithm to Recursively Construct a Target Graph	61
Figure 3-2:	An Algorithm to Recursively Generate Graphs	61
Figure 3-3:	A Sample Run of GENERATE	62
Figure 3-4:	An Algorithm to Generate Graphs without Edges	63
Figure 3-5:	A Sample Run of EDGELESS	63
Figure 3-6:	Orderings for Property Languages	73
Figure 3-7:	EDGELESS ⁻¹ in Operation	79
Figure 3-8:	GENERATE ⁻¹ in Operation	80
Figure 3-9:	The Behavior of f and f^{-1} on the Set of All Graphs	82
Figure 3-10:	Some Acyclic Graphs	83
Figure 3-11:	A Sample Run of ACYCLIC	84
Figure 3-12:	ACYCLIC ⁻¹ in Operation	85
Figure 3-13:	Some Trees	86
Figure 3-14:	A Sample Run of TREE	86
Figure 3-15:	TREE ⁻¹ in Operation	87
Figure 3-16:	Some Loopfree Graphs	88
Figure 3-17:	A Sample Run of LOOPFREE	89
Figure 3-18:	LOOPFREE ⁻¹ in Operation	90
Figure 3-19:	Some Chains	90
Figure 3-20:	A Sample Run of CHAIN	91
Figure 3-21:	CHAIN ⁻¹ in Operation	93
Figure 3-22:	CHAIN ₂ ⁻¹ in Operation	94
Figure 3-23:	Some Cycles	94

Figure 3-24:	A Sample Run of CYCLE	95
Figure 3-25:	CYCLE ⁻¹ in Operation	96
Figure 3-26:	Some Star Graphs	97
Figure 3-27:	A Sample Run of STAR	97
Figure 3-28:	STAR ⁻¹ in Operation	98
Figure 3-29:	Some Wheels	99
Figure 3-30:	A Sample Run of WHEEL	99
Figure 3-31:	WHEEL ⁻¹ in Operation	100
Figure 3-32:	Some Complete Graphs	101
Figure 3-33:	A Sample Run of COMPLETE	102
Figure 3-34:	COMPLETE ⁻¹ in Operation	103
Figure 3-35:	Some Graphs with an Even Number of Vertices	103
Figure 3-36:	A Sample Run of EVEN-N	104
Figure 3-37:	EVEN-N ⁻¹ in Operation	105
Figure 3-38:	Some Graphs with an Odd Number of Vertices	105
Figure 3-39:	A Sample Run of ODD-N	106
Figure 3-40:	ODD-N ⁻¹ in Operation	107
Figure 3-41:	Some Graphs with an Even Number of Edges	107
Figure 3-42:	A Sample Run of EVEN-M	108
Figure 3-43:	EVEN-M ⁻¹ in Operation	109
Figure 3-44:	Some Graphs with an Odd Number of Edges	109
Figure 3-45:	A Sample Run of ODD-M	110
Figure 3-46:	ODD-M ⁻¹ in Operation	110
Figure 3-47:	Some Eulerian Graphs	111
Figure 3-48:	A Sample Run of EULERIAN	111
Figure 3-49:	EULERIAN ⁻¹ in Operation	113
Figure 3-50:	Some Graphs with 3 Vertices	114
Figure 3-51:	5-VERTICES in Operation	114
Figure 3-52:	4-VERTICES ⁻¹ in Operation	115
Figure 3-53:	Some Graphs with 3 Edges	116
Figure 3-54:	5-EDGES in Operation	116
Figure 3-55:	4-EDGES ⁻¹ in Operation	118
Figure 3-56:	Some Graphs of Minimum Degree 3	119
Figure 3-57:	MIN-2 in Operation	119
Figure 3-58:	MIN-4 ⁻¹ in Operation	121

Figure 3-59:	Some Graphs of Maximum Degree 3	121
Figure 3-60:	MAX-5 in Operation	122
Figure 3-61:	MAX-4 ⁻¹ in Operation	123
Figure 3-62:	Some Pinwheels	124
Figure 3-63:	PINWHEEL in Operation	125
Figure 3-64:	PINWHEEL ⁻¹ in Operation	126
Figure 3-65:	Some Graphs with Three Components	127
Figure 3-66:	2-COMPONENTS in Operation	127
Figure 3-67:	1-COMPONENTS ⁻¹ in Operation	129
Figure 3-68:	Some Regular Graphs	130
Figure 3-69:	EVEN-REGULAR in Operation for $k = 4$	132
Figure 3-70:	EVEN-REGULAR ⁻¹ in Operation for $k = 2$	134
Figure 3-71:	ODD-REGULAR in Operation for $k = 5$	135
Figure 3-72:	ODD-REGULAR ⁻¹ in Operation for $k = 3$	137
Figure 3-73:	Some Connected Graphs	138
Figure 3-74:	CONNECTED in Operation	139
Figure 3-75:	CONNECTED ⁻¹ in Operation	140
Figure 3-76:	Some Biconnected Graphs	140
Figure 3-77:	BICONNECTED in Operation	141
Figure 3-78:	BICONNECTED ⁻¹ in Operation	143
Figure 3-79:	Some 5-Connected Graphs	144
Figure 3-80:	4-CONNECTED in Operation	144
Figure 3-81:	3-CONNECTED ⁻¹ in Operation	146
Figure 4-1:	VERTICES in Operation	148
Figure 4-2:	EDGES in Operation	149
Figure 4-3:	VERTICES ⁻¹ in Operation	151
Figure 4-4:	EDGES ⁻¹ in Operation	151
Figure 4-5:	DEGREE in Operation	152
Figure 4-6:	DEGREE ⁻¹ in Operation	153
Figure 4-7:	MAX in Operation	154
Figure 4-8:	MAX ⁻¹ in Operation	155
Figure 4-9:	Some Bipartite Graphs	156
Figure 4-10:	BIPARTITE in Operation	157
Figure 4-11:	BIPARTITE ⁻¹ in Operation	158
Figure 4-12:	Some Complete Bipartite Graphs	159

Figure 4-13:	COMPLETE-BIPARTITE in Operation	159
Figure 4-14:	COMPLETE-BIPARTITE ⁻¹ in Operation	161
Figure 4-15:	Some 5-Vertex-Covered Graphs	161
Figure 4-16:	4-VERTEX-COVERED in Operation	162
Figure 4-17:	3-VERTEX-COVERED ⁻¹ in Operation	163
Figure 4-18:	Some Graphs with 3 Independent Vertices	164
Figure 4-19:	3-INDEPENDENT in Operation	164
Figure 4-20:	2-INDEPENDENT ⁻¹ in Operation	166
Figure 4-21:	Some 3-Colored Graphs	168
Figure 4-22:	4-COLORED in Operation	169
Figure 4-23:	2-COLORED ⁻¹ in Operation	170
Figure 4-24:	Some 3-Chromatic Graphs	171
Figure 4-25:	4-CHROMATIC in Operation	172
Figure 4-26:	The Generation of $W_{1,5}$	172
Figure 4-27:	The Grotzsch Graph	173
Figure 4-28:	Generating the Grotzsch Graph	173
Figure 4-29:	2-CHROMATIC ⁻¹ in Operation	175
Figure 4-30:	Some Graphs with Vertex Covering Number 5	177
Figure 4-31:	VERTEX-COVER in Operation	178
Figure 4-32:	VERTEX-COVER ⁻¹ in Operation	182
Figure 4-33:	Some Graphs with Independence Number 3	182
Figure 4-34:	INDEPENDENCE-3 in Operation	183
Figure 4-35:	INDEPENDENCE-4 ⁻¹ in Operation	185
Figure 4-36:	Some Graphs with Circumference 5	186
Figure 4-37:	A Graph and its Blocks	186
Figure 4-38:	CIRCUMFERENCE-6 in Operation	187
Figure 4-39:	CIRCUMFERENCE-3 ⁻¹ in Operation	188
Figure 4-40:	Some Graphs with Edge Covering Number 4	189
Figure 4-41:	The Seed Graphs for Edge Cover 4	190
Figure 4-42:	6-EDGE-COVER in Operation	191
Figure 4-43:	3-EDGE-COVER ⁻¹ in Operation	192
Figure 4-44:	Some Graphs with 3-Factors	193
Figure 4-45:	4-FACTOR in Operation	194
Figure 4-46:	2-FACTOR ⁻¹ in Operation	197
Figure 4-47:	Some 2-Factorable Graphs	197

Figure 4-48:	3-FACTORABLE in Operation	199
Figure 4-49:	2-FACTORABLE⁻¹ in Operation	200
Figure 4-50:	Some Hamiltonian Graphs	214
Figure 4-51:	HAMILTONIAN in Operation	214
Figure 4-52:	HAMILTONIAN⁻¹ in Operation	215
Figure 5-1:	Graph Properties with Edge-Set L-Language Grouped by Floors	227
Figure 5-2:	Graph Properties with Edge-Set L-Language Ranked by P-Language and Σ-Language	229
Figure 5-3:	A Graph with Variable Testing Time	232
Figure 5-4:	Some Graphs and Their Planarity	234
Figure 5-5:	The Construction of a Homeomorph to K_5	234
Figure 5-6:	A Sample Run of NON-PLANAR	235
Figure 5-7:	A Sample Run of PLANAR	236

LIST OF TABLES

Table 2-1:	Equivalence Classes for Undirected Graphs in L_1	28
Table 2-2:	Equivalence Classes for Directed Graphs in L_1	31
Table 2-3:	Properties of Undirected Graphs in L_2	37
Table 2-4:	Unique L_2 Characterizations for Undirected Graphs	38
Table 2-5:	Results of Program L2 on Undirected Graphs	39
Table 2-6:	Results of L2DI on Directed Graphs	42
Table 2-7:	Refinement of L_1 by L_2	45
Table 2-8:	Characterizations for Undirected Graphs in L_3	48
Table 2-9:	Results of Program L3DI on Directed Graphs	49
Table 2-10:	Undirected Graph Signatures in L_1^*	55
Table 3-1:	Primitive Operators for R-Grammars	66
Table 3-2:	Some Composite Operators for R-Grammars	68
Table 5-1:	Edge-Set Language Properties	222
Table 5-2:	Graph Properties Studied under Recursive Generation	225

CHAPTER 1

ARTIFICIAL INTELLIGENCE AND GRAPH THEORY

...if the science of number were merely analytical, or could be analytically derived from a few synthetic intuitions, it seems that a sufficiently powerful mind could with a single glance perceive all its truths; nay one might even hope that some day a language would be invented simple enough for these truths to be made evident to any person of ordinary intelligence.

--Poincare

1.1. Overview

"G is a line graph," says Theorem 8.4 in Harary's graph theory textbook, "if and only if none of the graphs in Figure 1-1 is an induced subgraph of G." We do not need to understand the terminology of the theorem to have the pictures arouse our curiosity. What do those graphs have in common with each other? Why, precisely those and no others? Consider too Statement 2.1.5 from Bondy and Murty's text "Let G be a graph [on v vertices] with $v-1$ edges. The following statements are equivalent

- (a) G is connected
- (b) G is acyclic
- (c) G is a tree"

What intertwines those properties? Are any others related to them? What other sets of properties are so commingled? Graph theory abounds with such questions. The challenge for a mathematician is to identify, discover and describe such properties and the relations among them. Assume we wish to address some of these questions with a computer. How would we go about it?

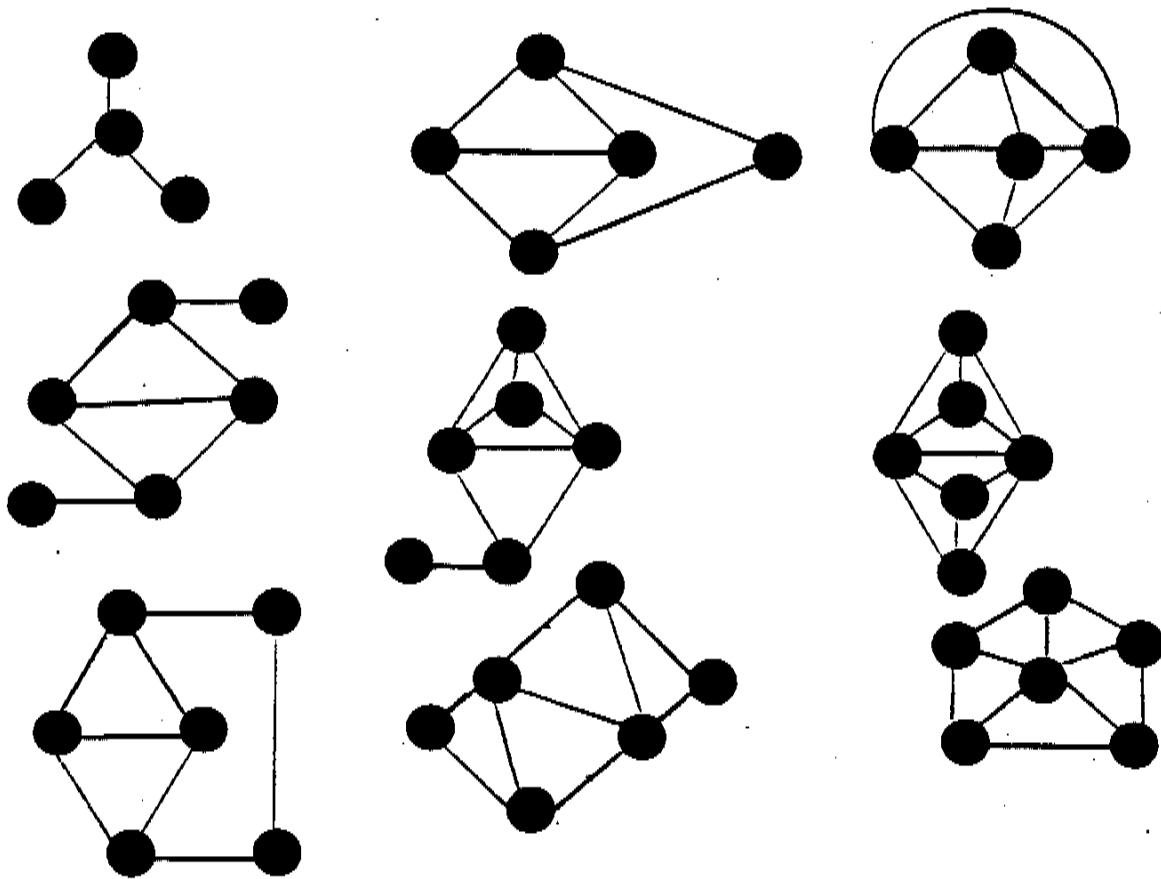


Figure 1-1: Forbidden Subgraphs for a Line Graph

We reflect first on the process of mathematical research. The questions posed above are those a mathematician might pose while doing research. Confronted with a mass of data (axioms, definitions, examples, theorems, algorithms), the research mathematician focuses selectively on an interesting subset [Pascal 64, Hardy 40], intuitively conjectures additional data [Poincare 70] and then attempts to satisfy the demands of rigor with formal definitions and proofs. How can we even begin to computerize a process so laden with value judgements and vagueness?

Consider the diverse role of language in mathematical research. The mathematician carefully formalizes a research result in the language of definitions and proofs. The promulgation of the result, however, is more likely to be in natural

(spoken, non-technical) language with recourse to vernacular and analogy to other fields. Yet the language (or languages) in which the focusing and conjecturing take place, we suspect, is quite different from the other two languages. (Recall the tremendous assistance a diagram offers to the construction of a proof in plane geometry, or the power functional analysis derives from viewing certain problems in the context of series.)

This multiplicity of representations, and the facility with which a mathematician moves from one representation to another, is significant. If we are interested in exploring graph theory on a computer, we require some way of representing graph theory to the machine. Until now, representation of mathematical knowledge for computers has been tailored to a single specific purpose. Example generation utilized one representation, and theorem proving another.

It is our thesis that *a single representation can support many of the behaviors observed in mathematical research*. This dissertation explores the thesis in the domain of graph theory. We consider many representations for graph theory and evaluate them. Our evaluation is both *theoretical* (calculating the power of the representation based only on its definitional structure) and *empirical* (observing the portions of the representation actually applicable to and of interest in graph theory).

In the course of research, a mathematician hopes to deepen her understanding of objects and relations among them. Understanding a concept means, among other things, being able to apply it. Imagine that a machine is presented with graph property p as an algorithm for generating the set of graphs with that property. Now the machine is confronted with an arbitrary graph. Can it determine whether or not that graph has property p ? Such behavior, creating a testing algorithm from a generating algorithm, we will call "inversion." Inversion is an example of machine learning and of automatic programming, both of substantial interest in computer science.

We have sketched a complex and challenging series of problems. Our next step is to define the segments we will examine. This chapter provides a framework for our study. First we place the task in its contexts: artificial intelligence, knowledge representation, mathematics and graph theory. In a subsequent section we define graph theory and formulate criteria for evaluating its representation. We then describe our approach to knowledge representation based upon formal languages. We pose the central questions and indicate, informally, where the answers lie. The basic terminology required for graph theory and its representation in formal languages completes the chapter.

In Chapter 2 we analyze a family of representations called the edge-set languages. They are used, in turn, as a foundation for the recursive languages. In Chapter 3 we present elementary concepts for recursive languages as a graph representation and begin an empirical examination of these recursive languages. Chapter 4 explores their power in greater depth. Chapter 5 considers the results.

1.2. Background

In this section we place the dissertation in the context of artificial intelligence, particularly knowledge representation, and justify our focus on mathematics and graph theory as objects of study.

Artificial intelligence simulates intelligent behavior on a computer. We may hypothesize intelligence as movement through a search space. A point in the search space is the currently known set of objects and relations among them. The operators in the search space define the moves available from one point to the next. The rules for selecting and applying these operators guide the search. The search is the dynamic, procedural aspect of intelligence. Essential to such search is the ability to use symbolic representation.

The symbolic structures we create and manipulate are for large amounts of diverse knowledge [Newell 76]. Each time we manipulate knowledge with a

computer, we need a way to represent the objects involved. These objects may be chess pieces, cannibals or viruses. A fundamental problem in the choice of a representation is its *grain* or fineness of detail. The omission of some significant detail about an object may hamper our ability to reason about it. A complete description, however, (including, perhaps, molecular structure and historical background) would probably be overwhelming, impossible and irrelevant to the task at hand. The number and identity of the details chosen for a representation typically depend upon the intended task.

Symbolic representations are required for not only the objects, but also the relations among them, and the operators and rules of the search space. The ability to extract and apply data about details with heuristic significance is an important feature of intelligence [Minsky 63, Newell 75]. These difficult problems in symbolic representation have limited artificial intelligence exploration to toy domains: mathematical puzzles, games and theoretical tasks far from the real, physical, human world. The objects have been finite in number and the features chosen as salient have been relatively obvious. Even such simple domains have presented rich and challenging questions. The search, and a representation for the search space, have been the major challenges. Research is now meeting these challenges. Many game-playing programs have begun to surpass their human opponents. Theoretical results can often predict or bound the complexity of the search in question. New challenges are required.

Between the well-plumbed toy domains and the ill-understood physical world, lies another rich problem area: mathematics. Thousands of years of human thought have structured a complex web of objects and relations beyond the scope of any game designer. Yet, because people discovered mathematics and require no instrument (beyond a well-tuned mind) to study it, mathematics should be a fertile domain for research in artificial intelligence. Logic and theorem proving were among the earliest targets of artificial intelligence study and remain as active areas of exploration. Initial efforts in calculus [Slagle 63, Moses 75] were challenges in formula manipulation. Only recently has the focus for mathematical domains shifted

to the detection of (presumably existent) underlying patterns. Particularly noteworthy and relevant to this dissertation are the work of Mitchell, Lenat and Michener. Mitchell's LEX [Mitchell 83] formulates heuristics for the application of integration formulae in calculus, an example of a machine learning "unwritten" rules. Lenat's AM [Lenat 76] modelled mathematical research in set theory, making observations and conjectures on its discoveries. Michener [Michener 78] designed a set of three spaces (examples, results and concepts) to model a mathematician's understanding of mathematics. Mathematics is not a new domain for artificial intelligence, but an under-explored one.

To make the task more manageable, in this dissertation we restrict our focus to a single area of mathematics, graph theory. Graphs are an excellent subject of study because:

- There are representations for graphs which display much of the graphs' structure.
- There are representations for graphs which readily display a graph's relation to other graphs.
- Graphs are powerful representational data structures for many computer science problems, encoding semantic information syntactically.

In contrast, whether or not a group is cyclic is not immediately discernible from its operation table, nor would homomorphism between two groups be readily apparent. The maximum degree of a vertex in a graph, or whether two graphs share a particular vertex, is transparent in an adjacency list representation.

Another reason to study graphs, particularly digraphs, is their importance in artificial intelligence. Any problem search space is traditionally thought of as a graph [Nilsson 80]. Graphs have been used to represent real-world knowledge [Fahlman 77, Quillian 67], meaning in natural language [Sowa 79], hierarchical structures and planning [Sridharan 80], axioms and default rules for default reasoning [Sridharan 81], abstraction hierarchies for reasoning by analogy [Winston 80], psychological models of memory [Anderson 73, Winston 80, Schank 75], and concept descriptions [Winston 75]. Graphs have also been used to develop

resolution plans for theorem proving [Chang 79] and signal understanding [Feigenbaum 77]. In the social, biological and environmental sciences, graphs have proven constructive for such diverse problems as genetic substructure, archaeological seriation, trait development in child psychology, traffic flow management, food webs, garbage collection, electrical energy demand, health care delivery, phosphorus in a pasture ecosystem, and mathematical models of learning [Roberts 76].

In computer science, graphs are both the classic representation of a search space [Nilsson 80] and, increasingly, the symbolic structure for objects of study [Roberts 76]. Computer scientists have explored the explicit representation and search of graphs. Now we have a vast body of (ill-organized) mathematical knowledge ("graph theory") and some well-entrenched data structures (matrices, lists) for representing graphs on a computer. If we are to explore this material on a computer, a point in the search space will be our knowledge of graph theory at that instant, the rules will focus our attention and the operators will construct "discoveries." Such a search is open-ended. We require a clearer definition of our goal and some criteria to evaluate our performance.

1.3. Graph Theory and Its Representation

This section defines graph theory and posits criteria for its evaluation.

The evolving body of mathematical knowledge known as graph theory includes definitions, examples, theorems, algorithms, conjectures and proofs. We adopt the definition of graph theory as formulated by experts, represented in three general texts. One [Ore 62] is a classical development in elegant mathematical fashion. The second [Harary 72] encompasses a broader range of topics, presented as definitions and theorems. The third [Bondy 76] takes an algorithmic approach. Together, these texts are our benchmark; their contents are assumed to be *graph theory* and their contents "of interest" to graph theorists. We observe from them that typical theorems in graph theory describe the relations among graph properties.

For example:

- If a graph has property p and property q , then it has property r .
- A graph has property p if and only if it has property q .
- It is not possible for a graph to have both property p and property q .

Once we delineate graph theory, we need a formal representation for it. How do we evaluate such a representation? We identify two criteria:

- expressive power
- procedural power

A representation's *expressive power* is measured by its ability to describe correctly properties and objects of interest in graph theory. "Connected," "complete" and "acyclic" are examples of properties. Specific graphs are examples of objects. A representation with expressive power can be used to emphasize significant features and deliberately obliterate irrelevant ones. We gauge the expressive power of a representation against the texts we have chosen as a benchmark. It is somewhat more difficult to gauge a representation's procedural power. Imagine a mathematician doing research in graph theory. We have described in 1.1 the judgmental and intuitive nature of such work. We will therefore not concern ourselves with reproducing or quantifying the methods of the mathematician, but only with simulating the behaviors of the mathematician. Examples of such behavior are formulating a conjecture, testing a graph for a property and proving a theorem. All these behaviors are intended to add to the body of mathematical knowledge. We gauge *procedural power* by the number of such behaviors supported by the representation and the adequacy of their performance.

1.4. Languages for Graph Theory

This section outlines our general approach to graph theory representation based upon formal languages.

Consider the following fundamental aspects of graph theory:

- An object in graph theory is a finite graph, whose segments may be viewed as details in its description. A particular graph can be significant in graph theory as an example or a counterexample. Thus an object in graph theory is a set containing a single graph. There are infinitely many such objects.
- By extension, a graph property is a set of graphs. For many properties of interest in graph theory, a graph property is an infinite set.
- Theorems in graph theory, as we observed in 1.3, are essentially about graph properties. Thus important mathematical research behaviors (such as conjecture and proof) can be expressed with respect to sets of graphs.
- Mathematicians and scientists build graphs and then manipulate them with algorithms. Many algorithms are only applicable to graphs with specific properties. To reason about the applicability of an algorithm we must be able to describe a set of graphs.

The first two arguments address expressive power. The second two are relevant to procedural power. We conclude, then, that a good knowledge representation for graph theory will focus upon both finite and infinite sets of finite graphs. The explicit listing of all graphs with property p , each as a list of vertices and edges, would be impossible for an infinite set and inefficient in most finite cases. We require an alternative, a language in which to represent graph theory.

A grammar is an accepted way to represent a language. If we describe a language formally, we can explore its expressive power and construct from it a hierarchy of languages of increasing expressive power. In this dissertation, each of our representations is a grammar whose terminal strings may be interpreted as graph properties (sets of graphs). Our study proceeds from the simple to the complex. The edge-set languages of Chapter 2 are highly restricted. An edge-set language property describes graphs in terms of their edge sets and operations interpreted on those edge sets. These languages can express only finitely many graph properties. The need to represent infinite sets and expert observation that

pattern recognition is essentially recursive in nature [Poincare 70, Minsky 63], leads us to substantial exploration (in Chapters 3 and 4) of languages with a recursive procedural interpretation. These languages are not finite and have greater expressive power.

We distinguish two kinds of representational languages: declarative and procedural. A declarative language constructs descriptions in terms of objects, their existence and their properties. For example, "a graph has property p if *there exists some subgraph...*" or "*every graph* with property p has property q ." A declarative language is oriented to expressive power. A procedural language, on the other hand, is structured to simplify the specification of algorithms. In this dissertation we present languages which are simultaneously declarative and procedural. The languages of Chapter 2 express a well-structured finite set of graph properties with natural and efficient algorithms for them. The languages of Chapters 3 and 4 embody recursive algorithms within the formulation of each property.

Work in problem transformation has indicated that a relation between the representational and reasoning aspects of a search space can provide substantial problem solving power (See, for example, [Amarel 81].) By designing our languages to facilitate implementation, we should make coding easy and produce efficient algorithms. We deliberately impose upon our languages this procedural orientation.

1.5. Questions and Answers

We have now assembled enough background to state the key questions in this work, and point to the answers.

- To what extent can expressive and procedural power be incorporated into a single representation?

The edge-set languages of Chapter 2 have a surprisingly limited (finite) expressive ability but impressive procedural power. This highlights their potential for representing infinite sets, for hashing graphs and for

discovering commonalities and differences within a set of graphs. Unfortunately, a finite language's idea of an interesting property is unlikely to appear in a graph theory text. The recursively-formulated languages of Chapters 3 and 4 better meet the benchmark for expressive power. Much of graph theory seems representable in a consistent and unified fashion in this recursive formulation. Even better, the languages readily adapt to extensions (such as labelled graphs) and are amenable to procedural goals (such as inversion).

- How can we compare the expressive power of two representations?

We base comparisons of expressive power on hierarchies of formal grammars with identical semantic interpretations.

- How can we evaluate the complexity of a property?

We introduce, in the context of recursive languages, the floor of a property. Informally, this is the least powerful language in which the property is expressible.

This dissertation is not about algorithmic complexity, although it effectively isolates it within each representation. Nor is it about formal languages, although they support its exploration. Rather, this dissertation is about compact and elegant knowledge representation which draws its power from its descriptive focus and its dependency upon recursion.

1.6. Some Fundamental Definitions

This section formulates the most basic definitions we use in our work. We begin with the familiar classical definitions from graph theory. Next we construct our own definitions for a graph property, a graph characteristic and a graph description. Finally we explain how this terminology effects our formal language formulation. Additional, more special definitions will be introduced as needed, throughout the dissertation.

1.8.1. Basic Graph Terminology

We begin with the basic definitions for graph theory:

- Let V be a finite set of elements called *nodes* or *vertices*.
- Let I be the Cartesian product $V \times V$, $I = \{(x,y) \mid x,y \in V\}$.
- If E is any subset of I , the ordered pair $\langle V,E \rangle$ is used as the standard representation of a *graph*.
- If $(x,y) \in E$ such that x and y are distinct, (x,y) is called an *edge* and is abbreviated as xy . If $x,y \in V$ and $xy \in E$, vertices x and y are said to be *adjacent*, and x is said to be a *neighbor* of y .
- A graph $G = \langle V,E \rangle$ is *undirected* when $xy \in E$ if and only if $yx \in E$. Otherwise the graph is *directed*.
- If $(x,x) \in E$, (x,x) is called a *loop* and is abbreviated as xx . Every element of I is either an edge or a loop and not both. The set of all possible loops on V is $l = \{(x,x) \mid x \in V\}$.
- The *cardinality* of a set S is the number of distinct elements it contains and is denoted by $|S|$. We define $|V| = n$ and $|E| = m$.
- If $n = 0$ then $m = 0$ and $G = \langle \emptyset, \emptyset \rangle$ is defined to be the *empty graph*.
- Two graphs $G_1 = \langle V_1, E_1 \rangle$ and $G_2 = \langle V_2, E_2 \rangle$ are *isomorphic* to each other if there exists a one-to-one mapping $\pi : V_1 \rightarrow V_2$ such that $xy \in E_1$ if and only if $\pi(x)\pi(y) \in E_2$. The mapping π is called an *isomorphism*.

By now the reader will be grateful to learn that Appendix I is a reference table of symbols and their definitions. There is also an index (immediately following the references) citing all definitions and algorithms in this document. Definitions labelled "thesis specific" are our own terminology. All others are drawn from the benchmark texts. b Algorithms appear in all capital letters. The ordered pair $\langle V,E \rangle$ is the standard representation of a graph G ; all subsequent semantics will be given in terms of this standard representation.

1.6.2. Graph Properties, Characteristics and Descriptions

Let U be the set of finite graphs closed under isomorphism. A *graph property* p is a function mapping U into some range S of values. A graph property is said to be *boolean* if $S = \{\text{true}, \text{false}\}$. (e.g., planarity, completeness); it is said to be *numeric* if S is the set of non-negative integers (e.g., chromatic number, circumference). Two graph properties p_1 and p_2 are *equal* if and only if $p_1(G) = p_2(G)$ for all $G \in U$.

Graph theory includes:

- the definition of graph properties
- theorems establishing necessary and sufficient conditions for these properties to assume particular values
- algorithms to calculate the value of a property on a specific graph

A *characteristic* of a graph is an ordered pair (p, s) where p is a graph property with range S and $s \in S$.

A *description* d is a set of such characteristics.

A description $d = \{(p_1, s_1), (p_2, s_2), \dots, (p_k, s_k)\}$ is *satisfied* by a graph G if and only if $p_i(G) = s_i$ for $i = 1, 2, \dots, k$. It is possible for a description to be *unsatisfiable* with respect to U (satisfied by no members of U , e.g., "self-complementary and $n = 3$ "), *unique* (satisfied by exactly one isomorphism class of members of U , e.g., "self-complementary and a cycle"), or *general* (satisfied by more than one isomorphism class of members of U , e.g., "cyclic").

Let D be a finite set of descriptions. If for every two descriptions $d_1, d_2 \in D$ and graph $G \in U$, either G does not satisfy d_1 or G does not satisfy d_2 , then D is said to be *mutually exclusive*. If for every $G \in U$, there exists some $d \in D$ such that G satisfies d , D is said to be *collectively exhaustive*. A set D of descriptions which is mutually exclusive and collectively exhaustive *partitions* U into equivalence

classes, (for example, $D = \{(cyclic, true)\}, \{(cyclic, false)\}$). A major objective in Chapter 2 is to formulate languages whose semantic interpretations are graph characteristics from which a D may be created whose partition of U is describable by a concise syntactic form or *signature* for each class.

1.6.3. Graph Terminology and Graph Grammars

This section explains the interaction between a formal language representation and graph theory.

In Chapters 2, 3 and 4 we define a set of graph grammars. Each grammar will generate a language L of *L-expressions*. (For example, " $m = 3$ " might be a terminal string in language L .) The semantic interpretation of each L -expression will be a graph characteristic; we call such a characteristic an *L-characteristic* of a graph. (Continuing the example, the semantic interpretation of " $m = 3$ " might be "the number of edges in the graph is 3") Two L -expressions in a language L are *equivalent* if and only if their semantic interpretations are the same. (For example, in language L , " $m = 3$ " and " $2 < \text{integer } m < 4$ " might be shown to be equivalent.) Equivalence of semantic interpretation defines a set of equivalence classes on L -expressions. (Thus " $m = 3$ " and " $2 < \text{integer } m < 4$ " will both be in the same equivalence class of L -expressions.) An equivalence class of L -expressions designates a subset of U , namely, the set of all those graphs in U satisfying that L -characteristic. (Among others, $G_1 = \langle \{1,2,3\}, \{12,13,23\} \rangle$ and $G_2 = \langle \{1,2,3,4,5\}, \{12,34,25\} \rangle$ satisfy " $m = 3$ " and are in the same subset of U .) The number of distinct equivalence classes of L -expressions is exactly the number of distinct L -characteristics. Another aspect of the equivalence of L -expressions is that there may exist a finite set T of equations on L -expressions such that A and B are equivalent L -expressions if and only if one is derivable from the other using T as a replacement system. Members of T for edge-set languages are displayed in Chapter 2, and consideration of a way to demonstrate such equivalence appears in the discussion of subsumption in Chapter 4.

If a set P of L -characteristics designates a partition of U such that all the subsets are non-empty, P is said to be an L -property. By restricting our definition to non-empty subsets, we require that L -characteristics be satisfied by some graph in U . (Continuing the example, {the number of edges in the graph is 0, the number of edges in the graph is 1, the number of edges in the graph is 2, ...} would partition the set of all finite graphs.) The sets formed by an L -property partition the set U into equivalence classes. The language L allows us to name these equivalence classes and describe the one containing any given graph in U . (In our example, the classes could be named 0,1,2,..., and any finite graph would belong to that class whose name was equal to the number of edges in the graph.) Thus if there are precisely k distinct L -properties, the L -characterization of a graph G is the L -description of length k which G satisfies; this is as much as L can say about a given graph. An L -characterization is the most detailed description possible within L . The set of all satisfiable L -characterizations partitions the set U into equivalence classes; we call each such class an L -class. Any element in a class can serve as a representative or *signature* for its corresponding L -class of graphs.

As we postulate and explore languages for graph theory representation, we will focus on the preceding definitions. In particular:

- An L -characterization may or may not be satisfiable. The number of satisfiable L -characterizations is a way to measure the expressive power of the language.
- An L -characterization may or may not be unique. The number of unique characterizations is a way to measure the expressive power of the language. When an L -characterization is general, some graphs will be indistinguishable from each other.
- There may be finitely many or infinitely many L -classes of relatively equal or unequal cardinality. The evenness with which U is distributed among the L -classes is another way to measure the expressive power of the language.

A representation's procedural power will be measured by its ability to generate

examples, to test objects for properties, to construct algorithms, to hypothesize, to prove theorems, and to perform any other research behaviors observable when a mathematician thinks about graph theory. Methodology (e.g., focusing, intuition) is not part of procedural power.

We have now laid the framework for describing classes of graphs and modelling graph theory in formal languages. We begin with the edge-set languages.

CHAPTER 2

EDGE-SET GRAPH LANGUAGES

All mathematicians ... would be of nimble discernment if they had good sight, for they do not argue falsely upon principles familiar to them; and discerning minds would be mathematical if they could turn their eye towards the unfamiliar principles of mathematics.

--Pascal

This chapter develops the first of two major, interrelated families of languages for graph theory. The first section is an overview of the seven edge-set languages and their salient features. The next four sections describe edge-set languages in detail. The last section is an assessment.

2.1. General Overview

A family of languages is a collection of languages whose grammars and semantic interpretations are hierarchical and mutually consistent. A family of languages is always based on a (not necessarily explicitly defined) bounding language. The bounding language contains the underlying set of symbols, terms and expressions in the family. A hierarchy is defined by gradually including more symbols, terms and expressions in each new language, without eliminating those in the preceding language. The evolution of this hierarchy is motivated by a desire to increase the expressive power of a language.

In extending a language to formulate more expressive languages in the hierarchy, we postulate the following principles:

- Each language extension should be a refinement of that which precedes it, preventing loss of expressive capability.

- Additional expressive capability should partition many previously-existing classes, rather than a few, and particularly the largest previously-existing classes.
- Finiteness in the number of classes is a property to be preserved as long as possible.

This section describes, briefly and informally, the seven edge-set languages, which become a cornerstone of the recursive languages in Chapters 3 and 4. The terms in the grammars are always edge sets. Traditional set theoretic relations between edge sets are L-characteristics. These relations are selected so that properties are frequently boolean. The number and nature of the L-classes formed under the partition of L-characterizations is always finite for fixed n .

We would expect these edge-set languages to have a broad expressive ability. The simplest edge-set language is L_1 . L_2 , L_3 , L_{1n} , L_{2n} and L_{3n} are all extensions of L_1 and closely interrelated. The inability of these six to express certain graph properties suggests a somewhat different extension of L_1 to L_1^* , the seventh edge-set language. More extensive details are available in the remainder of this chapter.

2.1.1. Language L_1 Summary

In 1.6.3 we said that the expressions in a formal language have semantic interpretations which are graph characteristics. Language L_1 begins with primitive symbols whose semantic interpretations are edge-sets. The initial edge sets are I , E , l , and 0 . I is the Cartesian product $V \times V$.

$$I = \{xy \mid x, y \in V\}$$

E is any subset of I , l is the set of all loops on V ,

$$l = \{xx \mid x \in V\}$$

and 0 is the empty set $\emptyset$. We introduce two unary operators on any edge set S : reversal of the direction of all edges (denoted by S') and complementation with respect to I (denoted by $\bar{S}$). We define the binary operations of union and

intersection on edge sets. Finally we allow an equality relation (denoted as $=$) or an inequality relation (denoted as $\neq$) between any two edge sets constructed with the operators from the original four. Some sample L_1 -expressions follow:

$$E \cap \underline{1} \neq \phi$$

$$E \cup E' = E \cap E'$$

$$I = E \cup E' \cup \underline{1}$$

Our interpretation of these expressions will be in terms of the standard representation of the graph G by $\langle V, E \rangle$. Thus the first expression is interpreted as "the intersection of E and $\underline{1}$ is not the empty set" or "the elements of E include some non-loops" or " G includes at least one edge." Similarly, the second is interpreted as "if an edge is in E so is its reverse," and the third as "every edge or its reverse is in E ." Many L_1 -expressions, however, are equivalent (via this interpretation) to others. For example,

$$E \cap \underline{1} = \phi$$

is equivalent to

$$(E \cap \underline{1})' = (\phi)'$$

in its semantic interpretation.

Although the language contains infinitely many L_1 -expressions, they have only finitely many distinct interpretations. L_1 , being interpreted as only a finite number, say p , of distinct L_1 -properties, can be used to produce a description of length p for each graph. Because the L_1 -properties are all boolean and appear in complementary pairs, an L_1 -characterization can be efficiently represented as a binary vector of length p . There are 2^p such L_1 -characterizations and thus at most 2^p L_1 -classes for all finite graphs. L_1 identifies the L_1 -characteristics shared by two graphs as their matching vector entries. Because there are only finitely many L_1 -classes, a given description is not likely to describe only a single graph, although it may be taken as a canonical form for an equivalence class of graphs whose L_1 -properties are interpreted from L_1 -expressions. Further details on this language are provided in 2.2.

2.1.2. Language L_2 Summary

Language L_2 is an extension of L_1 which permits the relations of cardinal equality (denoted as $\sim$) and cardinal inequality (denoted as $\neq$) between two edge sets.

Because an interpretation of an L_2 -expression does not use integers specifically in its property statements, L_2 manages to remain finite in the number of distinct properties which can be interpreted from it, although it provides a superset of the descriptions available in L_1 . Characteristics which can be interpreted from L_2 -expressions but not from L_1 -expressions include:

$$(\bar{E})' \sim E$$

$$1 \cup E \neq \bar{E}$$

Based again on the standard representation $G = \langle V, E \rangle$, the first expression is interpreted as "the reversal of the complement of the edge set has as many edges as the edge set does"; the second as "there are not the same number of edges in the complement of the edge set as there are in the edge set and all the loops."

The statements pertaining to descriptions and deterministic algorithms in L_1 are equally applicable to L_2 . Further details on this language are provided in 2.3.

2.1.3. Language L_3 Summary

Language L_3 is an extension of L_2 which permits the relation of lesser cardinality (denoted as $<$) instead of cardinal inequality ($\neq$). The interpretation of an L_3 -expression does not use integers specifically in its property statements, and L_3 also remains finite in the number of distinct properties which can be interpreted from it. L_3 provides a superset of the properties available L_2 . Characteristics which can be interpreted from L_3 -expressions but not from L_2 -expressions include:

$$(\bar{E})' < E$$

$$E < 1 \cup \bar{E}$$

Based again on the standard representations $G = \langle V, E \rangle$, the first expression is interpreted as the "the reversal of the complement of the edge set has fewer

edges than the edge set"; the second as "there are fewer edges than the loops in the graph plus the edges in the complement."

The statements pertaining to descriptions and deterministic algorithms in L_2 are equally applicable to L_3 . Further details on this language are provided in 2.4.

2.1.4. Summary of Languages L_{1n} , L_{2n} and L_{3n}

Recall that n denotes the number of vertices in the graph. We will extend language L_i , for $i = 1, 2, 3$, to language L_{in} by permitting as expressions:

$$n = 1$$

$$n = 2$$

$$n = 3$$

Each of the finitely many equivalence classes in L_i is thus split into finitely or infinitely many equivalence classes in L_{in} . Language L_{in} provides a superset of the properties available in language L_i . Further results on language L_{in} are provided with the details on language L_i . (See 2.2, 2.3 and 2.4.)

2.1.5. Language L_1^* Summary

Language L_1^* was motivated by the inability of the six earlier edge-set languages to express most properties commonly appearing in graph theory texts, and does enhance the expressive power of L_1 to a limited extent. Language L_1^* is an extension of L_1 which includes the symbol E^* , the transitive closure of E . E^* has a recursive definition:

$$xy \in E^* \text{ if } xy \in E \text{ or if } xp, py \in E^*$$

Thus xy is in E^* for $G = \langle V, E \rangle$ if and only if there is an alternating sequence (a path) $x, xv_1, v_1, v_1v_2, \dots, v_k, v_ky$ of distinct vertices in V and edges in E beginning with x and ending with y . We chose to introduce the notion of transitive closure as a single symbol, rather than a unary operator on a term, in order to control the

combinatorics. Some sample L_1^* -expressions follow:

$$E^* \cap 1 = \phi$$

$$1 = E \cup E^* \cup 1$$

Using the standard representation $G = \langle V, E \rangle$, the first expression is interpreted as "no element of the transitive closure of the graph is a loop." The second is interpreted as "every edge is in the edge set or represents a path in the edge set." Any L_1^* -expression not including the symbol E^* is an L_1 -expression. Further results on this language are provided in 2.5.2.

2.2. Language L_1

This section describes, in detail, the theoretical nature of language L_1 and the empirical results achieved with it.

2.2.1. A Grammar for Language L_1

The formal grammar for L_1 on a graph $G = \langle V, E \rangle$ is
symbol: $E \mid 1 \mid 1 \mid 0$

term: $\text{symbol} \mid (\text{term})' \mid \underline{(\text{term})} \mid (\text{term} \cup \text{term}) \mid (\text{term} \cap \text{term})$

expression: $\text{term} = \text{term} \mid \text{term} \neq \text{term}$

For all the grammars in this family, we accept the convention of avoiding parentheses whenever a construction would be unambiguous without them.

Although the grammar clearly generates infinitely many L_1 -expressions (for example, $E = 0$, $(E)' = 0$, $((E)')' = 0$, ...), the semantic interpretations we give these L_1 -expressions place them in only finitely many equivalence classes. We interpret E as the edge set of the graph. We interpret 1 as the Cartesian product $V \times V$ for the vertex set V of the graph:

$$1 = \{xy \mid x, y \in V\}$$

We interpret 1 as the set of all loops on V ,

$$1 = \{xx \mid x \in V\}$$

and 0 as the empty set ϕ .

We interpret the construction $\text{term} = \text{term}$ as the binary relation of *set equality* defined on edge sets in the customary fashion. For edge sets S_1 and S_2 , $S_1 = S_2$ if and only if for every $xy \in S_1$, $xy \in S_2$ and for every $xy \in S_2$, $xy \in S_1$. Similarly, the construction $\text{term} \neq \text{term}$ is interpreted as *set inequality*. For edge sets S_1 and S_2 , $S_1 \neq S_2$ if and only if $S_1 = S_2$ is false. $E \cap I = 0$ is an expression in L_1 , interpreted as "none of the elements of E is an edge." Such a graph is $G_1 = \langle \{1,2,3\}, \{11\} \rangle$ or $G_2 = \langle \{1,2\}, \emptyset \rangle$. Another example of an expression in L_1 is $E \cup E' \neq E \cap E'$, interpreted as "there is a difference between the set of edges whose reverses are in E and the set of edges in E and its reverse." Such a graph is $G_1 = \langle \{1,2,3\}, \{12\} \rangle$ or $G_2 = \langle \{1,2,3\}, \{12,21,23\} \rangle$.

We interpret the construction $(\text{term})'$ as the unary operator *reversal*, which interchanges the order of the vertices in each element of an edge set, i.e., if S is an edge set

$$S' = \{yx \mid xy \in S\}$$

We interpret the construction $\underline{(\text{term})}$ as the unary operator *complement*, which replaces an edge set by its complement with respect to the universal set I . Thus for any edge set S

$$\underline{S} = \{xy \mid xy \notin S, xy \in I\}$$

Now we begin to assemble a set T of valid transformations on L_1 -expressions. (A transformation is valid if it preserves the semantic interpretation of an expression.) T includes the following transformations, for any edge set S , by definition of reversal:

$$((S)')' \Leftrightarrow S$$

$$I' \Leftrightarrow I$$

$$1' \Leftrightarrow 1$$

$$0' \Leftrightarrow 0$$

where " $\Leftrightarrow$ " means that one expression may be replaced by the other without altering the semantic interpretation. Any odd number of successive applications of reversal is equivalent to one reversal, and any even number is equivalent to no

reversal at all.

T includes the transformation "the complement of the complement of S is S itself", so any odd number of successive applications of the complement is equivalent to one complementation, and any even number is equivalent to no complementation at all. T also includes:

$$1 \Leftrightarrow 0$$

$$0 \Leftrightarrow 1$$

$$(E)' \Leftrightarrow (\underline{E})$$

The validity of the last transformation is due to the fact that both $(E)'$ and $(\underline{E})$ are $\{xy \mid yx \in E\}$.

We interpret the constructions (term $\cup$ term) and (term $\cap$ term) as the binary operations of *union* and *intersection*, in the traditional set operations. For edge sets S_1 and S_2 .

$$S_1 \cup S_2 = \{xy \mid xy \in S_1 \text{ or } xy \in S_2\}$$

$$S_1 \cap S_2 = \{xy \mid xy \in S_1 \text{ and } xy \in S_2\}$$

T includes the following transformations from set theory:

$$(\underline{A} \cap \underline{B}) \Leftrightarrow \underline{A \cup B}$$

$$(\underline{A} \cup \underline{B}) \Leftrightarrow \underline{A \cap B}$$

and

$$(A \cap B)' \Leftrightarrow A' \cap B'$$

$$(A \cup B)' \Leftrightarrow A' \cup B'$$

These, along with our earlier observations about reversal and complementation, make it possible to restrict both those unary operators to symbols rather than terms, without loss of expressive ability. That is, under this restriction, the same equivalence classes will be formed. Thus the following grammar will have the same L_1 -properties, although its expressions are a subset of the first grammar's.

symbol: $E \mid 1 \mid 0 \mid E' \mid \underline{E} \mid \underline{\underline{E}} \mid 1$

term: $\text{symbol} \mid (\text{term} \cup \text{term}) \mid (\text{term} \cap \text{term})$

expression: $\text{term} = \text{term} \mid \text{term} * \text{term}$

This is the grammar we will use for L_1 .

We are now interested in simplifying strings such as

$$((S_1 \cap S_2) \cup \dots) \cap (\dots)$$

From set theory we have, for any edge sets S_1, S_2, S_3 :

$$(S_1 \cup S_2) \cap S_3 \iff (S_1 \cap S_3) \cup (S_2 \cap S_3)$$

$$(S_1 \cap S_2) \cup S_3 \iff (S_1 \cup S_3) \cap (S_2 \cup S_3)$$

It remains, then, only to simplify such pairings from among the eight symbols. For any set S :

$$I \cup S \iff I$$

$$I \cap S \iff S$$

$$0 \cup S \iff S$$

$$0 \cap S \iff 0$$

Thus we need only consider expressions of the form $S_1 \cup S_2$ and $S_1 \cap S_2$, where S_1 and S_2 are chosen from among $E, \bar{E}, 1, \bar{1}, E', \bar{E}'$. We also know that T includes:

$$S \cup \bar{S} \iff 1$$

$$S \cap \bar{S} \iff 0$$

for any set S . What distinct sets of graphs will L_1 describe? Fortunately, set theory provides us with a classical problem transformation: the Venn diagram. For all possible unions and intersections of sets $S_1, S_2, \dots, S_k$ and their complements with respect to a superset I , the Venn diagram draws k intersecting circles in a rectangle. Every L_1 term corresponds to exactly one of the finitely many regions in the diagram. Thus L_1 has only finitely many terms and, therefore, finitely many expressions. How many such expressions are there? We must consider undirected and directed graphs separately. Since undirected graphs are combinatorially simpler, we look at them first.

2.2.2. L_1 for Undirected Graphs

For an undirected graph, the symbols E and E' refer to the same edge set, as do $\underline{E}$ and $\underline{E}'$. A Venn diagram for an undirected graph appears in Figure 2-1.

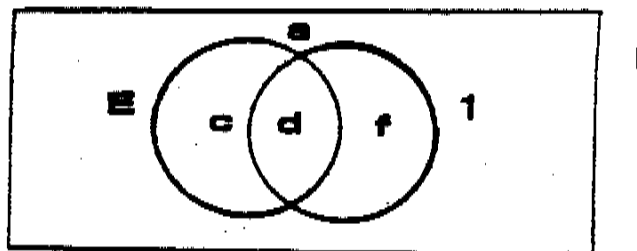


Figure 2-1: A Venn Diagram for Undirected Graphs

The symbol I is, by definition, the union of the others

$$I = E \cup 1 \cup 0 \cup \underline{E} \cup \underline{1}$$

and the symbol 0 , as the empty set, requires no explicit description on a Venn diagram. Complements with respect to I have a natural representation in a Venn diagram. Thus Figure 2-1 is justified in explicitly labelling only the symbols I , E and 1 . Observe that I is partitioned into four subsets, which we call *regions*. We have labelled these regions with a shorthand to be used throughout this chapter:

a denotes $\underline{E} \cap \underline{1}$, the non-loop edges not in the graph

c denotes $E \cap \underline{1}$, the non-loop edges in the graph

d denotes $E \cap 1$, the loops in the graph

f denotes $\underline{E} \cap 1$, the loops not in the graph

The interpretation of any term in L_1 for undirected graphs is either the empty set or the union of some of these four regions. Any L_1 -expression is interpreted as a statement of set equality or inequality between two such terms. There are only $2^4 = 16$ distinct interpretations of L_1 -terms. Since these relations are symmetric and complementary, there are at most $2\binom{16}{2} = 240$ L_1 -characteristics. Because these characteristics represent the truth or falsity of a boolean relation (set equality), there

are at most $240/2 = 120$ L_1 -properties and at most 2^{120} different L_1 -characterizations.

L_1 -properties, however, are far more manageably finite than that. We need only test for set equality, because all the properties are boolean. Since the regions a, c, d, and f partition the space, the correct interpretation of a statement of equality between L_1 -terms is really a list of empty regions, those appearing on only one side of the equal sign. For example, the L_1 -expression $\underline{1} = \underline{1} \cup \underline{e}$ is viewed in the Venn diagram representation as $a \cup c = a \cup d \cup f$. Because we always have $a = a$, the non-trivial portion of this is $c = d \cup f$ but, since c, d and f are disjoint, the equivalent statement is that c, d and f are all empty, which we denote as simply cdf. (Note that this implies $|V| = n = 0$ and the only graph with this particular property will be the empty graph.) Thus there are really only 4 non-trivial distinct L_1 -properties:

- a is empty
- c is empty
- d is empty
- f is empty

The L_1 -characterization of a finite undirected graph therefore consists of four characteristics, one for each boolean property. There are $2^4 = 16$ such L_1 -characterizations. We denote each L_1 -characterization by the list of regions it declares to be empty. Four of these L_1 -characterizations (df, adf, cdf and acdf) are satisfied only by the empty graph $\langle \emptyset, \emptyset \rangle$, and are consolidated as acdf. The characterization ac is equivalent to saying $|V| = 1$. Since there are only two such graphs, one satisfying description acd and the other satisfying description acf, the L_1 -characterization ac is eliminated. The 12 remaining L_1 -characterizations partition the set of all finite graphs and may be regarded as signatures for their respective classes. In Table 2-1 the 12 L_1 -classes of undirected graphs are listed. The signature of a class is a canonical form given as a list of empty regions. In the table's interpretations "edge" continues to denote a non-loop and "some" denotes a

non-empty proper subset. Subsequent languages will have signature computations performed by machine; these were performed by hand.

Class	Signature	Interpretation
1	none	some edges and some loops
2	a	all possible edges and some loops
3	c	no edges and some loops
4	d	some edges and no loops
5	f	some edges and all possible loops
6	ad	all possible edges and no loops
7	af	all possible edges and all possible loops
8	cd	no edges or loops but at least two vertices
9	cf	no edges and all possible loops
10	acd	$V = \{1\}, E = \emptyset$
11	acf	$V = \{1\}, E = \{11\}$
12	acdf	$V = \emptyset, E = \emptyset$

Table 2-1: Equivalence Classes for Undirected Graphs in L_1

What appeared to be a rich language is really quite coarse. Three of these signatures, (acd, acf and acdf) are for unique characterizations. Four more (ad, af, cd and cf) would describe a unique graph, up to isomorphism, if accompanied in L_{1n} by a value for n . Specifically, all members of class 6 are complete graphs of the form $\langle V, 1 \rangle$; all members of class 7 are complete graphs with all their loops $\langle V, 1 \rangle$; all members of class 8 are of the form $\langle V, \emptyset \rangle$; and all members of class 9 are of the form $\langle V, 1 \rangle$. A potential of 2^{120} classes has been reduced to 12, of which 5 will hold the majority of the graphs. It is to the credit of L_1 , however, that its interpretation is able to describe three graphs without explicitly stating the elements of either V or E . L_{1n} is able to characterize the 6 finite undirected graphs uniquely for $n = 2$. For each fixed $n > 2$ and each of the first 9 classes, there is at least one graph.

2.2.3. L_1 for Directed Graphs

For a directed graph we return to the original seven symbols in the L_1 grammar: $E, I, 1, 0, E, \underline{E}, \underline{E}', 1$. Once again $\underline{E}, \underline{E}', 1, 0$ have inherent interpretations in the Venn diagram, leaving us with three sets ($E, 1$ and E') to explore in Figure 2-2.

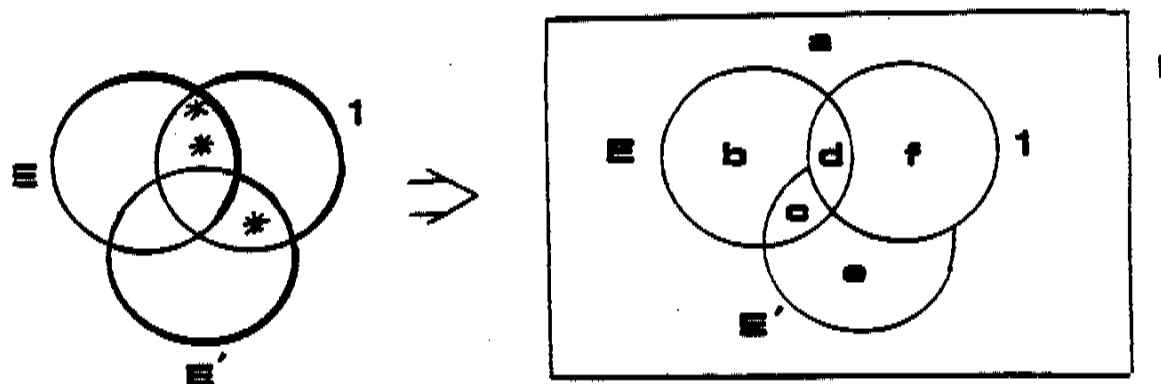


Figure 2-2: A Venn Diagram for Directed Graphs

This time I is partitioned into eight subsets. The following calculations, however, show that the two starred subsets of Figure 2-2 are always empty:

For the $*$ region: If $xy \in (E' \cap \underline{E} \cap 1)$ then $xy \in 1$ and $x = y$. If $xx \in E$ then $xx \in \underline{E}$ and $xx \in \underline{E}'$. Thus $(\underline{E} \cap E' \cap 1)$ is empty.

For the $**$ region: If $xy \in (E \cap \underline{E} \cap 1)$ then $xy \in 1$ and $x = y$. If $xx \in E$ then $xx \in E'$ and $xx \in \underline{E}$. Thus $(E \cap \underline{E} \cap 1)$ is empty.

Thus we are left with the six labelled regions in Figure 2-2. The labelling is interpreted as follows:

- a denotes $\underline{E} \cap \underline{E}' \cap 1$, the non-loop edges not in the graph whose reverses are not in the graph either
- b denotes $E \cap \underline{E}' \cap 1$, the non-loop edges in the graph whose reverses are not in the graph

- c denotes $E \cap E' \cap 1$, the non-loop edges in the graph whose reverses are in the graph
- d denotes $(E \cup E') \cap 1$, the loops in the graph
- e denotes $\bar{E} \cap E' \cap 1$, the non-loop edges not in the graph whose reverses are in the graph
- f denotes $(\bar{E} \cup \bar{E}') \cap 1$, the loops not in the graph

The interpretation of any term in L_1 for a directed graph is the union of some of these six regions. Any L_1 -expression is interpreted as a statement of set equality or inequality between two such terms. Although there are potentially $2^6 = 64$ interpretations of L_1 -terms, at most $2^{\binom{64}{2}} = 4032$ L_1 -characteristics and at most 2^{2016} L_1 -characterizations, the number of distinct L_1 -properties can be reduced using the same reasoning as in the undirected case. The empty graph this time has signature abcdef, subsuming classes which would have had the signatures df, adf, cdf, acdf, bdef, abdef and bcdef. The relationship between E and E' requires that $b \sim e$, so b is empty if and only if e is empty. The signature abce means $n = 1$ and is subsumed by abcde and abcef. This results in only 5 L_1 -properties and 24 equivalence classes for finite directed graphs based on L_1 -properties. The classes and their signatures are listed in Table 2-2 with an interpretation. Note that the undirected case is equivalent to both b and e being empty, which occurs in exactly 12 instances. A "one-way edge" denotes either xy in E or yx in E and not both. A graph $G = \langle V, E \rangle$ is said to be *weakly-complete* if and only if $xy \in E$ or $yx \in E$ for every distinct pair $x, y \in V$. In the Venn diagram representation, G is weakly-complete if and only if a is empty. In the table a "two-way edge" denotes both xy and yx in E . The calculations for the table were performed by hand.

Thus L_1 -properties may be used to partition all finite directed graphs into 24 equivalence classes. Again 3 signatures are for unique characterizations and 4 describe a unique graph, up to isomorphism, if accompanied in L_{1n} by a value for n . The remaining 17 might be a useful categorization technique for small undirected

Class	Signature	Weakly- Complete	One-way Edges	Two-way Edges	Loops
1	none	no	some	some	some
2	a	yes	some	some	some
3	c	no	some	none	some
4	d	no	some	some	none
5	f	no	some	some	all
6	ac	yes	some	none	some
7	ad	yes	some	some	none
8	af	yes	some	some	all
9	cd	no	some	none	none
10	cf	no	some	none	all
11	be	no	none	some	some
12	abe	yes	none	all	some
13	acd	yes	some	none	none
14	acf	yes	some	none	all
15	bce	no	none	none	some
16	bde	no	none	some	none
17	bef	no	none	some	all
18	abde	$E = 1$			
19	abef	$E = 1$			
20	bode	$E = 0$			
21	bcef	$E = 1$			
22	abcde	$V = \{1\}, E = \emptyset$			
23	abcef	$V = \{1\}, E = \{11\}$			
24	abcdef	$V = \emptyset, E = \emptyset$			

Table 2-2: Equivalence Classes for Directed Graphs in L_1

graphs. For $n > 2$ there is at least one graph in each of the first 21 classes. For

$n = 2$ there are 7 finite non-isomorphic directed graphs, 5 of which have distinct signatures in L_1 and two of which $\langle \{1,2\}, \{12,11\} \rangle$ and $\langle \{1,2\}, \{12,22\} \rangle$ are members of the same class (with signature ac).

2.2.4. An L_1 Graph Generator

A graph generator accepts an L -description and produces an arbitrary graph which satisfies that L -description. Since undirected graphs are a subset of directed graphs, in L_1 we can write a single graph generator for them both. Since there are only 5 L_1 -properties for directed graphs, any L_1 -description may be given as a five-place vector specifying whether a region is empty (1), not empty (0), or its contents are undefined (u), where the second region is b and e taken together. To use the generator we write a "front-end algorithm" which takes an L_1 -description as input (e.g., $\langle 1 \ 0 \ u \ u \ u \rangle$) and chooses, in a non-deterministic fashion, a more detailed version of the signature, replacing the u's with 0's or 1's, to create and output an L_1 -characterization.

FRONT-END

```

Dimension S(5)
For k = 1 to 5
  Read S(k)
  If S(k) = u then S(k) <- 0 or S(k) <- 1
Next k
Print S
```

Now we can write a generator which accepts an L_1 -characterization from FRONT-END and creates a graph which satisfies it. L_1 -GENERATOR labels the relationship between each unordered pair of vertices by the number of edges they will share. "Find" may be interpreted as "check to see that there exists." Only or-labels (e.g., "1 or 2") or u (undefined) labels may be changed. The algorithm embodies, in its case statement, knowledge of the minimal number of vertices possible in a graph satisfying each L_1 -characterization. The options permit all graphs to be accessible through the generator: the choice of the number of vertices ("do for a while"), the choice made when labelling, the elimination of or-labels at the end, and the final edge choice for a vertex pair labelled "1". The

algorithm's worst time complexity is quadratic in the internally generated (not input) value of n .

L1-GENERATOR

```

Read  $L_1$ -characterization  $S$ 
Case: /*minimal case knowledge*/
    abcdef in  $S$ : output  $\langle \phi, \phi \rangle$ , halt
    abc in  $S$ :  $V = \{1\}$ 
    ab or ac or bc in  $S$ :  $V = \{1, 2\}$ 
    else:  $V = \{1, 2, 3\}$ 

Do for a while
    add a vertex to  $V$ 

Create all loops and edges and label them  $u$ 
If  $a$  in  $S$ 
    then label all edge pairs "1 or 2"
    else find or label some edge pair "0"
If  $b$  in  $S$ 
    then label all edge pairs "0 or 1"
    else find or label some edge pair "2"
If  $ce$  in  $S$ 
    then label all edge pairs "0 or 2"
    else find or label some edge pair "1"
If  $d$  in  $S$ 
    then label all loops "F"
    else find or label some loop "T"
If  $f$  in  $S$ 
    then label all loops "T"
    else find or label some loop "F"

For each edge pair labelled  $u$ , relabel as "0" or "1" or "2"
For each edge pair labelled "1 or 2", relabel as "1" or "2"
For each edge pair labelled "0 or 1", relabel as "0" or "1"
For each edge pair labelled "0 or 2", relabel as "0" or "2"
For each edge pair  $xy$  labelled "1", place only one of  $xy$  or  $yx$  in  $E$ 

```

For each edge pair xy labelled "2", place xy and yx in E
 For each loop labelled u , relabel as "T" or "F"
 For each loop xx labelled "T", place xx in E
 Output $\langle V, E \rangle$

2.2.5. An L_1 Testing Algorithm

A testing algorithm for language L accepts an L -description and a graph, and returns "true" if the graph satisfies the L -description and "false" if it does not. Since an L_1 -description may be written as a 5-place vector, the testing algorithm L_1 -TESTER is as follows:

L_1 -TESTER

```

Read graph  $G = \langle V, E \rangle$ 
Read  $L_1$ -description  $S$ 
Create all loops and edges and label them  $u$ 
Do for each  $xy \in E$ ,  $x$  and  $y$  distinct
    Relabel edge  $xy$  "T"
Do for each  $xx \in E$ 
    Relabel loop  $xx$  "T"
If ( $\exists xy$ , distinct  $x$  and  $y$ , labelled  $u$  and  $yx$  labelled  $u$ ,  $a$  in  $S$ )
  or ( $\exists xy$ , distinct  $x$  and  $y$ , labelled  $u$  and  $yx$  labelled  $T$ ,  $b$  in  $S$ )
  or ( $\exists xy$ , distinct  $x$  and  $y$ , labelled  $T$  and  $yx$  labelled  $T$ ,  $c$  in  $S$ )
  or ( $\exists xx$  labelled  $u$  and  $f$  in  $S$ )
  or ( $\exists xx$  labelled  $T$  and  $d$  in  $S$ )
    then print "false"
    else print "true"
```

This testing algorithm is quadratic in n .

2.2.8. Transition from L_1 to L_2

We have demonstrated that L_1 is a very limited language, contrary to our expectations. For undirected graphs, L_1 is coarse in that it does not distinguish many disjoint subsets of U . L_1 fares somewhat better for directed graphs. The graph generator and testing algorithms for L_1 are quadratic in the number of vertices. A description composed of values for all the properties partitions the set of all finite graphs into 12 classes for undirected graphs and 24 classes for directed graphs. In three of these classes the signature is for a unique characterization. L_{1n} has 5 properties for undirected graphs, 6 for directed graphs, and creates infinitely many classes. There are four L_1 -properties which can become unique characterizations in L_{1n} .

With the principles of 2.1 in mind, we will proceed to language L_2 .

2.3. Language L_2

This section describes, in detail, the theoretical nature of language L_2 and the empirical results observed for it on the DEC-20.

2.3.1. A Grammar for Language L_2

The formal grammar for L_2 on a graph $G = \langle V, E \rangle$ is
 symbol: $E \mid 1 \mid 1 \mid 0$

term: $\text{symbol} \mid (\text{term})' \mid \underline{(\text{term})} \mid (\text{term} \cup \text{term}) \mid (\text{term} \cap \text{term})$

expression: $\text{term} = \text{term} \mid \text{term} \neq \text{term} \mid \text{term} \sim \text{term} \mid \text{term} \neq \text{term}$

We interpret the construction $\text{term} \sim \text{term}$ as the binary relation of *equal set cardinality*. For edge sets S_1 and S_2 , $S_1 \sim S_2$ if and only if $|S_1| = |S_2|$. Similarly, the construction $\text{term} \neq \text{term}$ is interpreted as *inequality of set cardinality*. For edge sets S_1 and S_2 , $S_1 \neq S_2$ if and only if $S_1 \sim S_2$ is false.

Since the grammar for L_2 differs from the grammar for L_1 only in its use of set cardinality, we may reformulate it, as we did the grammar for L_1 , without loss of expressive capability, to be:

symbol: $E \mid 1 \mid 1 \mid 0 \mid E \mid \underline{E} \mid \overline{E} \mid 1$

term: $\text{symbol} \mid (\text{term} \cup \text{term}) \mid (\text{term} \cap \text{term})$

expression: $\text{term} = \text{term} \mid \text{term} \neq \text{term} \mid \text{term} \sim \text{term} \mid \text{term} \neq \text{term}$

Once again we will consider undirected and directed graphs separately.

2.3.2. L_2 for Undirected Graphs

Every L_1 -property is an L_2 -property. With L_2 we can supplement the Venn diagram representation by stating that certain regions, or unions of regions, have the same number of elements. By reasoning similar to that for L_1 , we can show that there are at most $2^{\binom{16}{2}} = 240$ L_2 -characteristics which are interpretations of L_2 -expressions involving the relation $\sim$ or $\neq$. Many of these L_2 -characteristics, however, are equivalent to L_1 -characteristics. For example,

$$E \cup \underline{1} \sim (E \cap 1) \cup (\underline{E} \cap \underline{1})$$

is, in the Venn diagram,

$$|a \cup c \cup d| = |a \cup d|$$

or, more briefly,

$$|acd| = |ad|$$

Because regions are disjoint, this suggests an equation in integer unknowns:

$$|a| + |c| + |d| = |a| + |d|$$

which we will abbreviate as

$$a + c + d = a + d$$

This provides no more information about the nature of G than does $|c| = 0$, which is equivalent to the L_1 -expression c is empty. As in L_1 , all properties are boolean and thus we may restrict our attention to only $=$ and $\neq$. Since $d + f = n$ and $a + c = n(n-1)/2$, the property $d \sim acf$ implies $d = n(n-1)/2 + n - d$. Since d is no larger than n , we have $n < 4$. Since $ac \sim df$ implies $n = 3$, and it is not possible for $d \sim acf$ if $n = 1$ or $n = 2$, the property $d \sim acf$ is redundant and is excluded,

as is $f \sim acd$ in a similar proof. After these eliminations, only 27 L_2 -properties remain; they are listed in Table 2-3.

Number	Property	Number	Property
1	a	15	$af \sim c$
2	c	16	$af \sim d$
3	d	17	$cd \sim a$
4	f	18	$cd \sim f$
5	$a \sim c$	19	$cf \sim a$
6	$a \sim d$	20	$cf \sim d$
7	$a \sim f$	21	$df \sim a$
8	$c \sim d$	22	$df \sim c$
9	$c \sim f$	23	$ac \sim df$
10	$d \sim f$	24	$ad \sim cf$
11	$ac \sim d$	25	$af \sim cd$
12	$ac \sim f$	26	$a \sim cdf$
13	$ad \sim c$	27	$c \sim adf$
14	$ad \sim f$		

Table 2-3: Properties of Undirected Graphs in L_2

The calculations for Table 2-3 were performed by hand, but here the manual labor ends. Cardinal set inequality does not readily lend itself to an elegant proof of the number of distinct characterizations possible in L_2 . Thus we chose to create a FORTRAN program (called L2 and on view with its results in Appendix II) to explore exactly how many of those 2^{27} possible L_2 -characterizations ever occur. "Ever" is a long time in an infinite class, so we ran L2 until we despaired of ever finding a new signature. L2 examined every graph for which $n < 26$ and found only 106 distinct L_2 -characterizations. The last new one occurred at $n = 12$.

Manual computations indicate that among the 106 signatures for these classes, 9 are unique (in the sense defined in 1.6.2) descriptions and are listed in Table 2-4. All edges are undirected and only listed once. The first three of these were also available in L_1 .

Signature	V	E
acd	{1}	$\emptyset$
acf	{1}	{11}
acdf	$\emptyset$	$\emptyset$
$a, c \sim d \sim f$	{1,2}	{12,11}
$c, a \sim d \sim f$	{1,2}	{11}
$ad, c \sim f$	{1,2,3}	{12,13,23}
$af, c \sim d$	{1,2,3}	{12,13,23,11,22,33}
$cd, a \sim f$	{1,2,3}	$\emptyset$
$cf, a \sim d$	{1,2,3}	{11,22,33}

Table 2-4: Unique L_2 Characterizations for Undirected Graphs

An L_{2n} graph testing algorithm requires only the number of elements in each of a , d and n in order to generate the L_2 -characterization for a graph. We call such a value triple a case. The material in Table 2-5 is drawn from machine-generated computations. For a fixed number of vertices, the table compares the number of L_2 -characterizations which actually occurred for a given n to the number of possible cases. Since $d + f = n$ and $a + c = n(n-1)/2$, we have $(n+1)(1+n(n-1)/2)$ possible cases for a graph on n vertices. The signature which satisfies none of the 27 properties is by far the largest class for $n > 5$. The initially declining value of the percentage of cases in the largest class is attributable to the relatively few cases for $n < 5$. The class with signature "none" is increasingly more populated as n increases, especially for prime n , where none of the modulo-oriented restrictions apply.

Two finite graphs which have the same values for a , d and n will be indistinguishable from each other via their L_2 -characterizations and will lie in the

n	Cases	Characterizations	Largest Class Size	Largest Class Percent
0	1	1	1	100
1	2	2	1	50
2	6	6	1	17
3	16	12	2	13
4	35	33	2	6
5	66	28	8	12
6	112	42	24	21
7	176	29	48	27
8	261	50	76	29
9	370	34	196	53
10	506	36	272	54
11	672	35	400	60
12	871	58	512	59
13	1106	30	792	72
14	1380	36	960	70
15	1696	43	1268	75
16	2057	50	1460	71
17	2466	30	1984	80
18	2926	40	2276	78
19	3440	35	2808	82
20	4011	50	3136	78
21	4642	34	3964	85
22	5336	36	4400	82
23	6096	35	5236	86
24	6925	58	5732	83
25	7826	30	6912	88

Table 2-5: Results of Program L2 on Undirected Graphs

same L_2 -class. For $n = 2$, however, each a and d value pair defines a unique (up

to isomorphism) undirected graph and thus the L_2 -characterization is unique, i.e., no two non-isomorphic undirected graphs on 2 vertices have the same L_2 -characterization. For $n = 3$ the cases are spread among 12 classes, with never more than 2 in a class. For $n = 4$ the cases are spread among 33 classes, with never more than 2 in a class. For fixed $n > 7$, a minimum of 30 different L_2 -characterizations occur, but as n increases the grain of this partition coarsens. Forty-eight of the signatures turn out to be applicable to only a single value of n , and 16 more restrict n to values modulo some integer. These results are due to the fact that an L_2 -characterization is interpreted as a system of equations and inequalities in non-negative integer unknowns (a , c , d and f) which may be solved for n . The following are always a part of this system:

$$a + c = n(n-1)/2$$

$$d + f = n$$

$$0 \leq d \leq n$$

$$0 \leq f \leq n$$

$$0 \leq a \leq n(n-1)/2$$

$$0 \leq c \leq n(n-1)/2$$

For example, consider the L_2 -description $c = f$ and $cf = d$. This may be rewritten as:

$$c = f$$

$$c + f = d$$

or

$$2f = d$$

which, by substitution, yields

$$3f = n$$

so n is congruent to 0 modulo 3. This example is intended to demonstrate the strengths and weaknesses of L_2 .

2.3.3. L_2 for Directed Graphs

For directed graphs in L_2 the same six-region Venn diagram is applicable. Using the established reasoning pattern we make initial estimates of $2^6 = 64$ interpretations of L_2 terms, $4\binom{64}{2} = 8064$ L_2 -characteristics and at most 2^{4032} L_2 -characterizations. We have already shown, however, that $2\binom{64}{2}$ of these result in only 5 properties using $=$. The other $2\binom{64}{2}$ properties arising from $\sim$ are reducible, by manual calculations, to 197. The program L2DI (on view with its output in Appendix III) explored how many of these possible characterizations occur up to $n = 25$. (Limiting values for n are based upon space and time limitations.) The material in Table 2-6 is drawn from L2DI output. 4849 distinct signatures were found; 2572 of them for more than a single value of n . The last new signature appeared at $n = 25$. For $n = 1$ and $n = 2$, L_2 provides no finer a partition than L_1 . For $n > 2$ the partition is a substantial improvement over L_1 , of increasing refinement until $n = 9$. A minimum of 911 classes appear for $n > 7$.

2.3.4. Algorithms for Generating and Testing in L_2

The construction of an arbitrary graph satisfying an L_2 -description requires the solution of a system of linear inequalities in the non-negative integer variables b , c , d and n . The same six equations from 2.3.2 form the basis for this system. Each of the 27 boolean properties without a u value in the signature contributes another equation. For example, $af \sim d$ is interpreted as $a + f = d$.

The approach of L1-GENERATOR, where we reset u values to 0 or 1 would be inefficient here, because so many properties are incompatible with each other. Instead we search for the constraints on the variables first and then set their values arbitrarily. (There is, therefore, no consideration of a minimum case.) GENERATOR reads in the dimension of the signature and then the signature itself, constructing the relevant equations and inequalities. GENERATOR then calls a package (such as IBM's Mixed Integer Programming) [75] to solve the system of inequalities established. GENERATOR then selects arbitrary values for a , c , d , f and n consistent with the solution. The construction of a graph with these values is similar to that

n	Cases	Characterizations	Largest Class Size	Largest Class Percent
0	1	1	1	100
1	2	2	1	50
2	6	6	1	17
3	24	20	2	8
4	80	78	2	3
5	216	141	8	4
6	504	336	24	5
7	1056	484	48	5
8	2025	956	76	4
9	3610	911	196	5
10	6072	1065	416	7
11	9744	1045	1086	11
12	15028	1750	2496	17
13	22400	998	5746	26
14	32430	1098	9758	30
15	45792	1584	16156	35
16	63257	1785	23508	37
17	85698	968	40284	47
18	114114	1438	55838	49
19	149640	1104	77874	52
20	193536	1651	101792	53
21	247192	1255	150060	61
22	312156	1107	189316	61
23	390144	1081	250364	64
24	483025	2104	305916	63
25	592826	1108	413362	70

Table 2-6: Results of L2DI on Directed Graphs
for L1-GENERATOR. The algorithm follows:

GENERATOR

```

Read k
Dimension S(k)
a + b + c + e <- n(n-1)/2
d + f <- n
0 ≤ d ≤ n
0 ≤ f ≤ n
0 ≤ a ≤ n(n-1)/2
0 ≤ b ≤ n(n-1)/2
0 ≤ c ≤ n(n-1)/2
0 ≤ e ≤ n(n-1)/2
For j = 1 to k
Read S(j)
Write equation (inequality) based on S(j)
Next j
Call Mixed Integer Programming to solve system
Choose values for b, c, d and n consistent with the solution
e <- b
a <- n(n-1)/2 - b - c - e
f <- n - d
Call Heap's program to construct graph G
Append d loops to G
Output G

```

The last steps of GENERATOR call a package (perhaps Heap's program from the National Physical Laboratory at Middlesex) to construct a graph G with the b , c and n values specified, and then appends d loops before outputting the graph. Alternatively, the tuple $\langle a, b, c, d, e, f, n \rangle$ may be output. Although this form of the graph is one to which we are unaccustomed, it is really all L_2 is capable of saying about G .

The L_2 graph testing algorithm is a simplistic procedure. It reads in the graph and the L_2 -signature, and then confirms each of the properties flagged as true

(denoted by 1):

TESTER

Read k

Read graph $G = \langle V, E \rangle$

Calculate a, b, c, d, e, f and n

For $i = 1$ to k

Read $S(i)$

If $S(i) = 1$ and interpretation($S(i)$) is false

then print FALSE and halt

else continue

Next i

Print TRUE

The testing is quadratic in n.

23.5. A Comparison of L_1 and L_2

Clearly L_2 is an extension of L_1 and fits the criteria for extension suggested in 2.1. L_2 -characterizations subdivide each of the 9 non-unique classes of the partition of all finite graphs formed by L_1 -characterizations, as shown in Table 2-7. L_2 -characterizations offer further information on the values of a and d without explicitly stating them. There are still a finite number of L_2 -classes.

L_2 appears to extend L_1 in the desired fashion, concentrating much of its precision where L_1 was weakest. For undirected graphs with n less than 7 or 8, L_2 -characterization may be an adequate categorization.

L_2 is certainly an improvement on L_1 . It provides substantially more equivalence classes and refines the largest L_1 classes. The graph generator is based upon a problem transformation into a system of linear inequalities. Both the exploratory program and the graph tester find the numerical values of a, b, c, d, e, f and n an adequate description of G. A description composed of values for the L_2 -properties appears to partition the set of all finite graphs into 106 classes for undirected graphs and at least 4849 classes for directed graphs. L_{2n} has 28

L_1 Signature	Number of L_2 Subdivisions
none	70
a	4
c	4
d	10
f	10
ad	2
af	2
cd	2
cf	2

Table 2-7: Refinement of L_1 by L_2

properties for undirected graphs, no more than 202 properties for directed graphs, and creates infinitely many classes. In the spirit of 2.1 we will now expand our edge-set language hierarchy once again.

2.4. Language L_3

This section describes, in detail, the theoretical nature of language L_3 and the empirical results achieved with it.

2.4.1. A Grammar for Language L_3

The formal grammar for L_3 on a graph $G = \langle V, E \rangle$ is

symbol: $E \mid 1 \mid 1 \mid 0$

term: $\text{symbol} \mid (\text{term})' \mid \underline{(\text{term})} \mid (\text{term} \cup \text{term}) \mid (\text{term} \cap \text{term})$

expression: $\text{term} = \text{term} \mid \text{term} \neq \text{term} \mid \text{term} \sim \text{term} \mid \text{term} \neq \text{term} \mid \text{term} < \text{term}$

We interpret the construction $\text{term} < \text{term}$ as the binary relation of *lesser cardinality* between sets. For sets S_1 and S_2 , $S_1 < S_2$ if and only if $|S_1|$ is less than $|S_2|$. Since the grammar for L_3 differs from the grammar for L_2 only in its

introduction of lesser set cardinality (as denoted by $<$), we may reformulate it (with the T transformations as we did the grammars for L_1 and L_2) without loss of expressive capability, to be:

symbol: $E \mid 1 \mid 0 \mid E' \mid \bar{E} \mid \bar{E}' \mid 1$

term: $\text{symbol} \mid (\text{term} \cup \text{term}) \mid (\text{term} \cap \text{term})$

expression: $\text{term} = \text{term} \mid \text{term} \neq \text{term} \mid \text{term} \sim \text{term} \mid \text{term} \nabla \text{term} \mid$
 $\text{term} < \text{term}$

Language L_3 is an extension of L_2 which permits the relation of lesser cardinality between two sets. The interpretation of L_3 does not specifically use integers, and L_3 also partitions U into finitely many classes. Properties which can be interpreted from L_3 -expressions but not from L_2 -expressions include:

$$\bar{E} < E'$$

$$1 \cap E < \bar{E}$$

The first may be interpreted as "the complement of the edge set has fewer edges than the reversal of the edge set"; the second as "there are fewer loops in the graph than there are edges in the complement of the reversal of the edge set."

We will again consider undirected and directed graphs separately.

2.4.2. L_3 for Undirected Graphs

L_3 includes all L_1 -properties and L_2 -properties. In addition to the L_2 -expression $\text{term} \nabla \text{term}$, L_3 uses $\text{term} < \text{term}$. There are $2\binom{16}{2} = 240$ L_3 -characteristics which are interpretations of L_3 -expressions involving the asymmetric relation $<$. The L_3 -expressions $\text{term}_1 < \text{term}_2$ and $\text{term}_2 < \text{term}_1$ are refinements on the L_2 -expression $\text{term}_1 \neq \text{term}_2$. Some of these, such as $ac < a$, would be mathematically impossible. If we restrict $\text{term}_1 < \text{term}_2$ so that term_1 is not a proper subset of term_2 , we have a potential of only 175 L_3 -characteristics involving $<$.

The count of L_3 -characteristics is therefore 54 L_2 -characteristics plus 175

characteristics new to L_3 , for a total of 229, suggesting a potential of 2^{229} different L_3 -characterizations. We can reduce this estimate substantially by observing that many such characterizations would be mathematically unacceptable. L_3 still has a valid transformation as a system of equations and inequalities, but a set of L_3 -expressions such as

$$a < c$$

$$c < d$$

$$d < a$$

would be entirely unacceptable. We observe that the most complete and consistent set of statements L_3 could formulate about a graph would be an ordering of the distinct non-empty subsets available as unions of the four regions in Figure 2-1. (This also indicates that the introduction of the relations $>$, $\leq$ and $\geq$ would not increase the number of L -classes and should not be considered.) There are $2^4 - 1 = 15$ such subsets and in any such ordering we could force $acdf$ to be the last. There are therefore $14!$ permutations of the subsets. Between every pair of subsets in a permutation either $=$ or $<$ must appear, in order to construct an ordering. Our bound on the number of L_3 -characterizations has now improved to $2^{13} 14! < 2^{50}$.

We created a FORTRAN program (called L3 and on view with its results in Appendix IV) to explore how many of those $2^{13} 14!$ possible L_3 -characteristics ever occur. L3 examined every graph for which $n < 26$ and found only 259 distinct L_3 -characterizations. The last new one (as with L_2) occurred at $n = 12$. Of these, 157 were for more than a single value of n . The 102 signatures restricted to a single value of n occurred only for values of n less than 8.

The material in Table 2-8 is from L3 output. For a fixed value of n , the table compares the number of L_3 -characterizations which actually occurred for a given n to the number of possible cases. For fixed n , $7 < n < 26$, L_{3n} separates graphs into at least 108 equivalence classes, with no class containing more than 17% of the cases. For $n = 1, 2, 3$ and 4 , L_3 was able to uniquely characterize every case

n	Cases	Characterizations	Largest Class	
			Size	Percent
0	1	1	1	100
1	2	2	1	50
2	6	6	1	17
3	16	16	1	6
4	35	35	1	3
5	66	52	2	3
6	112	90	4	4
7	176	96	6	3
8	261	129	7	3
9	370	112	16	4
10	506	118	28	6
11	672	120	50	7
12	871	149	70	8
13	1106	108	114	10
14	1380	122	144	10
15	1696	128	203	12
16	2057	145	245	12
17	2466	108	336	14
18	2926	126	392	13
19	3440	120	504	15
20	4011	145	576	14
21	4642	112	730	16
22	5336	122	820	15
23	6096	120	1001	16
24	6925	153	1111	16
25	7826	108	1344	17

Table 2-8: Characterizations for Undirected Graphs in L_3
(not graph) submitted to it

2.4.3. L_3 for Directed Graphs

For directed graphs in L_3 the same six-region Venn diagram remains applicable. This time we have 2567 mathematically acceptable L_3 -characteristics. There are now $2^6 - 1 = 63$ subsets to permute, and a bound of $2^{62} 63! < 2^{72}$ possible L_3 -characterizations. The program L3DI (on view with its output in Appendix V) explored how many of these possible characterizations occur up to $n = 13$. (The limiting value of 13 was based upon space constraints.) The material in Table 2-9 is drawn from L3DI output.

n	Cases	Characterizations	Largest Class Size	Largest Class Percent
0	1	1	1	100
0	1	1	1	100
1	2	2	1	50
2	6	6	1	17
3	24	24	1	4
4	80	80	1	1
5	216	200	2	1
6	504	476	4	1
7	1056	876	6	1
8	2025	1670	9	0
9	3610	2734	16	0
10	6072	4080	28	0
11	9744	5848	50	1
12	15028	7809	73	0

Table 2-9: Results of Program L3DI on Directed Graphs

20,001 distinct signatures were found; 5191 of them for more than a single value of n . The last new signature occurred at $n = 13$. Because the program exhausted its space constraints and never completed $n = 13$, it is likely that there are far

more than 20,001 distinct signatures and that fewer than 5191 of them are restricted to a single value of n . For $n = 1$ and $n = 2$, L_3 provides no finer a partition than L_1 or L_2 . For $n > 2$ the partition is a substantial improvement over L_2 , of increasing refinement at least until $n = 13$. For $n > 3$, no class contains more than 1% of the cases.

2.4.4. Algorithms for Generating and Testing in L_3

The construction of an arbitrary graph satisfying an L_3 -description requires the solution of a system of linear equations and inequalities, just as it did for L_2 . For undirected graphs there are five integer variables (a, c, d, f, n), three of which are independent (a, d, n). For directed graphs there are seven integer variables (a, b, c, d, e, f, n), of which four (a, b, d, n) are independent. Each boolean L_3 -property contributes an inequality or an equation to the basic system of six. If we set the dimension of the signature to 258 for undirected graphs, or 2567 for directed graphs, the algorithm for generating graphs with L_3 -properties is identical to the GENERATOR in 2.3.4. The L_3 graph testing algorithm is also a reproduction of TESTER in 2.3.4.

2.4.5. A Comparison of L_3 with L_2

L_3 is definitely an improvement on L_2 . The problem transformation into an ordering of the subsets in the Venn diagram provides a much greater bound on the number of distinct signatures. For $n > 2$, the density is substantially reduced for both directed and undirected graphs. L_3 -characterizations offer further information on the values of a , b and d without explicitly stating them. There are still a finite number of L_3 -classes.

L_3 appears to extend L_2 in the desired fashion, concentrating much of its precision where L_2 was weak. For undirected graphs with n less than 16 or 17, L_3 -characterization may be an adequate characterization. For directed graphs with n less than 13, this is certainly true, and the value of n may even be higher.

L_3 provides a remarkable number of equivalence classes. It appears to partition the set of all finite graphs into 259 classes for undirected graphs and more than 20,000 for directed graphs. L_{3n} has at most 229 properties for undirected graphs, at most 2567 for directed graphs and creates infinitely many classes. In the spirit of 2.1 we will now expand our edge-set language hierarchy once again.

2.5. The Language L_1^*

This section describes, in detail, the theoretical nature of language L_1^* and makes some empirical observations on it.

2.5.1. A Grammar for Language L_1^*

The formal grammar for L_1^* on a graph $G = \langle V, E \rangle$ is

symbol	$E \mid E^* \mid 1 \mid 0$
term	symbol $\mid$ (term)' $\mid$ <u>(term)</u> $\mid$ (term $\cup$ term) $\mid$ (term $\cap$ term)
expression	term $=$ term $\mid$ term $*$ term

We interpret the symbol E^* as the *transitive closure* of the edge set

$$E^* = \{xy \mid xy \in E \text{ or } xp, py \in E^*\}$$

Note that we have not introduced transitive closure (*) as a unary operator on edge sets, but instead have introduced a new symbol (E^*), effectively limiting transitive closure to E alone. This introduction of a new edge set symbol makes analysis of the language more manageable. Language L_1^* is an extension of L_1 which permits consideration of paths existing in the graph. We may reformulate the grammar for L_1^* without loss of expressiveness so that the unary operators are restricted to symbols:

symbol	$E \mid E^* \mid 1 \mid 0 \mid \underline{E} \mid \underline{E}^* \mid \underline{1} \mid \underline{0} \mid E' \mid E^* \mid \underline{E} \mid \underline{E}^* \mid \underline{1} \mid \underline{0}$
term	symbol $\mid$ (term $\cup$ term) $\mid$ (term $\cap$ term)

expression: $\text{term} = \text{term} \mid \text{term} * \text{term}$

Properties which can be interpreted from L_1^* -expressions but not from L_1 -expressions include:

$$\underline{E}^* = 0$$

$$E^* \cap 1 = E$$

The first may be interpreted as "the complement of the transitive closure of the edge set is empty"; the second as "all the non-loops in the transitive closure of the edge set are in the edge set already." For L_1^* we will explore only the undirected graphs.

2.5.2. L_1^* for Undirected Graphs

If there is a path from x to y in the undirected graph $G = \langle V, E \rangle$, then there is also a path from y to x , i.e.,

$$(E^*)' = E^*$$

For undirected graphs we still have $S' = S$ for any edge set, and thus a Venn diagram need only represent the relationships among the seven symbols E , E^* , 1 , 1 , 0 , 1 and $\underline{E}$. Using our traditional arguments we arrive at Figure 2-3. In order to interpret Figure 2-3 intelligibly, it helps to think about what effect the transitive closure of E has on $G = \langle V, E \rangle$. E is always a subset of E^* . In E^* , every vertex lying on a cycle will have a loop. Also in E^* every connected component of G will become a complete subgraph. Thus the labelling in Figure 2-3 is interpreted as follows:

- a denotes $\underline{E}^* \cap 1$, non-loops not in the transitive closure of the edge set
- c denotes $E \cap 1$, edges in the graph
- d denotes $E \cap 1$, loops in the graph
- f denotes $\underline{E}^* \cap 1$, loops neither in the graph nor in the transitive closure of its edge set
- p denotes $E^* \cap \underline{E} \cap 1$, edges not in the graph but in the transitive closure of its edge set

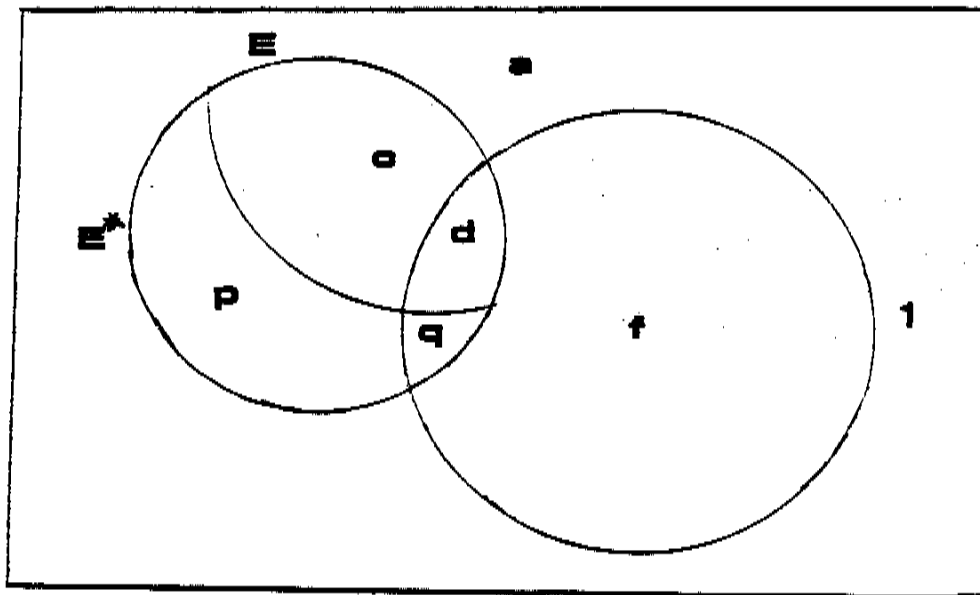


Figure 2-3: A Venn Diagram for Undirected Graphs in L_1^*

- q denotes $E^* \cap \bar{E} \cap 1$, loops not in the graph but in the transitive closure of its edge set

The interpretation of any term in L_1^* for an undirected graph is the union of some of these six regions. Any L_1^* -expression is interpreted as a statement of set equality between two such terms. There appear to be $2^6 = 64$ distinct L_1^* -terms. Using the same analysis we applied to L_1 , we see that an L_1^* -characteristic is a statement as to whether or not a subset is empty. This suggests the possibility of as many as 63 L_1^* -signatures. We interpret the first six on $G = \langle V, E \rangle$ to aid our analysis:

- a is empty means G is connected or $n < 2$.
- c is empty means G contains no edges
- d is empty means G is loopfree
- f is empty means there are no isolated vertices in G
- p is empty means every connected component in G is complete

- q is empty means every unisolated vertex in G has a loop in G

Given these interpretations we can now make some observations which substantially reduce the number of L_1^* -signatures. We abbreviate by omitting "is empty":

- If aq then f or c .

Explanation: If the graph is connected (a) but no new loops are derivable (q), then either all loops were already in the graph (f) or no edges were possible (ac) or $n < 2$ (acp or $acdfpq$).

- If c then pq .

Explanation: If the graph contained no edges (c), then E^* will be the same as E (pq).

- If dfq then $acdfpq$.

Explanation: If no loops are possible (dfq) then we have the empty graph ($acdfpq$).

- If adq then $acdfpq$ or $acdpq$.

Explanation: If adq then (by i) $adfq$ or $acdq$. If $adfq$ then (by iii) $acdfpq$. If $acdq$ then (by ii) $acdpq$ and $n < 2$ (acp).

- If a then f or cp .

Explanation: If G is connected (a) then every loop is in E or E^* (f) or $n < 2$ (acp or $acdfpq$).

- If dq then c .

Explanation: If every unisolated vertex is looped (q) and G is loopfree (d) then every vertex is isolated (c).

These interpretations leave us with only 22 signatures for undirected graphs in L_1^* , shown in Table 2-10.

The values for a, c, d, f, p, q , and n are very closely related. In particular, we can show that

$$d + f + q = n$$

$$a + c + p = n(n-1)/2$$

Class	Signature	Smallest n Value	Interpretation
1	d	4	disconnected, some edges, loopfree, some isolated vertices, not every connected component complete, not every unisolated vertex has a loop
2	f	4	disconnected, some edges, some loops, no isolated vertices, not every connected component complete, not every unisolated vertex has a loop
3	p	3	disconnected, some edges, some loops, some isolated vertices, every connected component complete, not every unisolated vertex has a loop
4	q	4	disconnected, some edges, some loops, some isolated vertices, not every connected component complete, every unisolated vertex has a loop
5	af	3	connected, some edges, some loops, no isolated vertices, not every connected component complete, not every unisolated vertex has a loop
6	df	5	disconnected, some edges, loopfree, no isolated vertices, not every connected component complete, not every unisolated vertex has a loop
7	dp	3	disconnected, some edges, loopfree, some isolated vertices, every connected component complete, not every unisolated vertex has a loop

Table 2-10: Undirected Graph Signatures in L_1^*

Class	Signature	Smallest n Value	Interpretation
8	fp	3	disconnected, some edges, some loops, no isolated vertices, every connected component complete, not every unisolated vertex has a loop
9	fq	4	disconnected, some edges, some loops, no isolated vertices, not every connected component complete, every unisolated vertex has a loop
10	pq	3	disconnected, some edges, some loops, some isolated vertices, every connected component complete, every unisolated vertex has a loop
11	adf	3	connected, some edges, loopfree, no isolated vertices, not every connected component complete, not every unisolated vertex has a loop
12	afp	2	connected, some edges, some loops, no isolated vertices, every connected component complete, not every unisolated vertex has a loop
13	afq	3	connected, some edges, some loops, no isolated vertices, not every connected component complete, every unisolated vertex has a loop
14	cpq	3	disconnected, edgeless, some loops, some isolated vertices, every connected component complete, every unisolated vertex has a loop

Table 2-10: Undirected Graph Signatures in L_1^* , continued

Class	Signature	Smallest n Value	Interpretation
15	dfp	4	disconnected, some edges, loopfree, no isolated vertices, every connected component complete, not every unisolated vertex has a loop
16	fpq	4	disconnected, some edges, some loops, no isolated vertices, every connected component complete, every unisolated vertex has a loop
17	adfp	2	1
18	afpq	2	1
19	cfpq	2	1
20	acdpq	1	$\langle \{1\}, \phi \rangle$
21	acfpq	1	$\langle \{1\}, \{11\} \rangle$
22	acdfpq	0	$\langle \phi, \phi \rangle$

Table 2-10: Undirected Graph Signatures in L_1^* , continued

if $c = 1$ then $f \leq n-2$

if $k(k-1)/2 < c \leq k(k+1)/2$ then $f \leq n-k$ for $k = 2, 3, 4, \dots$

$0 \leq q \leq \min(2c, n-d)$

$0 \leq p \leq \binom{n-f}{2}$

From these we observe that the values for c, d and n are independent variables and will determine the possible values for the dependant variables a, f, p and q .

2.5.3. Evaluation of L_1^*

L_1^* is a good refinement on L_1 for undirected graphs. For directed graphs, however, we will also have to consider the sets $\bar{E}$, $\underline{E}$, E^* , $\underline{E}^*$, E° , and $\underline{E}^\circ$. These lead to the unpleasant Venn diagram of Figure 2-4. The interpretations of the regions in the diagram become challenges to English grammar and resemble few properties appearing in graph theory texts. This awkwardness, coupled with a desire to explore recursive formulations, causes us to abandon further exploration

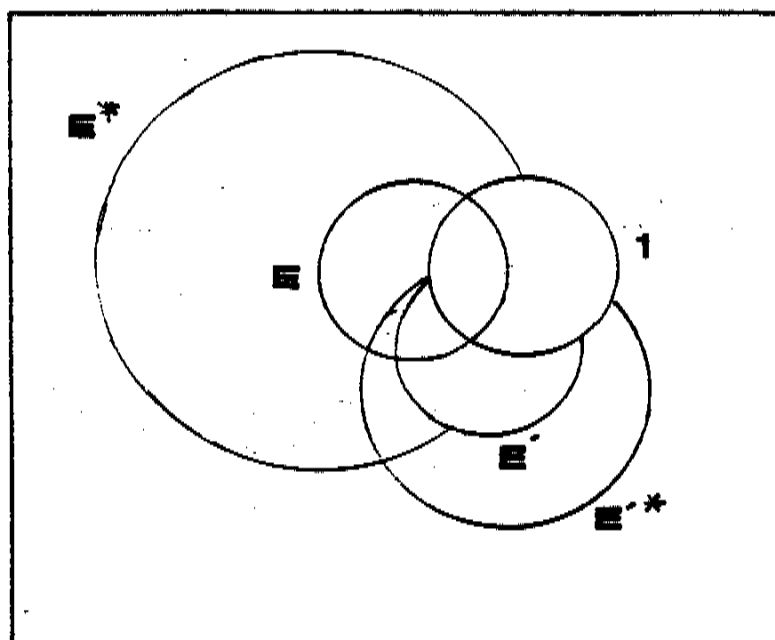


Figure 2-4: A Preliminary Venn Diagram for Directed Graphs in L_1^*

of L_1^* . The idea, however, of working with transformations (such as transitive edge closure) producing properties appearing in graph theory texts will not be abandoned. It motivates, as a matter of fact, the recursive formulation of graph theory discussed in Chapters 3 and 4.

2.6. The Edge-Set Languages: a Review

We conclude our exploration of the edge-set languages at this point. Each language refines the partition of the set of all finite graphs. The operations chosen for the grammars reflect our initial need to find similarities and differences in a set of graphs. The similarities and differences among graphs are readily available through their signatures. The fact that only finitely many, and far fewer than expected, properties appear, suggests that a primitive form of hashing based on the signature of a graph in an edge-set language, may be an acceptable solution for graphs of reasonable size (say $n < 17$).

There is no difference between the procedure for producing a graph with several specified edge-set language characteristics and that for only a single characteristic; both use the same generator. Similarly, testing for a set of characteristics uses the same procedure as testing for one. The edge-set languages describe very few of the graph properties customarily dealt with in books on graph theory. The recursive languages will attack this problem in the next two chapters.

CHAPTER 3

RECURSIVE LANGUAGES

...to prove even the smallest theorem [we] must use reasoning by recurrence, for that is the only instrument which enables us to pass from the finite to the infinite.

--Poincare

This chapter examines the fundamental concepts we use in the recursive description of graph properties. It begins with an explanation of incremental graph construction. Recursive graph grammars are defined and their components examined in detail. A minimality notion, the floor of a graph property, is discussed. We define inversion and present a technique for automated inverse construction. Finally, twenty three elementary recursive graph properties are described at length.

3.1. Graph Construction

This section introduces construction of graphs by a gradual, iterative process. The algorithm CONSTRUCT iterates toward a specific graph; the algorithm GENERATE iterates toward an arbitrary graph. The definition of a graph property (edgelessness) through a recursive algorithm motivates the remainder of the chapter.

A graph consists of finitely many vertices and finitely many edges. We therefore envision the creation of any graph as a construction process, in which we add one element (a vertex or an edge) at a time. Assume first that we have a specific goal, a graph we wish to copy. An algorithm to produce such a copy may be formulated recursively, and appears in Figure 3-1. CONSTRUCT has a target graph $G_T = \langle V_T, E_T \rangle$ which it is attempting to build from $G = \langle V, E \rangle$. Termination is guaranteed if CONSTRUCT is initially called on $(\langle V_T, E_T \rangle, K_1)$, beginning with the

```

CONSTRUCT(<VT,ET>,<V,E>)
  Either V ← V ∪ {x} for x ∈ VT x ∉ V
    or E ← E ∪ {yz} for y,z ∈ V, yz ∈ ET yz ∉ E
  If G = GT
  then halt
  else CONSTRUCT(<VT,ET>,<V,E>)

```

Figure 3-1: An Algorithm to Recursively Construct a Target Graph

smallest possible graph. (The empty graph will be studiously avoided.) Each iteration adds to G either a missing vertex in V or a missing edge in E between vertices already present in V . CONSTRUCT terminates when G is isomorphic to G_T . A trace of CONSTRUCT could be encoded as a sequencing of the set $V_T \cup E_T$ in which each edge xy is (not necessarily immediately) preceded by both x and y . There are many such construction sequences for any target graph G_T .

CONSTRUCT could be modified to produce an arbitrary graph. Rather than compare the progress of the algorithm against a target, we could "randomize" the process as in Figure 3-2.

```

GENERATE(<V,E>)
  Either V ← V ∪ {x} for x ∉ V
    or E ← E ∪ {yz} for y,z ∈ V, yz ∉ E
  Either output <V,E>
    or GENERATE(<V,E>)

```

Figure 3-2: An Algorithm to Recursively Generate Graphs

The initial call to GENERATE is on K_1 . GENERATE arbitrarily adds vertices and edges until it decides to halt. There is no guarantee that GENERATE will terminate, but at the end of each iteration its "current product" is a graph, and its output, if any, will always be a graph. Figure 3-3 show the iterative steps in "building" a graph during a sample run of GENERATE.

An alternative, recursive definition of graph might be: "A graph is K_1 or any output from GENERATE(K_1)." This definition *enumerates* the set of all graphs. The enumeration is in no particular order and may well be redundant because of the

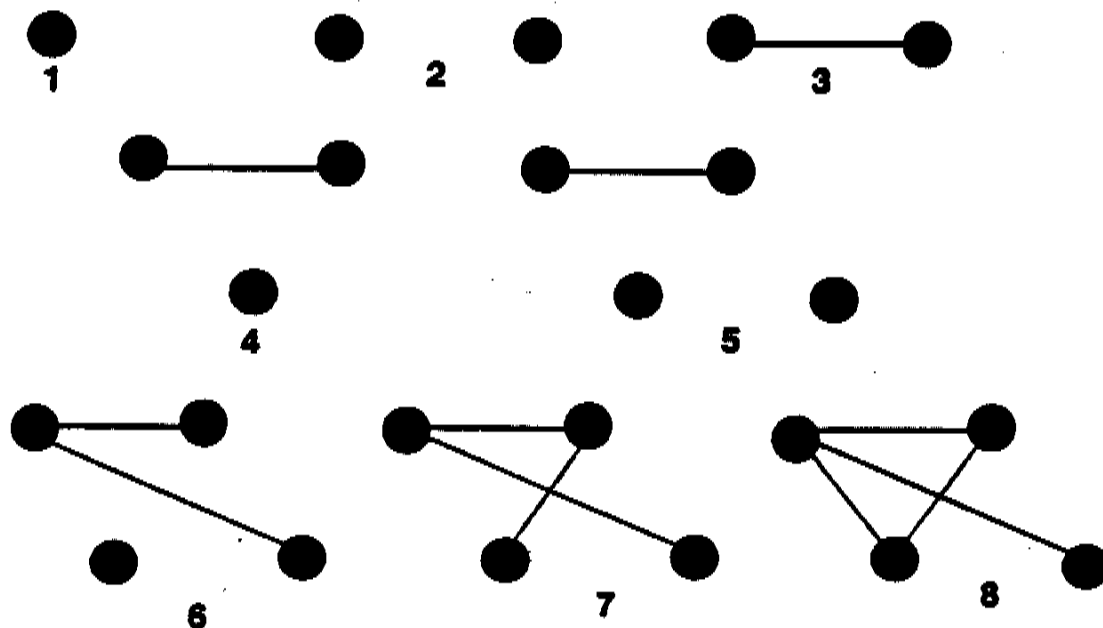


Figure 3-3: A Sample Run of GENERATE

many possible construction sequences. Yet, since every graph is constructable in this sense, the definition is equivalent to that given in Chapter 1.

A graph property is, as we have said, a partition of the set of all graphs into two classes: those graphs (G_p) which have the property and those which do not. Thus one way to define a graph property is to list all the graphs possessing it. For example "edgelessness" could be defined as:

$$G_E = \{(\langle 1 \rangle, \emptyset), (\langle 1, 2 \rangle, \emptyset), (\langle 1, 2, 3 \rangle, \emptyset), \dots\}$$

or if we let $E_k = \langle \{1, 2, 3, \dots, k\}, \emptyset \rangle$, more concisely, as:

$$G_E = \{E_k \mid k \text{ an integer, } k \geq 1\}$$

An alternative listing could be in the form of an algorithm which generated precisely that set: "Edgelessness is K_1 or any output from EDGELESS(K_1).". The algorithm EDGELESS appears in Figure 3-4. Figure 3-5 shows the iterative steps in a sample run of EDGELESS.

Let O_{edgeless} denote the set of all possible graphs output by EDGELESS. Since

EDGELESS($\langle V, E \rangle$)

$V \leftarrow V \cup \{x\}$ for $x \in V$

Either output $\langle V, E \rangle$

or EDGELESS($\langle V, E \rangle$)

Figure 3-4: An Algorithm to Generate Graphs without Edges

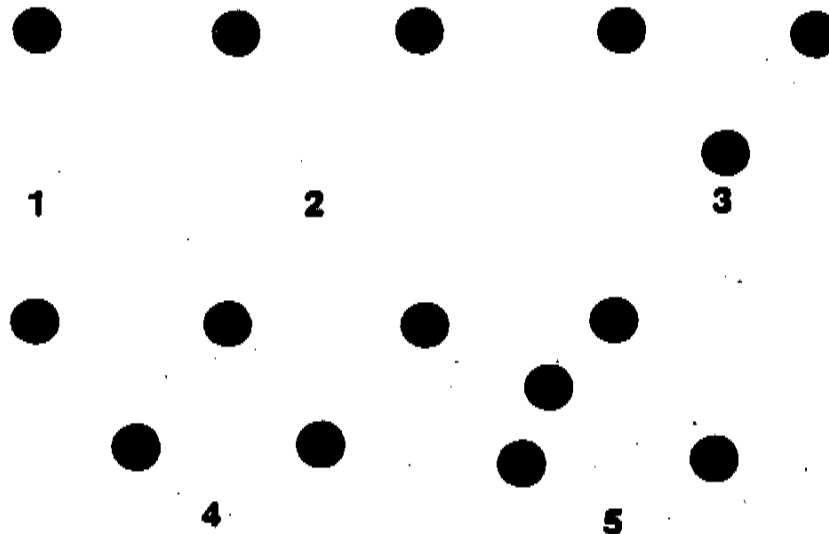


Figure 3-5: A Sample Run of EDGELESS

EDGELESS never changes E , E will remain empty, and every element of O_{edgeless} will be of the form $\langle V, \emptyset \rangle$, i.e., edgeless. Thus the algorithm produces only edgeless graphs and $O_{\text{edgeless}} \subseteq G_E$. Since any $\langle 1, 2, 3, \dots, k, \emptyset \rangle$ in G_E may be constructed by inserting 1, then 2, and so on, up to k , EDGELESS produces all edgeless graphs, i.e., $G_E \subseteq O_{\text{edgeless}}$. Therefore $G_E = O_{\text{edgeless}}$ and we have demonstrated the equivalence of the two definitions for edgelessness. EDGELESS is an example of a graph property definition in a recursive language.

on such a G has the R -property. In the event that no vertices or edges satisfy σ , or the algorithm "decides" to halt, "enough" is true. If G is any seed graph, the triple may be written grammatically as $(f\sigma)^*(G)$, i.e., "zero or more successive applications of f to G , each subject to selection criterion σ ." Thus an R -property p is a graph generator which may be stopped after any iteration, yielding a graph. The set of such possible outputs defines the graph property p . For example, if G is a seed graph, G , $f(G)$, $f^2(G)$ and $f^{17}(G)$ under σ may all be said to "have" the property $p = \langle f, S, \sigma \rangle$. Thus a variety of graphs having property p may be produced by varying G within the set described by S or varying the number of times f is applied. Even if those are kept constant, the selector σ makes arbitrary choices, so that several executions of $f^k(G)$ for fixed k and G will not necessarily produce the same graph, although all outputs will have property p .

A generator is said to be *correct* if every graph output by the generator must have property p . A generator is said to be *complete* if every graph with property p has an imaginable construction under some execution of the generator which makes appropriate selections on each iteration. (No attempt has been made, however, to prevent redundancy. A given generator may produce isomorphic graphs in different application sequences.) The triple $\langle f, S, \sigma \rangle$ is a valid syntactic representation of some graph property p if and only if the generator interpreted from the triple is correct and complete with respect the set of all graphs having property p .

Let us reexamine GENERATE and EDGELESS now within these definitions. Clearly the only seed graph for GENERATE is K_1 and the primitive operations we wish to allow are "add a vertex x ", which we shall denote as A_x , and "add an edge yz ", which we shall denote as A_{yz} . Then GENERATE is merely

$$(A_x + A_{yz})^*(K_1) \text{ where } x \in V, y, z \in V, yz \in E$$

The "+" sign denotes the option of choosing either A_x or A_{yz} on each iteration. We observe the addition of an element already in a set to that set can effect no change, and therefore revise GENERATE (and our recursive definition of a graph) to be:

$$(A_x + A_{yz})^*(K_1) \text{ where } y, z \in V$$

Here f is $A_x + A_{yz}$, S is $\{K_1\}$ and σ is " $y, z \in V$ ".

3.3. The Components of a Recursive Language

Although we have now established the nature of the terminal expression, it remains to identify the language R in which it lies. Thus we explore in this section the nature of the R -components P , L and Σ .

For the primitive language P we postulate some primitive operators, listed in Table 3-1. Each primitive operator is intended to modify a graph and return that modification.

Primitive	Effect	Representation
N	No change	$N\langle V, E \rangle = \langle V, E \rangle$
A_x	Add vertex x	$A_x\langle V, E \rangle = \langle V \cup \{x\}, E \rangle$
A_{xy}	Add edge xy	$A_{xy}\langle V, E \rangle = \langle V, E \cup \{xy\} \rangle$
D_x	Delete vertex x	$D_x\langle V, E \rangle = \langle V - \{x\}, E \rangle$
D_{xy}	Delete edge xy	$D_{xy}\langle V, E \rangle = \langle V, E - \{xy\} \rangle$
$I_{xv_1 \dots v_k}$	Identify vertices	$I_{xv_1 \dots v_k}\langle V, E \rangle = \langle V - \{v_1\}, E _{v_1 \leftarrow x} \rangle$
$F_{xv_1 \dots v_k}$	Fragment vertex x into vertices x and y	$F_{xv_1 \dots v_k}\langle V, E \rangle = \langle V \cup \{y\}, E _{xv_1 \rightarrow yv_1} \rangle$
L	Loop on all vertices	$L\langle V, E \rangle = \langle V, E \cup 1 \rangle$
$\underline{L}$	Unloop on all vertices	$\underline{L}\langle V, E \rangle = \langle V, E \cap 1 \rangle$

Table 3-1: Primitive Operators for R -Grammars

A primitive operator makes no assumptions about its ability to perform its operation

meaningfully; x and/or y may or may not be present in V and xy may or may not be present in E . Operators are provided for addition and deletion of a vertex (A_x , D_x) or an edge (A_{xy} , D_{xy}), for the merger of $k+1$ vertices ($I_{xv_1 \dots v_k}$), for the splitting of one vertex into two ($F_{xyv_1 \dots v_k}$), and for the introduction (L) and removal ($\underline{L}$) of loops on all the vertices.

Now we postulate some possible P 's for the R-grammar $\langle P, L, \Sigma \rangle$. Each P has a set $\Pi = \{\pi_1, \pi_2, \dots, \pi_k\}$ of terminal symbols and the following grammatical rules:

$$I \rightarrow I + I \mid I \mid I$$

$$I \rightarrow \pi_1 \mid \pi_2 \mid \dots \mid \pi_k$$

This P grammar permits both primitive operators (members of Π) and composite operators constructed from them. With this grammar understood, it is sufficient to define a P language by its terminal symbols. In particular we define:

$$P_1 = \{A_x, A_{xy}, N\}$$

$$P_2 = P_1 \cup \{D_x, D_{xy}\}$$

$$P_3 = P_2 \cup \{L, \underline{L}\}$$

$$P_4 = P_2 \cup \{I_{xy}, F_{xy}\}$$

$$P_5 = P_3 \cup \{I_{xy}, F_{xy}\}$$

Note that P_1 will be adequate for GENERATE.

Composite operators are introduced for conceptual and notational convenience. Each is expressible as a combination of primitive operators. For operators f and g , the composite fg means "first apply g and then apply f ." For operators f and g , the composite $f + g$ means "apply exactly one of f or g ." The function for a graph property could involve both kinds of composition, evolving forms such as $ff' + gg'$ or $(f + f')(g + g')$. We have found some composite operators to be so useful in developing graph properties that we have assigned them their own symbols. These appear in Table 3-2.

We stress again that no operator, primitive or composite, is assumed to be applicable to an arbitrary graph. The selection conditions in σ (such as "distinct

Composite	Effect	Equivalence
S_{xvy}	Subdivide edge xy by vertex v	$D_{xy} A_{xv} A_{vy} A_v$
B_{xy}	Branch from x to y	$A_{xy} A_y$
F_x	Fully connect vertex x to A	$A_{v_1x} A_{v_2x} \dots A_{v_nx} A_x$
$Y_{u_1 \dots u_k}$	Add cycle $u_1 u_2 \dots u_k u_1$	$A_{u_1 u_2} A_{u_2 u_3} \dots A_{u_{k-1} u_k} A_{u_k u_1} A_{u_1} A_{u_2} \dots A_{u_k}$
$\bar{Y}_{u_1 \dots u_k}$	Delete cycle $u_1 u_2 \dots u_k u_1$	$D'_{u_1} D'_{u_2} \dots D'_{u_k} D_{u_1 u_2} D_{u_2 u_3} \dots D_{u_{k-1} u_k} D_{u_k u_1}$

where $D'_i = D_i$ if degree of i is 0
 $= N$ else

Table 3-2: Some Composite Operators for R-Grammars

$u_1, u_2, \dots, u_k$ to guarantee that a cycle is simple) place restrictions on the bindings of the variables referred to by the operator. The complexity of any algorithm is dependent both on the matching required by σ and the resources needed to update the graph.

For L in the R-grammar $\langle P, L, \Sigma \rangle$, we can use any graph property language. In particular, the languages $L_1, L_2, L_3, L_{1n}, L_{2n}$ and L_{3n} of Chapter 2 are excellent candidates. It is also permissible for L itself to be a recursive graph property language. (We shall have more to say about this later.) We will also reluctantly permit L_0 , the language which precisely lists the vertices and edges of a graph.

The Σ in the R-grammar $\langle P, L, \Sigma \rangle$ will affect the complexity of the algorithm. Any constructive algorithm to produce a specific graph will be at least $O(\max(m, n))$ as long as $m+n$ is increasing from one iteration to the next and the selector is not of greater order. (We employ the traditional definitions for the order of an

algorithm throughout.) Thus we focus wherever possible on simple selector languages (preferably of $O(1)$ or $O(n)$), leaving the data structure implicit. In Σ vertices are $v_1, v_2, \dots$ and edges are ordered pairs of vertices. We offer the following selector languages with their generating grammars. Many others are, of course, possible:

Σ_1

A formal grammar for Σ_1 is

I	$\rightarrow I, I \mid \text{vertex sign } V \mid \text{edge sign } E$
vertex	$\rightarrow v_1 \mid v_2 \mid v_3 \mid \dots$
edge	$\rightarrow (\text{vertex}, \text{vertex})$
sign	$\rightarrow \in \mid \neq$

Selector expressions such as

$x \in V$

or

$yz \in E$

are possible in Σ_1 . Note that we could produce the selector for GENERATE in Σ_1 as follows:

I	$\rightarrow I, I$
	$\rightarrow \text{vertex sign } V, \text{vertex sign } V$
	$\rightarrow y \in V, z \in V$

Σ_2

A formal grammar for Σ_2 is

I	$\rightarrow I, I \mid \text{vertex sign } V \mid \text{edge sign } E \mid \text{vertex} \neq \text{vertex} \mid$
	$\text{edge} \neq \text{edge}$
vertex	$\rightarrow v_1 \mid v_2 \mid v_3 \mid \dots$
edge	$\rightarrow (\text{vertex}, \text{vertex})$
sign	$\rightarrow \in \mid \neq$

Σ_2 contains all the terminal expressions of Σ_1 . In addition, selector expressions such as

$x \in V, y \in V, x \neq y$

are possible in Σ_2 , but not in Σ_1 . The expression $\neq$ will be interpreted semantically as "is distinct from." The expression $x \neq y$ would therefore mean x and y are

distinct vertices. The expression " $yz \neq vw$ " for undirected graphs means that neither " $y = v$ and $z = w$ " nor " $y = w$ and $z = v$ " is true. The expression " $yz \neq vw$ " for directed graphs means " $y \neq v$ and $z \neq w$."

Σ_3

A formal grammar for Σ_3 is

I	$\rightarrow I, I \mid \text{vertex sign } V \mid \text{edge sign } E \mid \text{vertex} \neq \text{vertex} \mid$ $\text{edge} \neq \text{edge} \mid d(\text{vertex}) \text{ rel number}$
vertex	$\rightarrow v_1 \mid v_2 \mid v_3 \mid -$
edge	$\rightarrow (\text{vertex}, \text{vertex})$
sign	$\rightarrow = \mid \neq$
rel	$\rightarrow = \mid > \mid \geq \mid < \mid \leq$
number	$\rightarrow 0 \mid 1 \mid 2 \mid \dots$

Σ_3 contains all the terminal expressions of Σ_1 and Σ_2 . In addition, selector expressions such as

$x \neq V, d(x) > 1$

are possible in Σ_3 , but not in Σ_1 or Σ_2 . We define $N(x)$, the *neighborhood* of a vertex, to be the set of vertices adjacent to x , other than x itself, i.e.,

$N(x) = \{xy \mid y \in V, xy \in E, x \neq y\}$

We then define the *degree* of a vertex x to be the cardinality of $N(x)$ with the stipulation that the degree is non-zero (say, one half) when the neighborhood is empty but there is a loop on x , i.e., $xx \in E$. This emphasis on loops is intentional and will be clarified in Chapter 4. The expression $d(x)$ will be interpreted semantically as the degree of vertex x .

Σ_4

A formal grammar for Σ_4 is

I	$\rightarrow I, I \mid \text{vertex sign } V \mid \text{edge sign } E \mid \text{vertex} \neq \text{vertex} \mid$ $\text{edge} \neq \text{edge} \mid d(\text{vertex}) \text{ rel number}$
vertex	$\rightarrow v_1 \mid v_2 \mid v_3 \mid -$
edge	$\rightarrow (\text{vertex}, \text{vertex})$
sign	$\rightarrow = \mid \neq$
rel	$\rightarrow = \mid > \mid \geq \mid < \mid \leq$
number	$\rightarrow \max \mid n \mid 0 \mid 1 \mid 2 \mid \dots$

Σ_4 contains all the terminal expressions of Σ_1 , Σ_2 and Σ_3 . In addition, selector expressions such as

$$x \in V, d(x) = \max$$

are possible in Σ_4 , but not in Σ_1 , Σ_2 or Σ_3 . The expression $\max$ will be interpreted semantically as the maximum degree of a vertex in G . The expression n will be interpreted semantically as the cardinality of V .

Σ_5

A formal grammar for Σ_5 is

	$\rightarrow \sim(l) \mid l, l \mid \text{vertex sign } V \mid \text{edge sign } E \mid$
	$\text{vertex} \neq \text{vertex} \mid \text{edge} \neq \text{edge} \mid$
	$d(\text{vertex}) \text{ rel number} \mid d(\text{vertex}) \text{ rel variable sum number} \mid$
	$ \{\text{vertexset}\} \cap V \text{ rel number} \mid$
	$ \{\text{edgeset}\} \cap E \text{ rel number} \mid \text{variable rel number} \mid$
	$ \{\text{vertexset}\} = V $
vertex	$\rightarrow v_1 \mid v_2 \mid v_3 \mid \dots$
edge	$\rightarrow (\text{vertex}, \text{vertex})$
sign	$\rightarrow = \mid \neq$
rel	$\rightarrow = \mid > \mid \geq \mid < \mid \leq$
number	$\rightarrow n \mid 0 \mid 1 \mid 2 \mid \dots$
vertexset	$\rightarrow \text{vertex} \mid \text{vertex}, \text{vertexset} \mid \{\text{vertex} \mid l\}$
edgeset	$\rightarrow \text{edge} \mid \text{edge}, \text{edgeset} \mid \{\text{edge} \mid l\}$
variable	$\rightarrow i \mid j \mid k \mid \dots$
sum	$\rightarrow + \mid -$

The expression $\sim(l)$ is interpreted semantically as "not l ," providing the negation of any expression. The expression $|A \cap B|$ is interpreted semantically as the cardinality of the intersection of the sets A and B . An expression of the form variable $k \geq n$ is intended to refer to the index numbers on the vertices. (See the property EULERIAN for an example.) Σ_5 contains all the terminal expressions of Σ_1 , Σ_2 , and Σ_3 , but not Σ_4 , because Σ_5 lacks $\max$. Selector expressions such as

$$k \geq 5$$

or

$$|\{x, y, z\} \cap V| \neq 2$$

are possible in Σ_5 , but not in Σ_1 , Σ_2 , Σ_3 or Σ_4 .

Σ_6

A formal grammar for Σ_6 is the Σ_5 grammar with one additional production:

$I \rightarrow \text{statement}$

"Statement" is any English language sentence. Σ_6 is a deliberate catchall, to be used, like L_Q , when all else fails.

In the event that the selector makes no restrictions at all, Σ is designated as ϕ .

We have now assembled the raw material from which to construct R-languages. We have arbitrarily mentioned five primitive languages P, eight seed languages L, and six selector languages Σ . (The reader is invited to define additional appropriate languages.)

3.4. The Floor of a Graph Property

Part of the challenge in writing a graph property recursively is choosing an appropriate recursive language. This section explores a minimality condition for R-properties. GENERATE and EDGELESS are again used as examples.

We return now to GENERATE, which we had identified as

$$(A_x + A_{yz})^*(K_1) \text{ where } y, z \in V$$

Clearly $f = A_x + A_{yz}$ could lie in any of the P's, $S = K_1$ in any of the L's and $y, z \in V$ in any of the Σ 's. Since the power of the languages we define is a primary concern, it seems reasonable to seek a minimal R-language for each property. Implicit in their definitions were partial or full orderings for the P's, the L's and the Σ 's. Diagrams of these orderings are pictured in Figure 3-6. Arrows point from less powerful languages to the more powerful ones which contain them.

We define a *floor* of a graph property $p = \langle f, S, o \rangle$ to be an R-grammar $\langle P, L, \Sigma \rangle$ such that for every other R-grammar $\langle P', L', \Sigma' \rangle$ in which p is a terminal expression, either $P' > P$ or $L' > L$ or $\Sigma' > \Sigma$. Note that, because of the partial

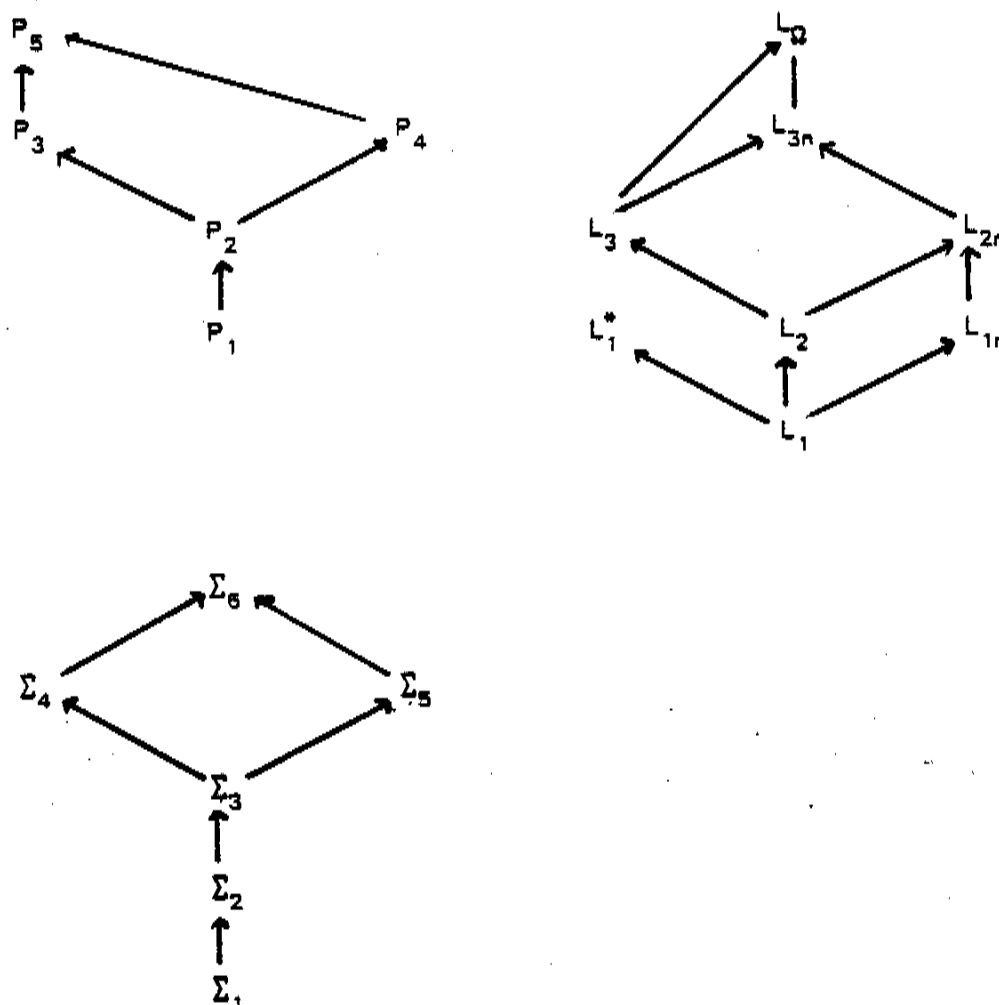


Figure 3-6: Orderings for Property Languages

order, a graph property may have more than one floor. Intuitively, we are identifying the weakest possible grammar(s) enabling the property. For GENERATE we have indicated that P_1 and Σ_1 are adequate. Since the expressions $\underline{E} \cap \underline{1} = 0$ and $\underline{E} = 0$ in L_1 are simultaneously satisfied only by graphs isomorphic to K_1 , L_1 is also adequate for GENERATE. Thus the floor for GENERATE is $\langle P_1, L_1, \Sigma_1 \rangle$. Similarly, EDGELESS may be written compactly as:

$$(A_x)^*(K_1)$$

Here the floor is $\langle P_1, L_1, \epsilon \rangle$ because no selector at all is required. As more complex graph properties are introduced, other floors will be required. It is interesting to note that EDGELESS has a lower floor than GENERATE. Although the property of being an edgeless graph is a special case of being a graph, the minimal

R-grammar required to achieve it is (counterintuitively) *less* complex.

3.5. Inversion

Now that we understand the nature of a recursively-defined graph property, this section defines the inverse of a graph property and the implications of its automatic construction.

If a graph may be constructed one edge or vertex at a time, it may also be dismembered in the same fashion. Given algorithm $p = \langle f, S, \sigma \rangle$ which generates precisely those graphs with property p , under certain circumstances it is possible to calculate an inverse, call it p^{-1} . This new algorithm methodically attempts to dismember an input graph until it is again a seed for p . Each testing algorithm may be stopped after any iteration, yielding a graph with the same truth value for the property being tested as each of the preceding graphs. For example, if p is the property of being Eulerian and the graph G is not Eulerian, then the graphs $f^{-1}(G)$ and $f^{-4}(G)$, if they exist, will also be non-Eulerian. More formally, a terminal R-expression $p^{-1} = \langle f^{-1}, S, \sigma^{-1} \rangle$ is said to be the *inverse* of another terminal R-expression $p = \langle f, S, \sigma \rangle$ if and only if the *testing* semantic interpretation of p^{-1} returns "TRUE" on all outputs of the *generator* which is the R-property defined by p , and "FALSE" on all other graphs. The testing semantic interpretation of $p^{-1} = \langle f^{-1}, S, \sigma^{-1} \rangle$ is the following recursive algorithm:

$f^{-1}(G)$ = TRUE if G is described by S
 = $f^{-1}(G')$ where $G' = f^{-1}(G)$ using elements from
 G selected by σ^{-1} in order to apply f^{-1}
 = FALSE if G is not described by S and σ^{-1} is
 not applicable

Note that *every* selection suiting the requirements in σ guarantees the correct results, not simply *some* selection. If G was a product of p , its dismemberment sequence and seed graph need not mirror its construction sequence and seed graph, since there are likely to be many such correct choices. If G has property p , and only if it does, repeated applications of f^{-1} will return it, in some fashion, to some

seed graph of p . A concise grammatical representation of p^{-1} is $(f^{-1}\sigma^{-1})^*(G)$. The seed set S is implicit in this representation. Thus $(f^{-1}\sigma^{-1})^*(G)$ is interpreted as "apply f^{-1} until an answer is reached, i.e., G is described by S or f^{-1} cannot be applied because σ^{-1} fails." The number of iterations required for testing may vary with σ^{-1} and is not in general predictable.

Certain graph properties have inverses which may be computed automatically from p . It is those properties which we examine in this chapter. It may be argued that a computer which is taught to generate objects with a given property, and can then calculate a procedure to test input objects for that property, has *understood* the nature of the property and has *learned* it. Thus we argue that this computation of a testing algorithm from a generator algorithm is both automated deduction and machine learning.

3.6. Automated Inversion

Having explained the significance of an inverse, in this section we present a mechanism for its construction. Not every R-property will be invertible via this mechanism. In particular, consider an R-property whose formulation includes I_{xy} , the primitive operator which identifies or merges vertex x with vertex y , leaving only the revised vertex x in the graph and assigning all the adjacencies of y to x . After such a merger occurs there is no indication of which vertex is the revised one, let alone which edges incident with x were attributable to x , to y , or to both of them. Thus a property whose formulation includes I_{xy} will not always be susceptible to inversion. Similarly, an R-property which employs the primitive operators L or $\bar{L}$, looping or unlooping all the vertices, will obscure the prior loop status. For properties which exploit loops, such inversion also presents a problem. This loss of information frequently causes difficulties for inversion, some of which are dealt with in Chapter 4. Properties whose floor requires P_3 or higher are rarely considered in this chapter, and a formulation of a given property with the lowest floor is always preferred.

Here is the technique for the automatic construction of p^{-1} from p . We emphasize that this technique is guaranteed only for R-properties whose floor includes P_1 or P_2 and that, under certain circumstances, it may not be applicable even to those.

Each of the five primitive operators under consideration should have a fairly obvious inverse, for example, we expect $A_x^{-1} = D_x$. Recall, however, that the formulation of EDGELESS was originally

$$A_x^*(K_1) \text{ where } x \in V$$

and was modified to

$$A_x^*(K_1)$$

If we were to undo each step in the construction of some edgeless graph G , we might find an instance of inverting the "addition" of some vertex that was in the graph prior to the "addition." Since the second addition made no change to the graph, the inverse of that addition should also make no change. Thus we have

$$\begin{aligned} A_x^{-1} &= D_x \text{ if } x \in V \text{ before } A_x \\ &= N \text{ else} \end{aligned}$$

and

$$\begin{aligned} A_{xy}^{-1} &= D_{xy} \text{ if } x \in E \text{ before } A_{xy} \\ &= N \text{ else} \end{aligned}$$

Rather than engage in existential debates, we prefer to invert the less elegant, more constricted algorithm formulation which avoids ineffectual iterations. Thus, although GENERATE is more concise as

$$(A_x + A_{yz})^*(K_1) \text{ where } y, z \in V$$

it is easier to invert as

$$(A_x + A_{yz})^*(K_1) \text{ where } x \in V, y, z \in V, yz \in E$$

A_x may be applied to a graph $G = \langle V, E \rangle$ whether or not x is in V . Inverting a particular application of A_x is an uncertain procedure because we have no way of knowing if A_x was *effective*, i.e., changed V . Similarly, A_{xy} may be meaningfully applied as long as x and y are in V , whether or not xy is in E . Again we have no way of knowing whether A_{xy} was effective. With the deletion operators D_x and

D_{xy} any meaningful application ($x \in V$ or $x,y \in V$, $xy \in E$) must also be effective. Hence we do not have the same tentativeness associated with D_x^{-1} and D_{xy}^{-1} . We also note that the inverse of the null operator N is itself. We now list five rules for the automatic construction of an inverse $p^{-1} = \langle f^{-1}, S, \sigma^{-1} \rangle$ from an R-property $p = \langle f, S, \sigma \rangle$. The initial rules are designed to construct f^{-1} from f .

RULE 1

Every primitive operator in P_2 has an inverse. The inverses are

$$\begin{aligned} A_x^{-1} &= D_x \\ A_{xy}^{-1} &= D_{xy} \\ D_x^{-1} &= A_x \\ D_{xy}^{-1} &= D_{xy} \\ N^{-1} &= N \end{aligned}$$

The inversion of the other primitive operators usually entails loss of information and is not discussed here.

RULE 2

The inverse of a sequential composite is the inverse of its elements, in the reverse order, i.e.,

$$(fg)^{-1} = g^{-1}f^{-1}$$

For example,

$$(A_{xy}A_y)^{-1} = A_y^{-1}A_{xy}^{-1} = D_yD_{xy}$$

RULE 3

The inverse of an additive composite is the sum of the inverses of its elements, in the same order, i.e.,

$$(f + g)^{-1} = f^{-1} + g^{-1}$$

For example,

$$(A_x + A_{yz})^{-1} = A_x^{-1} + A_{yz}^{-1} = D_x + D_{yz}$$

RULE 4

The inverse of an uncertain addition is a tentative deletion, i.e., if it is not known whether $d(x) = 0$ when f^{-1} arrives at D_x use

$$\begin{aligned} A_x^{-1} = D'_x &= D_x \text{ if } d(x) = 0 \\ &= N \text{ else} \end{aligned}$$

The construction of σ^{-1} from σ is a bit more complex. It is here that the inversion technique may fail. The major inversion heuristic is that the vertices and edges involved in the f iteration just completed are either immediately identifiable, so that $f(G)$ may be returned to G , or belong to a set of possible choices, any of which will move $f(G)$ back correctly toward some seed graph of p or FALSE, without necessarily returning to G at all. We define the *profile* of a variable to be a (not necessarily exhaustive) list of its distinguishing features in a selection language, for example " $x \in V, d(x) = 1$." A *pre-profile* is a profile immediately before the application of an operator. Although σ initially constitutes a profile, we expand σ to σ_{pre} . This new pre-profile excludes ineffectual (equivalent to N) operations. σ_{pre} also includes the properties of the seed preserved under f . A *post-profile* is a profile immediately after the application of an operator. For most cases, the construction of σ^{-1} from σ is embodied in

RULE 5

Let σ be a pre-profile of those variables involved. Expand σ to σ_{pre} . Compute the changes to σ_{pre} caused by f . The new description, σ^{-1} , is in Σ_4 . σ^{-1} is now a post-profile of the variables after f . (If the selector language for p is Σ_5 or Σ_6 , the new description must also be constructed in Σ_5 in Σ_6 .)

In other words, p singles out a variable x by its relationship to G and then applies f to it, changing in some fashion the nature of x with respect to G . This new description of x enables us to select it for inversion. What aspects of x (or xy) are significant? Most of the graph properties in this chapter find membership with respect to V and E , distinctness, degree of a vertex and maximum degree of any vertex in the graph to be an adequate perspective, hence the choice of Σ_4 . It is important to recognize that the floor may shift during inversion, i.e., the inverse may be stated in a more or less complex R -language.

Throughout this document, inverses whose floors are based on Σ_1 , Σ_2 , Σ_3 or Σ_4 are computed automatically. As simple examples, we compute the inverses of EDGELESS and GENERATE. (More complex examples are available in subsequent

sections.) For EDGELESS we have

$$f^{-1} = A_x^{-1} = D_x$$

In this example the initial pre-profile σ is empty. We expand σ to $\sigma_{pre} = x \in V$ to exclude the ineffectual operation of adding an already present vertex. Immediately after A_x we know that $x \in V$ and $d(x) = 0$. Thus we set σ^{-1} to $x \in V, d(x) = 0$. Since the maximum degree of a vertex in G could not be altered by the addition of a vertex of degree zero, the max is not mentioned in σ^{-1} . Therefore, the following algorithm tests to see if an arbitrary graph is edgeless:

$$\begin{aligned} f^{-1}(G) &= \text{TRUE if } G \text{ is } K_1 \\ &= f^{-1}(D_x(G)) \text{ where } x \in V, d(x) = 0 \\ &= \text{FALSE if } G \text{ is not } K_1 \\ &\quad \text{and there does not exist } x \in V \text{ such that } d(x) = 0 \end{aligned}$$

This edgelessness tester deletes vertices of degree zero until it arrives at the empty graph (success and the input graph was edgeless) or all vertices are of degree greater than zero (failure and the input graph was not edgeless). In Figure 3-7 we show the algorithm operating on a graph $G \in G_p$ and a graph $G \notin G_p$.

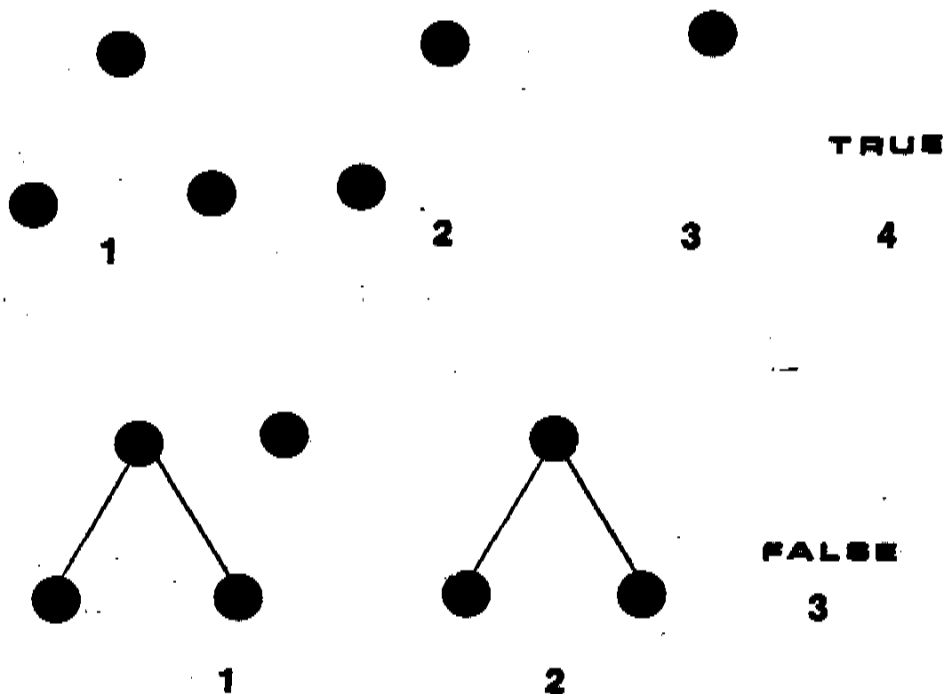


Figure 3-7: EDGELESS^{-1} in Operation

For GENERATE we have

$$f^{-1} = (A_x + A_{yz})^{-1} = A_x^{-1} + A_{yz}^{-1} = D_x + D_{yz}$$

The pre-profile σ is $yz \in E$, which we expand to $x \in V$, $yz \in V$, $yz \in E$. The post-profile σ^{-1} is $x, y, z \in V$, $yz \in E$, $d(x) = 0$. Since the maximum degree of a vertex in G could not be altered by A_x and is unpredictable under A_{yz} , the max is not mentioned in σ^{-1} . This yields the following algorithm for testing to see if an input ordered pair of sets (V, E) is a graph:

$f^{-1}(G)$ = TRUE if G is K_1
 = $f^{-1}((D_x + D_{yz})(G))$ where $x, y, z \in V$, $yz \in E$, $d(x) = 0$
 = FALSE if G is not K_1
 and there does not exist $x \in V$ such that $d(x) = 0$
 and there do not exist $y, z \in V$ such that $yz \in E$

Note that the selector variables are grouped for convenience of notation, but that they need not all be successfully bound in order for σ^{-1} to succeed, i.e., we need to find x or yz but not both, as distinguished by separate lines in the FALSE selector σ^{-1} . In Figure 3-8 we show the algorithm operating on a graph $G \in G_p$ and a graph $G \in G_p$.

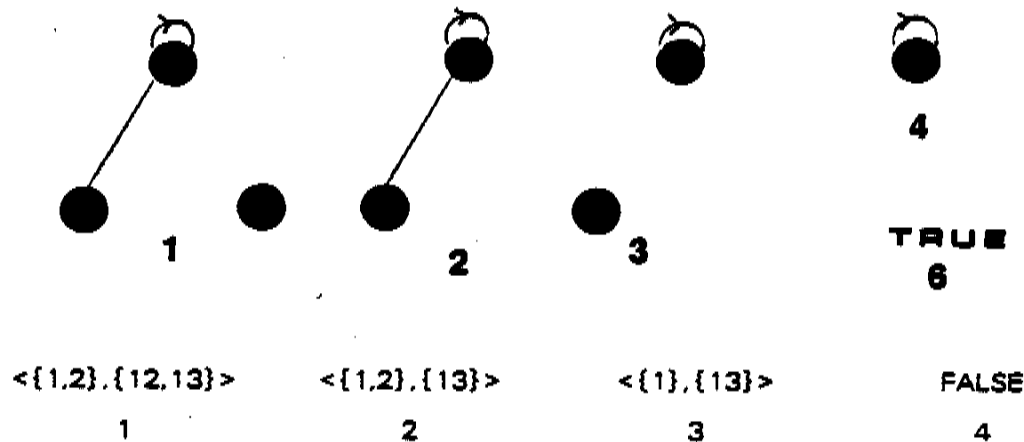


Figure 3-8: GENERATE^{-1} in Operation

From now on we will describe inverses merely by stating f^{-1} , σ_{pre} and σ^{-1} .

We observe that in both of these examples the floor shifts for the inverse.

For GENERATE the floor was $\langle P_1, L_1, \Sigma_1 \rangle$ and for GENERATE^{-1} it is $\langle P_1, L_1, \Sigma_3 \rangle$. For EDGELESS the floor was $\langle P_1, L_1, \phi \rangle$ and for EDGELESS^{-1} it is $\langle P_1, L_1, \Sigma_3 \rangle$. Because the post-profile is constructed with respect to Σ_4 whenever possible, we expect the floor for p^{-1} to involve Σ_4 , regardless of the Σ used in p . To the extent that features of Σ_4 are not applicable, lower Σ 's will appear for p^{-1} . Thus a shift in the floor suggests that perhaps the "true" context of a property resides in the more powerful of the two R-languages. We will pursue this further in Chapter 5.

3.7. Readily Invertible Graph Properties

All the fundamental concepts in our recursive formulation of graph theory are now established. Each of the segments in this section deals with a specific graph property. Each segment begins with the necessary definition(s) from graph theory. The R-property is formulated, proved correct, inverted and proved complete. Many graph properties have more than one formulation within a given R-grammar. In some instances, more than one valid formulation is provided, with relevant explanations.

In order to prove that an R-property is complete, we need only show that its inverse is correct. The situation is pictured in Figure 3-9. For $p = \langle f, S, \sigma \rangle$, f maps $G \in S$ into G_p and f maps G_p into G_p , where G_p is the set of all graphs with property p . For the inverse $p^{-1} = \langle f^{-1}, S, \sigma^{-1} \rangle$, f^{-1} maps $G \in G_p$ into G_p , and eventually back into S . Assume that f^{-1} only maps $G \in G_p$ into $f^{-1}(G) \in G_p$, i.e., f^{-1} is correct. Let $G \in G_p$. Since f^{-1} is defined on G and f^{-1} is correct, there exists some sequence of applications $f^{-1*}(G)$ which is a "trail" back to some seed graph $H \in S$. We need only automatically invert these applications into $f^*(H)$ to create a "trail" from H to G . Thus G is "reachable" via f and p is complete. Our completeness proofs will therefore consist in showing that the "automatic" inverse is correct.

Frequently there are several possible formulations for a graph property. Occasionally we will show more than one. In constructing the properties in this

$$\begin{aligned}
&= ((N + A_{xv_1}) \dots (N + A_{xv_r}) A_{xy} F_{xv_1 \dots v_r})^{-1} \\
&\quad + ((N + A_{wt_1}) \dots (N + A_{wt_{s-k+2}}) A_{wt_{s-k+3}} \dots A_{wt_{s+1}} \\
&\quad A_{wz} F_{wz t_1 \dots t_{s+1}})^{-1} \\
&= I_{xy} D_{xy} (N + D_{xv_r}) \dots (N + D_{xv_1}) + \\
&\quad I_{wz} D_{wz} D_{wt_{s+1}} \dots D_{wt_{s-k+3}} (N + D_{wt_{s-k+2}}) \dots \\
&\quad (N + D_{wt_1})
\end{aligned}$$

 σ_{pre}

= distinct $x, v_i \in V$, $y \in V$, $xv_i \in E$,

$k-1 \leq r \leq d(x) - k+1$, $d(x) > k-1$, $d(v_i) > k-1$

distinct $w, t_i \in V$, $z \in V$, $wt_i \in E$, $s = d(w) - k+1$,

$d(w) > k-1$, $d(t_i) > k-1$

 σ^{-1}

= distinct $x, y, v_i \in V$, $xy, yv_1, \dots, yv_r \in E$,

$k-1 \leq r < d(x) - k + 1$, $d(x) > k-1$, $d(y) > k-1$,

$d(v_i) > k-1$,

$|\{pv_i \mid p \in E, pv_i \in E, p \neq x, p \neq y\}| > k-2$,

$i=1, 2, \dots, r$

distinct $w, z, t_i \in V$, $wz, zt_1, \dots, zt_s, wt_{s-k+3}, \dots, wt_{s+1} \in E$,

$s = d(w) - k + 1$, $d(w) > k-1$, $d(z) > k-1$,

$|\{pt_i \mid p \in E, pt_i \in E, p \neq w, p \neq z\}| > k-2$,

$d(t_i) > k-1$, $i=1, 2, \dots, s$

The floor remains constant. Now the inversion procedure has noted that the degree of each $v_i(t_i)$ is at least k , and that each $v_i(t_i)$ will be adjacent to at least $k-1$ vertices other than x and y (w and z). Figure 3-81 shows $K\text{-CONNECTED}^{-1}$ operating on a graph $G \in G_p$ and a graph $G \in G_p$ for $k = 3$. A graph is k -connected if and only if there exist at least k edge-disjoint paths between any two vertices. $K\text{-CONNECTED}^{-1}$ collapses adjacent vertices x and y of degree at least $k-1$; any path available through x or y will now be available through the resulting merged x . In particular at most one xq -path (for any $q \in V$) used the edge xy and that path after the merger will be available directly from x . Thus $K\text{-CONNECTED}^{-1}$ performs properly on $G \in G_p$ and will eventually reduce it to K_{k+1} . If $G \in G_p$, there are $k-1$ points which disconnect it. This ability to

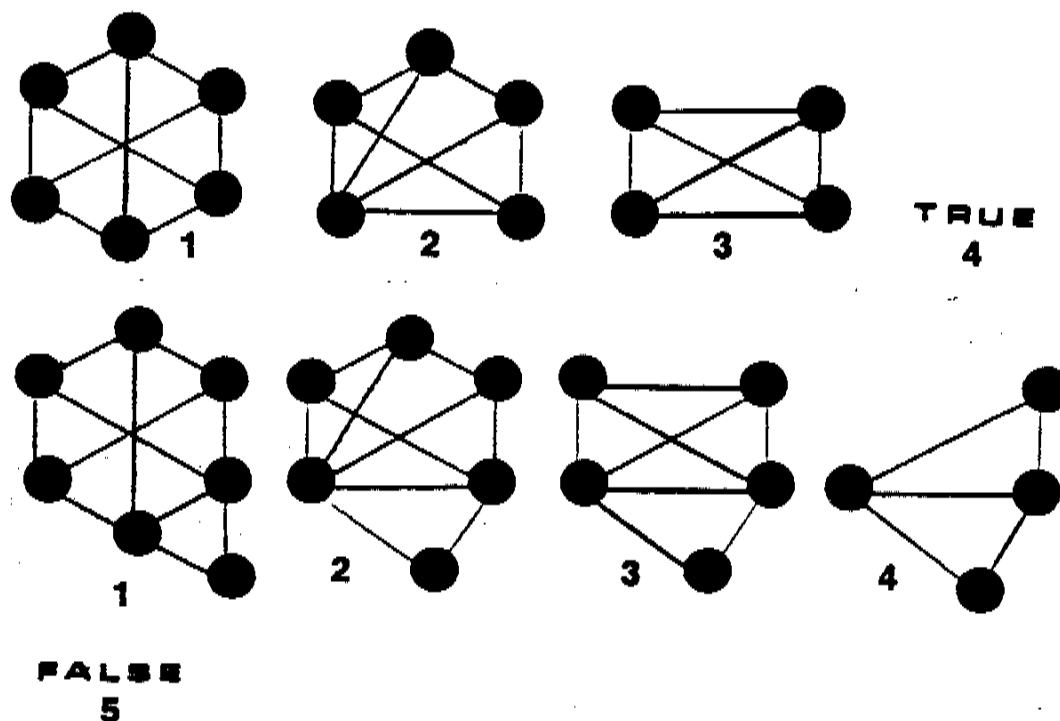


Figure 3-81: 3-CONNECTED^{-1} in Operation

disconnect the graph is retained under contraction (possibly resulting in even fewer vertices capable of disconnecting the graph), and must ultimately cause failure because contraction creates no more edge-disjoint paths than previously existed.

Thus $K\text{-CONNECTED}^{-1}$ is correct and $K\text{-CONNECTED}$ is complete.

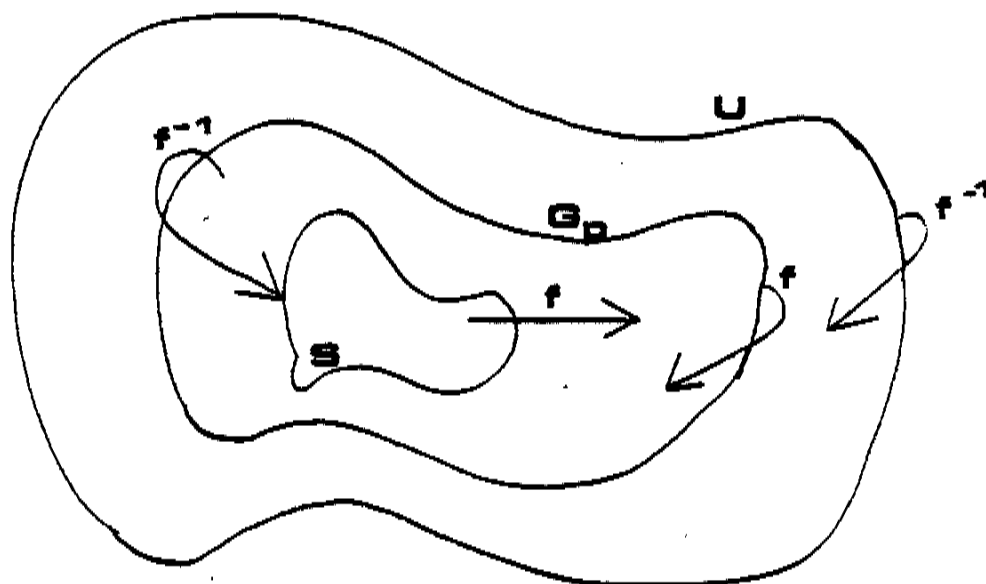


Figure 3-9: The Behavior of f and f^{-1} on the Set of All Graphs

chapter and the next, we strive to work in the simplest floor possible. Because our selection languages Σ are reasonably limited (e.g., there is no notion of a path before Σ_g), such construction may require considerable ingenuity.

3.7.1. Acyclic Graphs

A *walk* of a graph G is an alternating sequence of vertices and edges.

$v_1, v_1, v_2, v_2, v_3, v_3, \dots, v_{k-1}, v_{k-1}, v_k, v_k$ beginning and ending with vertices, in which each edge is incident with the two vertices immediately preceding and following it. (We will use the abbreviated form $v_1 v_2 \dots v_k$.) A walk is *closed* if $v_1 = v_k$, otherwise it is *open*. A *cycle* is a closed walk on k vertices, all distinct, with $k \geq 3$. We will describe such a cycle as $C_{v_1 v_2 \dots v_k}$. An arbitrary cycle on k vertices is written as simply C_k . A graph is *acyclic* if it contains no cycles, i.e., every walk is open. Several examples of acyclic graphs appear in Figure 3-10.

The R-property ACYCLIC is

$$(B_{xy} + A_{zz})^*(\langle V, \phi \rangle) \text{ where } x \in V, y \in V$$

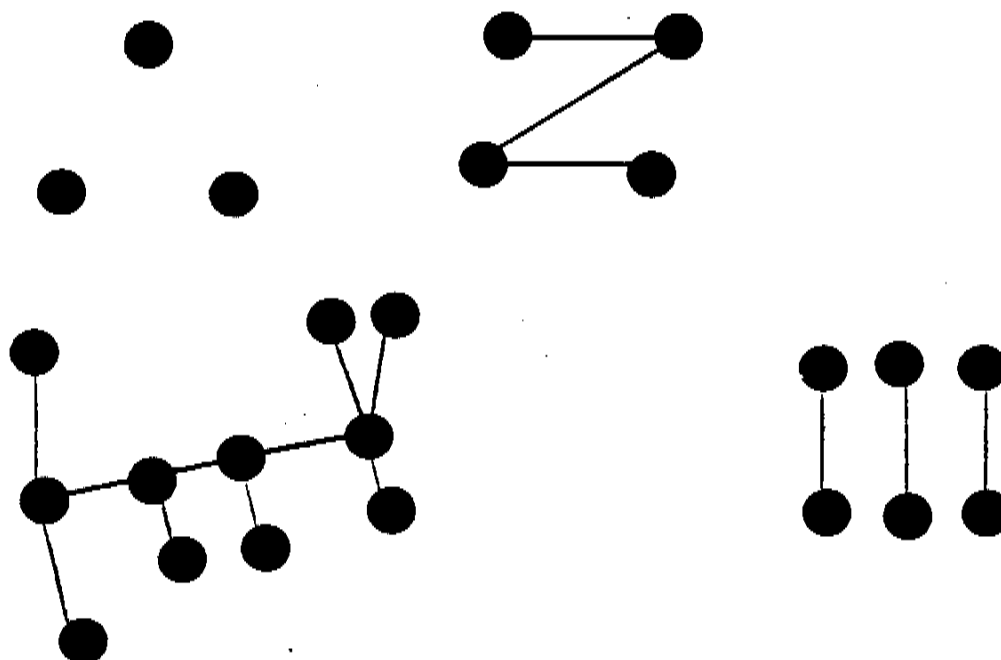


Figure 3-10: Some Acyclic Graphs

The seed set is intended *not* to include $\langle \phi, \phi \rangle$. Figure 3-11 shows the iterative steps in a sample run of ACYCLIC.

Since precisely all graphs of the form $\langle V, \phi \rangle$ other than $\langle \phi, \phi \rangle$ are identified in L_1 by $E = 0$, and since $B_{xy} = A_{xy}A_x$, we have a floor for acyclic graphs of $\langle P_1, L_1, \Sigma_1 \rangle$.

Clearly ACYCLIC is correct: the only edges it adds are loops (which do not occur as part of cycles or qualify as cycles) or edges from a vertex in the set V to one previously outside and of degree zero which has just been created. Thus no edge can, by its addition, complete a cycle. (We observe that, for the last edge in the construction of a cycle, the vertices involved must already both be of degree at least one.) Loops may be added at any time. A loopfree version is

$$B_{xy}^*(\langle V, \phi \rangle) \text{ where } x \in V, y \in V$$

The inverse of ACYCLIC is computed from:

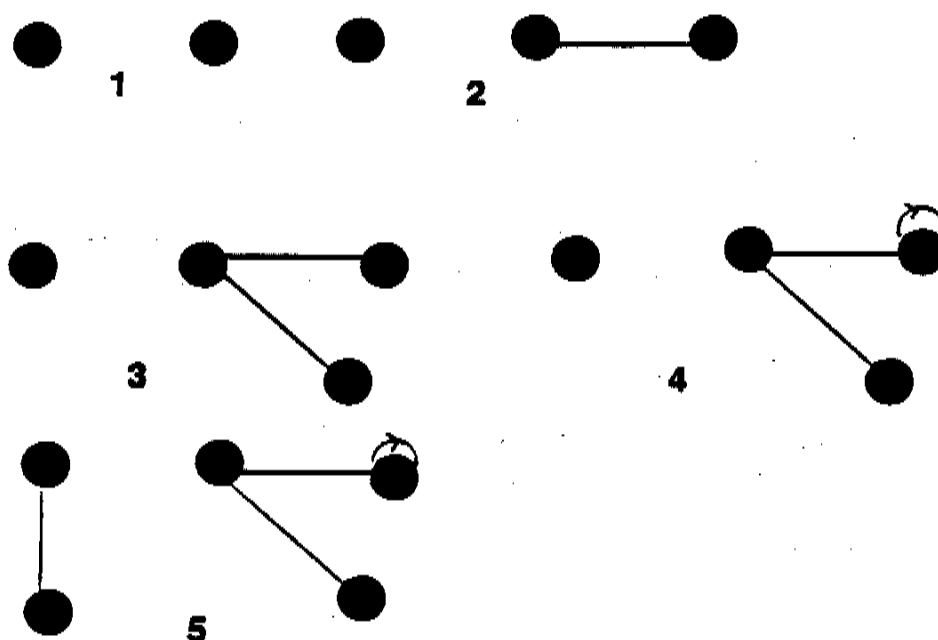


Figure 3-11: A Sample Run of ACYCLIC

$$\begin{aligned}
 f^{-1} &= (B_{xy} + A_{zz})^{-1} \\
 &= B_{xy}^{-1} + A_{zz}^{-1} \\
 &= (A_{xy}A_y)^{-1} + A_{zz}^{-1} \\
 &= A_y^{-1}A_{xy}^{-1} + A_{zz}^{-1} \\
 &= D_y D_{xy} + D_{zz} \\
 \sigma_{pre} &= x \in V, y \in V \\
 &\quad z \in V, zz \in E \\
 \sigma^{-1} &= x, y \in V, xy \in E, d(y) = 1 \\
 &\quad z \in V, zz \in E
 \end{aligned}$$

There is a shift in the floor to $\langle P_1, L_1, \Sigma_3 \rangle$. In Figure 3-12 we show $ACYCLIC^{-1}$ operating on a graph $G \in G_p$ and a graph $G \in G_p$. In order to show that an inverse is correct, we must demonstrate that it behaves properly both on $G \in G_p$ and on $G \in G_p$. If $G \in G_p$, p^{-1} will detach and delete only vertices of degree one. Thus any walk will be decimated from its endpoints inward, and any acyclic graph will be reduced ultimately to a set of isolated vertices $\langle V, \emptyset \rangle$, with one

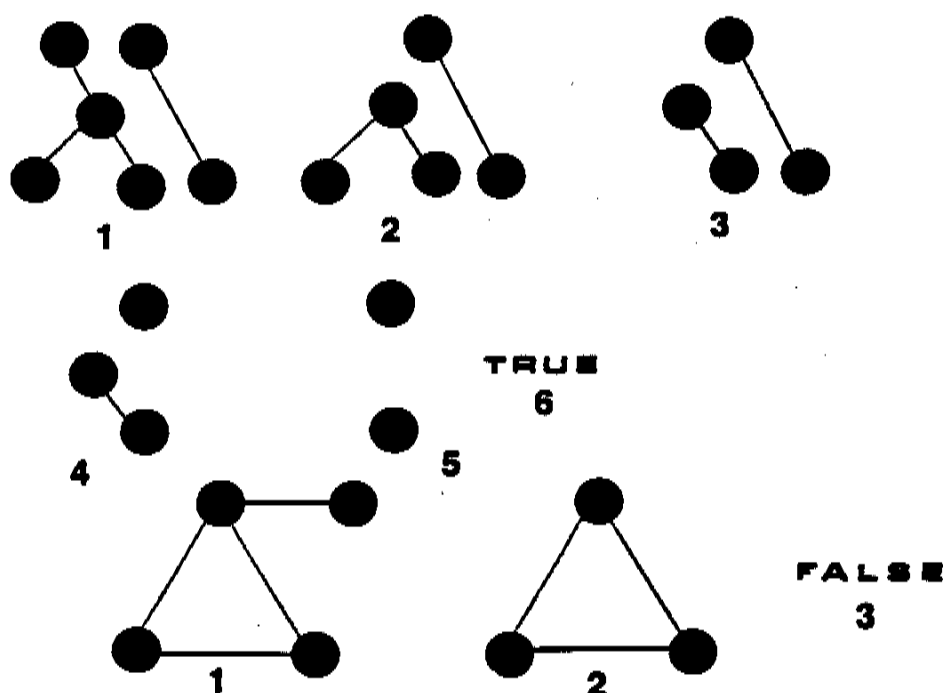


Figure 3-12: $ACYCLIC^{-1}$ in Operation

element remaining in V for each mutually accessible set of vertices originally in V . If $G \in G_p$, p^{-1} will decimate any acyclic protrusions. What remains will be a graph in which every vertex has degree at least two. Such a graph must contain a cycle, and σ^{-1} will not be applicable on it. Thus p^{-1} is correct and p is complete.

3.7.2. Trees

A graph is *connected* if every pair of vertices are joined by a path. A *tree* is a connected acyclic graph. Several examples of trees appear in Figure 3-13. The R-property TREE is

$$(B_{xy} + A_{zz})^*(K_1) \text{ where } x \in V, y \in V, z \in V$$

Figure 3-14 shows the iterative steps in a sample run of TREE.

Since the only graph matching the L_1 characterization $E \cup \bar{1} = 0$ is K_1 , the floor for trees is $\langle P_1, L_1, \bar{\Sigma}_1 \rangle$.

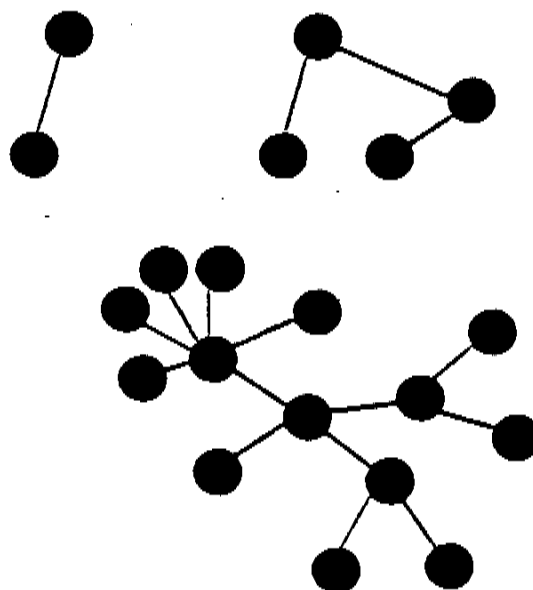


Figure 3-13: Some Trees

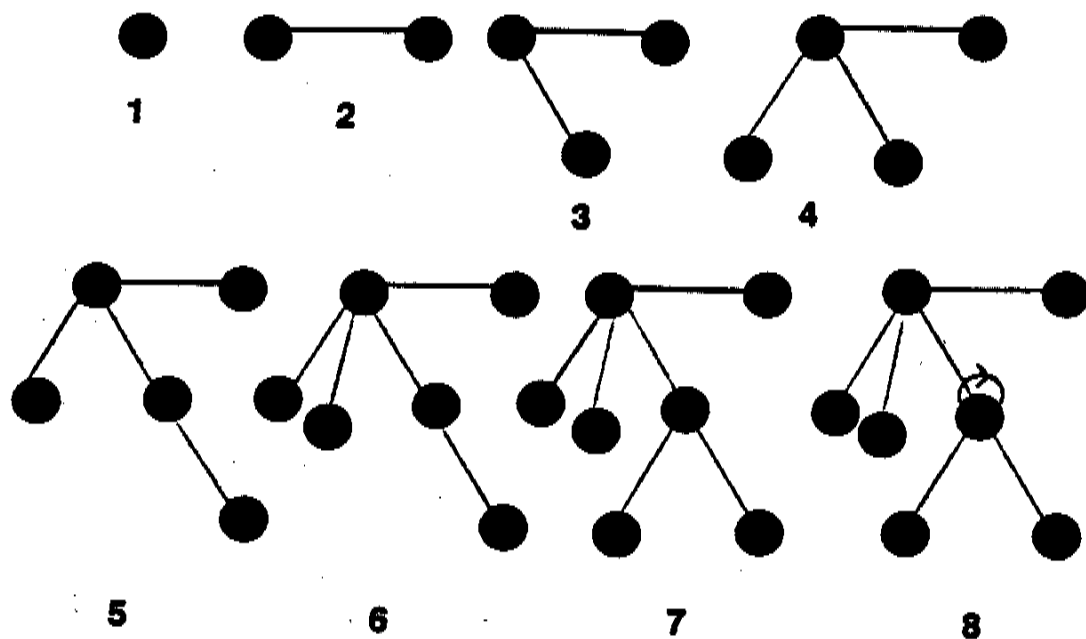


Figure 3-14: A Sample Run of TREE

Clearly, TREE is correct the only edges it adds are loops or edges which

cannot complete a cycle and are part of a single connected component.

The inverse for TREE has the same f^{-1} and σ^{-1} as ACYCLIC, namely

$$f^{-1} = D_y D_{xy} + D_{zz}$$

$$s^{-1} = x, y \in V, xy \in E, d(y) = 1$$

$$z \in V, zz \in E$$

Again there is a shift in the floor to $\langle P_1, L_1, \Sigma_3 \rangle$. This kinship is not accidental and will be discussed at length in Chapter 4. Figure 3-15 shows $TREE^{-1}$ operating on a graph $G \in G_p$ and a graph $G \notin G_p$.

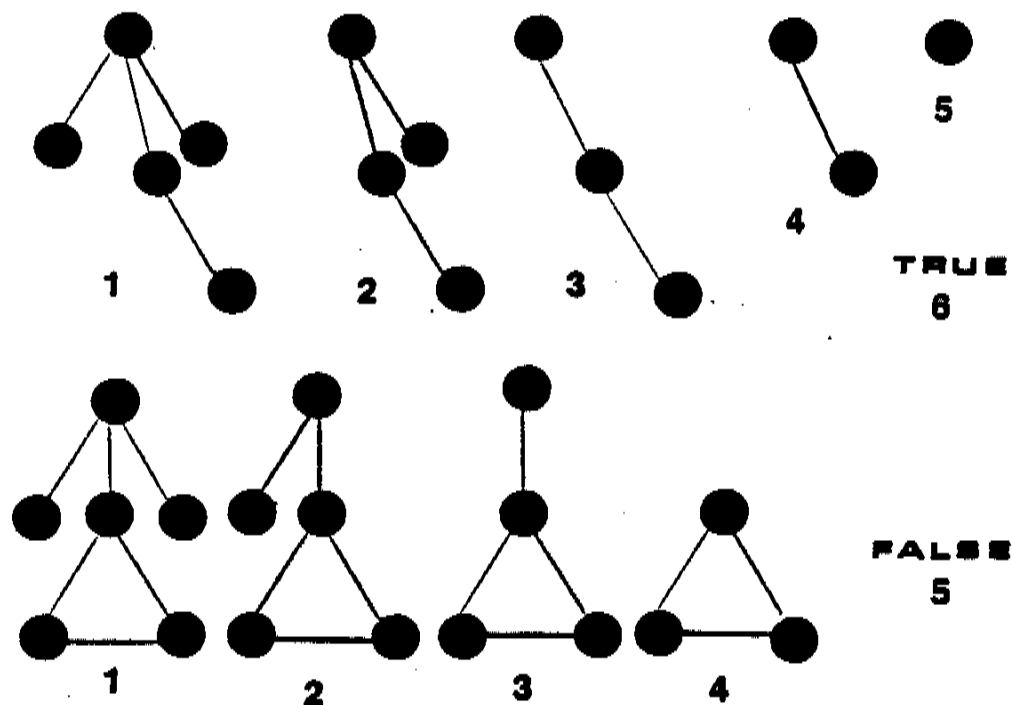


Figure 3-15: $TREE^{-1}$ in Operation

$TREE^{-1}$ reduces any tree to, ultimately, a single, isolated vertex isomorphic to K_1 . $TREE^{-1}$ on a graph which is not a tree will remove all tree-like protuberances, and then σ^{-1} will fail, leaving a graph composed of disconnected and/or cyclic graphs, which is not isomorphic to K_1 . Thus $TREE^{-1}$ is correct and TREE is complete.

A loopfree form of TREE is

$$B_{xy}^*(K_1) \text{ where } x \in V, y \in V$$

Although loops are useful and even significant in the formulation of certain properties, their presence in a graph normally does not add or detract from the presence of that property. Thus we will pay our dues to loopfree once, in the next subsection, and then write all our properties in a format which does not add extraneous loops.

3.7.3. Loopfree Graphs

A graph is *loopfree* if it contains no edge from a vertex to itself. Several examples of loopfree graphs appear in Figure 3-16.

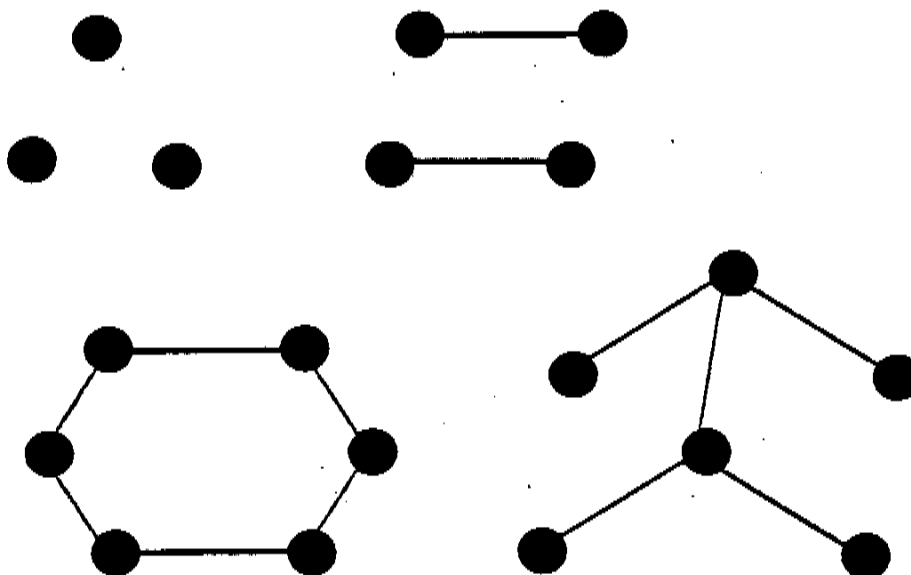


Figure 3-16: Some Loopfree Graphs

The R-property LOOPFREE is

$$(A_x + A_{yz}^*)(K_1) \text{ where } y, z \in V, y \neq z$$

Figure 3-17 shows the iterative steps in a sample run of LOOPFREE. The floor for loopfree graphs is $\langle P_1, L_1, \Sigma_2 \rangle$.

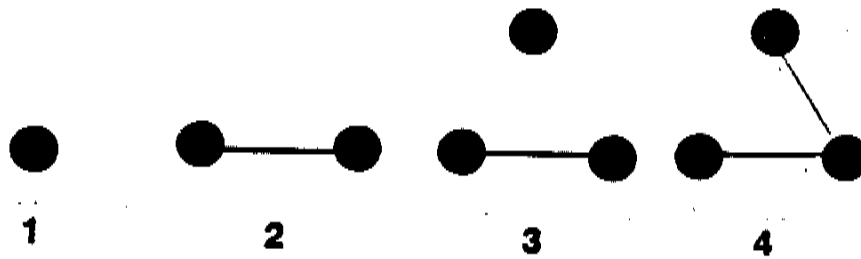


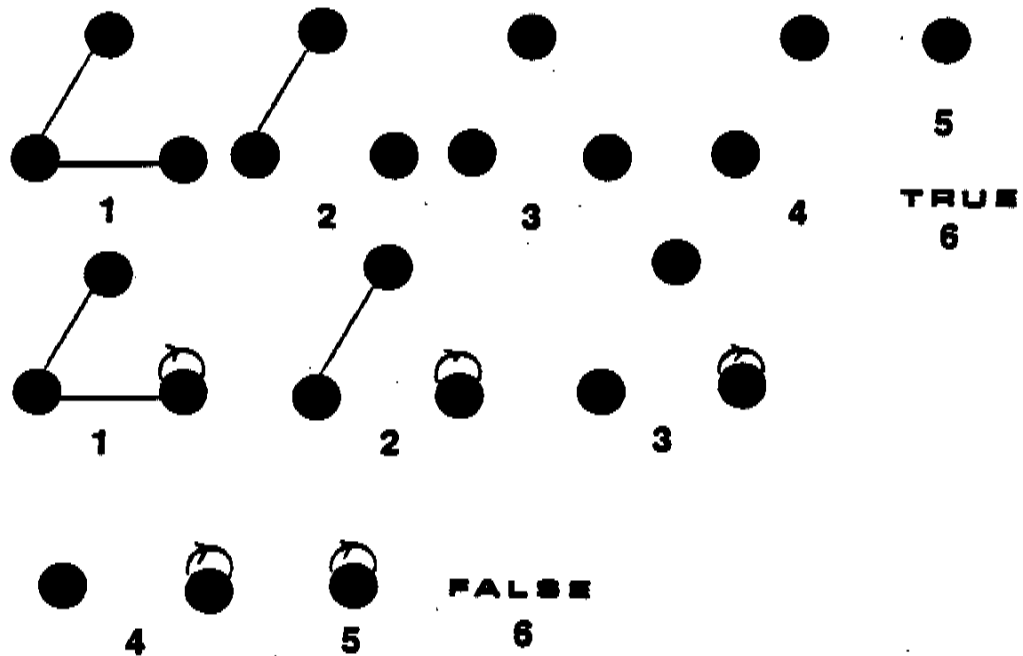
Figure 3-17: A Sample Run of LOOPFREE

LOOPFREE is correct; it adds vertices and non-loop edges only. The inverse is computed as:

$$\begin{aligned}
 f^{-1} &= (A_x + A_{yz})^{-1} \\
 &\approx A_x^{-1} + A_{yz}^{-1} \\
 &\approx D_x + D_{yz} \\
 \sigma_{pre}^{-1} &= x \in V \\
 &\quad y, z \in V, yz \in E, y \neq z \\
 \sigma^{-1} &= x \in V, d(x) = 0 \\
 &\quad y, z \in V, yz \in E, y \neq z
 \end{aligned}$$

There is a shift in the floor to $\langle P_1, L_1, \Sigma_3 \rangle$. Figure 3-18 shows LOOPFREE^{-1} operating on a graph $G \in G_p$ and a graph $G \in G_p$.

LOOPFREE^{-1} deletes all isolated vertices and non-loop edges of a loopfree graph until a graph isomorphic to K_1 is achieved. A graph with a loop will, after no more than $m + n$ applications of f^{-1} , have $E \cap 1 \neq \emptyset$ and $E \cap \bar{1} \neq \emptyset$, and LOOPFREE^{-1} will return FALSE. Thus LOOPFREE^{-1} is correct and LOOPFREE is complete.

Figure 3-18: LOOPFREE^{-1} in Operation

3.7.4. Chains

A *chain* is a graph consisting of a single open path on at least two vertices. The length of a chain is one less than the number of its vertices, i.e., $n - 1$. Several examples of chains appear in Figure 3-19.

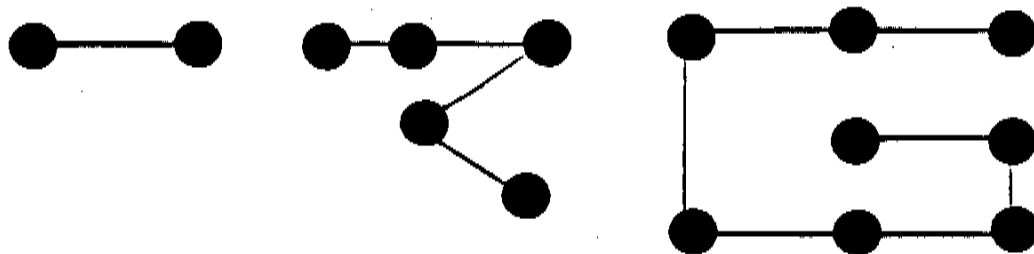


Figure 3-19: Some Chains

The R-property CHAIN is

$$S_{xy}^*(K_2) \text{ where } x, y \in V, v \notin V, xy \in E$$

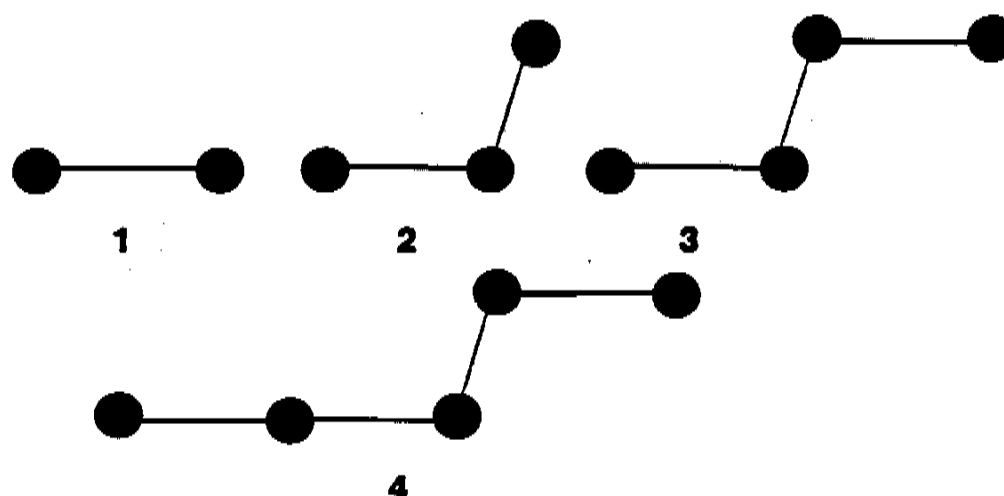


Figure 3-20: A Sample Run of CHAIN

Figure 3-20 shows the iterative steps in a sample run of CHAIN.

$S_{xvy} = D_{xy} A_{xv} A_{vy} A_v$ so P_2 is sufficient for CHAIN. In L_1 the characterization of K_2 is $\underline{E} \cap \underline{1} = 0$ and $\underline{E} \cap \underline{1} = 0$. This is also the characterization of any other loopfree complete graph; so L_1 is inadequate to precisely describe $\{K_2\}$. In L_2 , the characterization of K_2 is the same as in K_1 . In L_3 , however, K_2 has the following characterization:

$$\underline{E} \cap \underline{1} = 0$$

$$\underline{E} \cap \underline{1} = 0$$

$$|\underline{1}| < |\underline{E}|$$

Since $|\underline{1}| = n(n-1)/2$ and $|\underline{E}| = n$ we have $n(n-1)/2 < n$ so, given that $n \geq 1$, we must have $n = 1$ or $n = 2$. However, the only other graphs satisfying $n = 1$ and $n = 2$ have characterizations distinct from that of K_2 . Thus L_3 is adequate to describe $\{K_2\}$, as is L_{1n} in which K_2 is

$$\underline{E} \cap \underline{1} = \emptyset$$

$$\underline{E} \cap \underline{1} = \emptyset$$

$$n = 2$$

Therefore the floors for CHAIN are $\langle P_2, L_{1n}, \Sigma_1 \rangle$ and $\langle P_2, L_3, \Sigma_1 \rangle$.

CHAIN is correct: it replaces any edge xy with a chain of two edges, xv and vy , increasing the chain by one edge, either in the center or on the end, in each iteration. Because K_2 contains no loops, $x \neq y$ is unnecessary.

The inverse is computed by

$$\begin{aligned} f^{-1} &= S_{xy}^{-1} \\ &= (D_{xy} A_{xy} A_{xy}^{-1} A_{xy})^{-1} \\ &= A_v^{-1} A_{vy}^{-1} A_{xy}^{-1} D_{xy}^{-1} \\ &= D_v D_{vy} D_{xy} A_{xy} \\ \sigma_{pre} = \sigma &= x, y \in V, v \in V, xy \in E \\ \sigma^{-1} &= x, y, v \in V, xv, yv \in E, xy \in E, d(v) = 2 \end{aligned}$$

There is a shift in the floors to $\langle P_2, L_{1n}, \Sigma_3 \rangle$ and $\langle P_2, L_3, \Sigma_3 \rangle$. Figure 3-21 shows $CHAIN^{-1}$ operating on a graph $G \in G_p$ and a graph $G \in G_p$.

$CHAIN^{-1}$ contracts chains to length one and simple cycles to C_3 without collapsing them. Any vertices of degree greater than two will always remain in a graph under application of $CHAIN^{-1}$. Thus $CHAIN^{-1}$ is correct and CHAIN is complete.

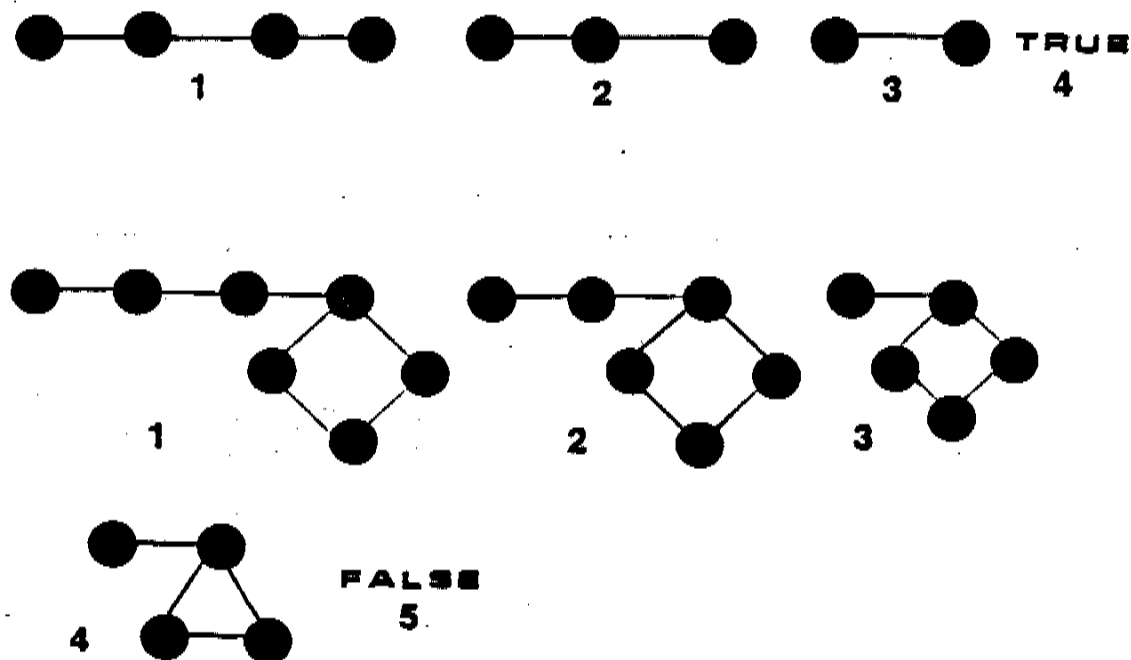
Here is another formulation, $CHAIN_2$, for the property of being a chair:

$$B_{xy}^*(K_2) \text{ where } x \in V, y \in V, d(x) = 1$$

Figure 3-20 could also be the iterative steps in a sample run of $CHAIN_2$.

The floors are $\langle P_1, L_{1n}, \Sigma_3 \rangle$ and $\langle P_1, L_3, \Sigma_3 \rangle$. $CHAIN_2$ is correct: it grows a chain by branching from an endpoint, adding one terminal link on each iteration. $CHAIN_2^{-1}$ is computable from:

$$f^{-1} = B_{xy}^{-1}$$

Figure 3-21: CHAIN^{-1} in Operation

$$\begin{aligned}
 &= D_y D_{xy} \\
 \sigma_{\text{pre}} &= x \in V, y \in V, d(x) = 1 \\
 \sigma^{-1} &= x, y \in V, xy \in E, d(y) = 1, d(x) = 2
 \end{aligned}$$

The floors remain constant. Figure 3-22 shows CHAIN_2^{-1} operating on a graph $G \in G_p$ and a graph $G \in G_p$.

CHAIN_2^{-1} removes terminal edges (to a vertex of degree one) in a chain until the chain is of length one. On a non-chain, CHAIN_2^{-1} retains simple cycles and vertices of degree greater than two. Thus CHAIN_2^{-1} is correct and CHAIN_2 is complete. We have shown two formulations for the property of being a chain, one with a lower floor than the other.

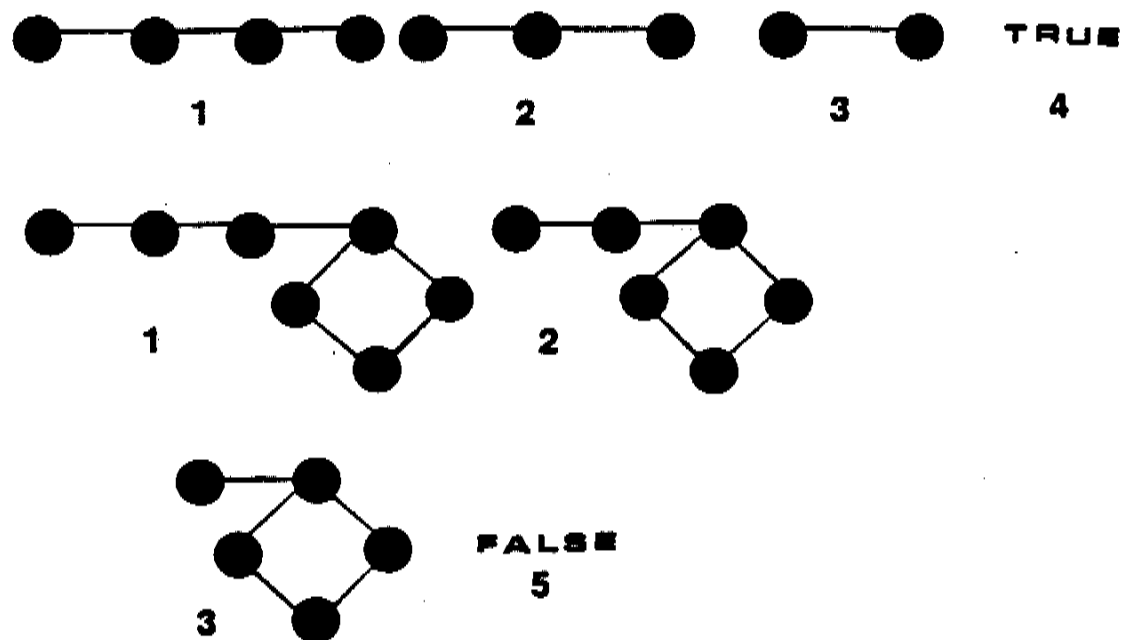


Figure 3-22: CHAIN_2^{-1} in Operation

3.7.5. Cycles

Several examples of cycles appear in Figure 3-23.

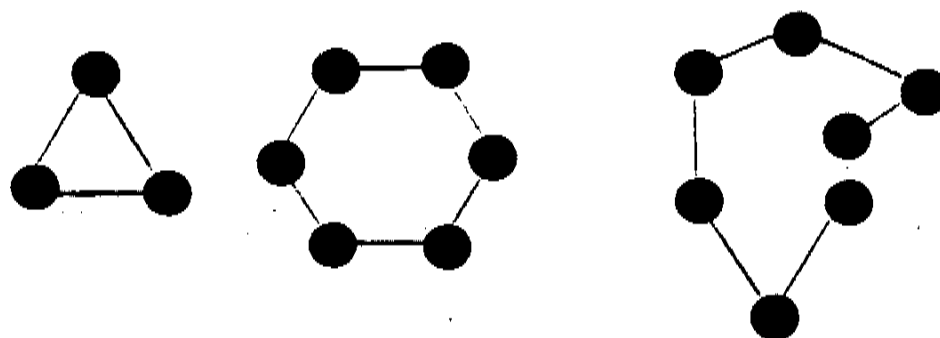


Figure 3-23: Some Cycles

The R-property CYCLE is

$$S_{xy}^*(K_3) \text{ where } x, y \in V, v \in V, xy \in E.$$

Figure 3-24 shows the iterative steps in a sample run of CYCLE.

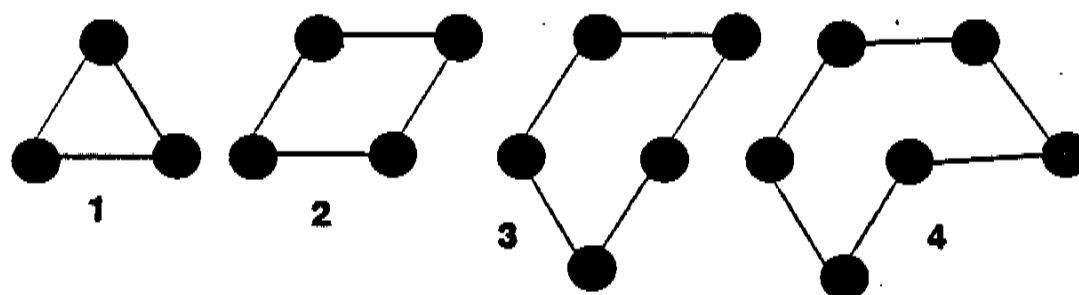


Figure 3-24: A Sample Run of CYCLE

In L_1 the characterization of K_3 is the same as K_2 . In L_2 , however, $\{K_3\}$ is uniquely defined by:

$$\underline{E} \cap 1 \neq 0$$

$$E \cap 1 = 0$$

$$|E \cap 1| = |E \cap 1|$$

It is also possible to reach K_3 in L_{1n} as $\underline{E} \cap 1 = 0$, $E \cap 1 = 0$ and $n = 3$. Thus the floors of CYCLE are $\langle P_2, L_2, \Sigma_1 \rangle$ and $\langle P_2, L_{1n}, \Sigma_1 \rangle$.

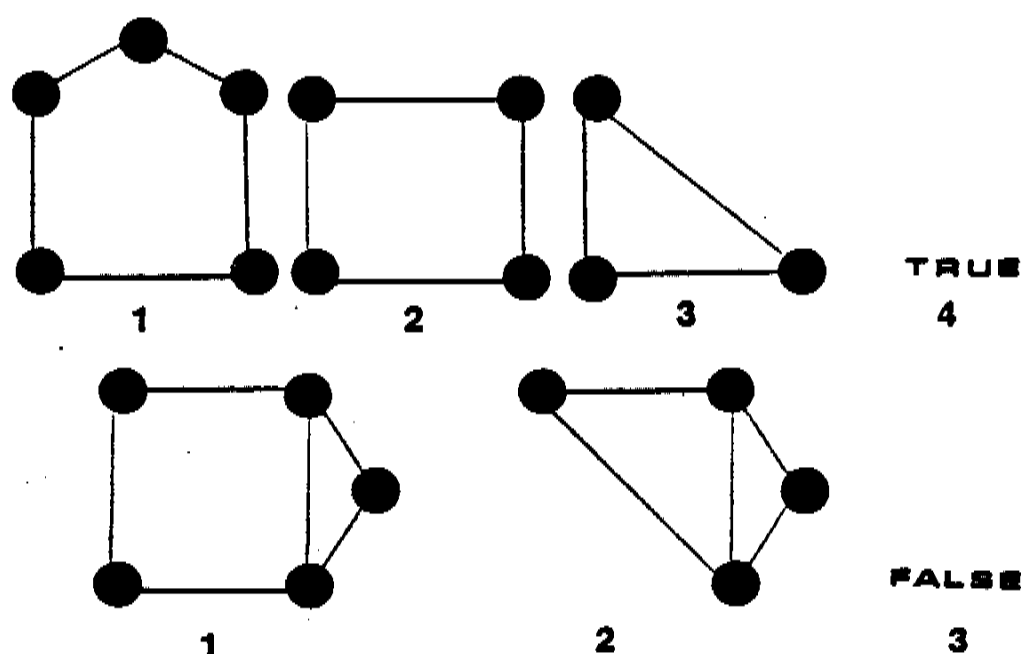
CYCLE is correct; on each iteration it replaces one edge in a cycle with a chain of length two. The inverse for CYCLE has the same f^{-1} and σ^{-1} as CHAIN, namely

$$f^{-1} = D_v D_{vy} D_{xv} A_{xy}$$

$$\sigma^{-1} = x, y, v \in V, xv, yv \in E, xy \notin E, d(v) = 2$$

Again, this is not accidental. There is a shift in the floors to $\langle P_2, L_2, \Sigma_3 \rangle$ and $\langle P_2, L_{1n}, \Sigma_3 \rangle$. Figure 3-25 shows CYCLE^{-1} operating on a graph $G \in G_p$ and a graph $G \in G_p$.

CYCLE^{-1} will contract any simple cycle until it is isomorphic to K_3 . Any non-cycle will have its chain-like portions contracted by CYCLE^{-1} to length one and its simple cycles to C_3 , leaving the remaining graph untouched. Thus CYCLE^{-1} is correct and CYCLE is complete.

Figure 3-25: $CYCLE^{-1}$ in Operation

3.7.6. Stars

A *bipartite* graph G is a graph whose vertex set V can be partitioned into sets V_1 and V_2 such that every edge of G is between a vertex in V_1 and a vertex in V_2 . If $E = V_1 \times V_2$ then G is a *complete bipartite* graph. If $|V_1| = a$, $|V_2| = b$, the complete bipartite graph on $V = V_1 \cup V_2$, where $V_1 \cap V_2 = \emptyset$, is denoted $K_{a,b}$. A *star* is a complete bipartite graph $K_{1,n}$ for $n \geq 3$. Several examples of stars appear in Figure 3-26. The R-property STAR is

$$B_{xy}^*(K_{1,3}) \text{ where } x \in V, y \in V, d(x) = \max$$

Figure 3-27 shows the iterative steps in a sample run of STAR.

In L_1 , $K_{1,3}$ is characterized by $E \cap 1 = 0$, but so are all loopfree graphs. In L_2 , $K_{1,3}$ is characterized by $E \cap 1 = 0$ and $|E| = |\underline{E}|$, but so is any graph with half its possible edges and no loops. In L_3 , $K_{1,3}$ has the same signature as a chain of length three. Here is our first example of a seed graph which defies definition in any of our preferred languages. The floor for star graphs is $\langle P_1, L_0, \Sigma_4 \rangle$. It is

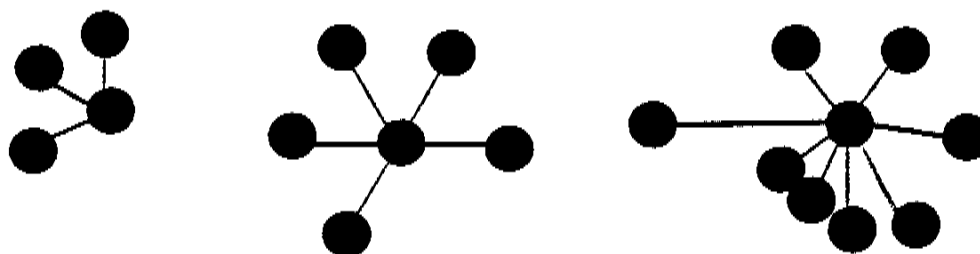


Figure 3-26: Some Star Graphs

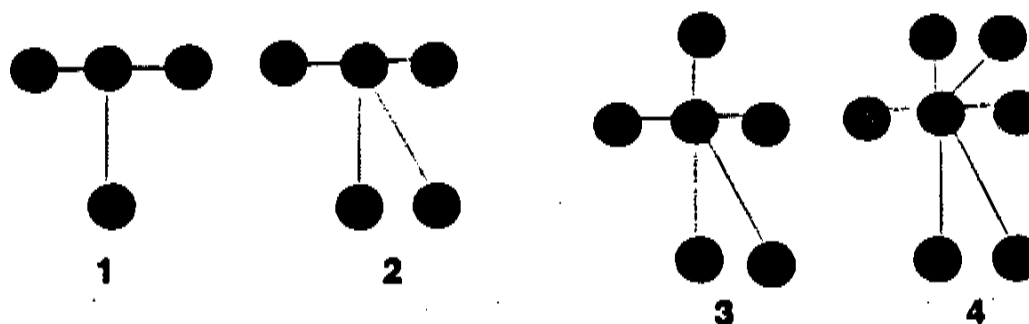


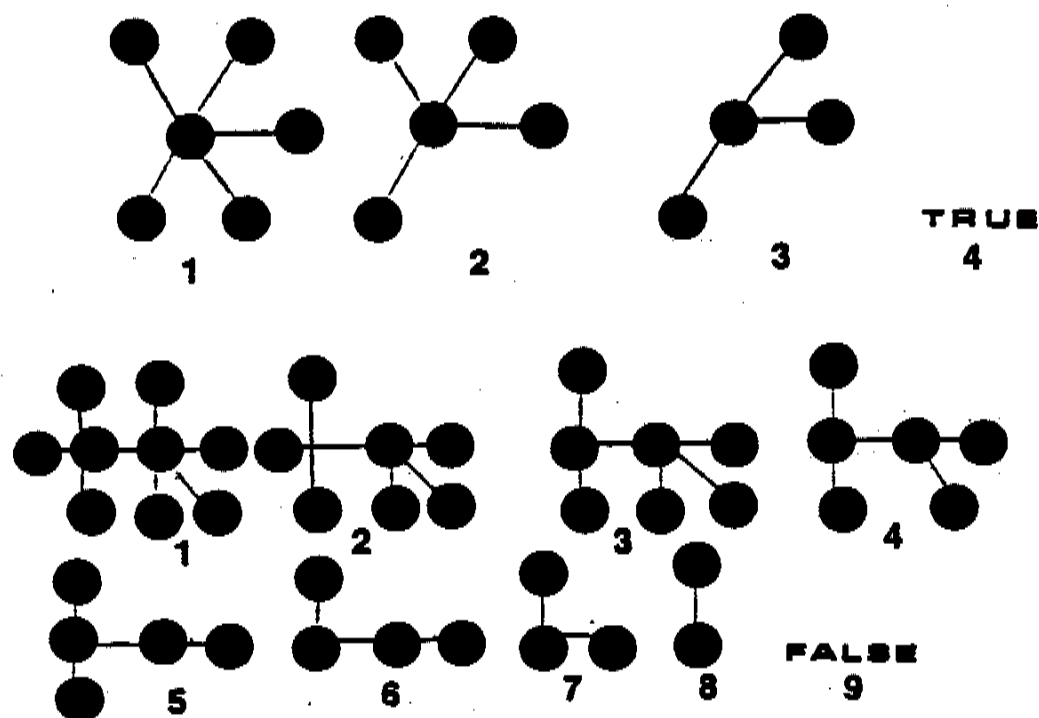
Figure 3-27: A Sample Run of STAR

possible to define star so that $K_{1,2}$ or $K_{1,1}$ is considered a star. This would require some σ statement about the maximum degree vertex being unique or about all vertices other than the one of maximum degree being of degree one, neither of which is available in any Σ postulated thus far. This is an example of the potential tradeoff between L and Σ .

We call the vertex of maximum degree in a star its *center*. STAR adds one *spoke* (degree one vertex and edge from the center to it) at a time. STAR is correct. The inverse is computed from:

$$\begin{aligned}
 f^{-1} &= B_{xy}^{-1} \\
 &= D_y D_{xy} \\
 \sigma_{pre} &= x \in V, y \in V, d(x) = \max \\
 \sigma^{-1} &= x, y \in V, d(x) = \max, d(y) = 1
 \end{aligned}$$

The floor shifts to $\langle P_2, L_0, \Sigma_4 \rangle$. Figure 3-28 shows $STAR^{-1}$ operating on a

Figure 3-28: $STAR^{-1}$ in Operation

graph $G \in G_p$ and a graph $G \in G_p$.

On a star graph, $STAR^{-1}$ will delete the spokes one at a time until arriving at $K_{1,3}$. On a non-star graph, $STAR^{-1}$ will repeatedly delete spoke-like constructs. A chain will contract to K_2 under $STAR^{-1}$ and thereby fail. Thus $STAR^{-1}$ is correct and $STAR$ is complete.

3.7.7. Wheels

A *wheel* $W_{1,n}$ is a graph in which $n \geq 3$ and

$$V = \{v, v_1, v_2, \dots, v_n\}$$

$$E = \{v_i v_{i+1} \mid i = 1, 2, \dots, n-1\} \cup \{v_1 v_n\} \cup \{v v_i \mid i = 1, 2, \dots, n\}$$

A wheel is composed of a *rim* ($C_{v_1 v_2 \dots v_n}$) and an additional vertex (the *hub* v) which is adjacent to all the other vertices. Several examples of wheels appear in Figure 3-29. The R-property WHEEL is

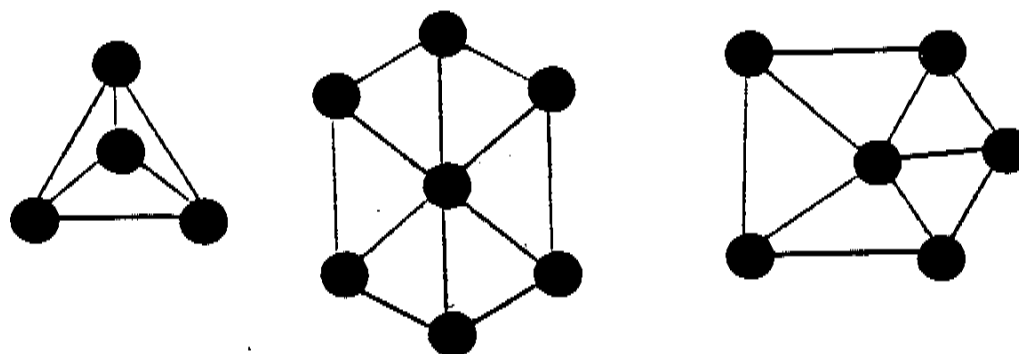


Figure 3-29: Some Wheels

$(A_{zv} S_{xvy})^*(K_4)$ where distinct $x, y, z \in V$, $v \in V$, $xy \in E$, $d(z) = \max$

Note that K_4 is merely another notation for $W_{1,3}$. Figure 3-30 shows the iterative steps in a sample run of WHEEL.

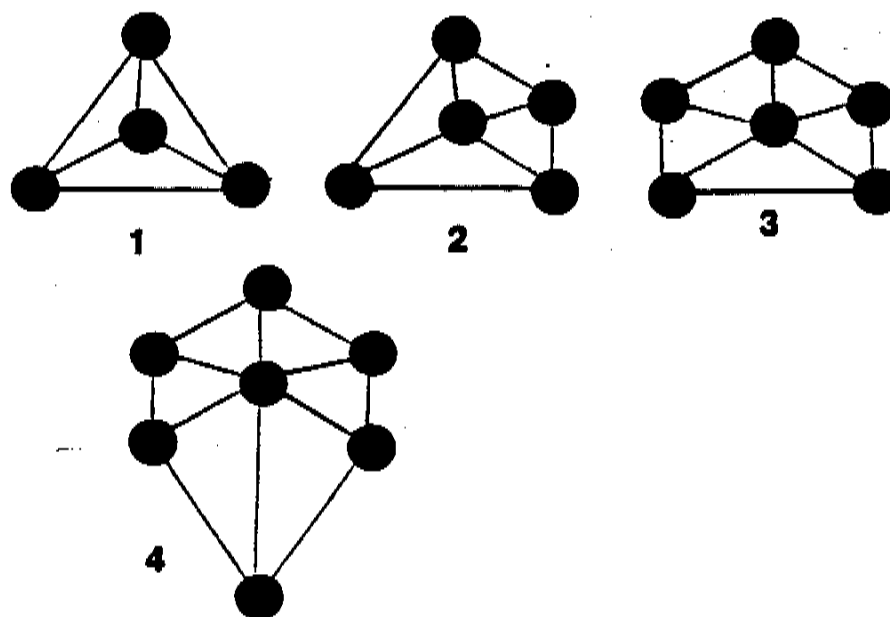


Figure 3-30: A Sample Run of WHEEL

We use "distinct x, y, z " here as an abbreviation for the $\sum_2$ notation " $x \neq y$, $x \neq z$, $y \neq z$ ". In L_1 , K_4 has the same characterization as any other loopfree complete graph. This is also true in L_2 and L_3 . In L_{1n} , however, $\{K_4\}$ is precisely specified by:

$$\sum_{i=1}^n 1 = 0$$

$$\sum_{i=1}^n 1 = 0$$

$$n = 4$$

Thus the floor for wheels is $\langle P_2, L_{1n}, \Sigma_4 \rangle$.

WHEEL is correct; it replaces any rim edge with a chain of two edges, connecting the new vertex to the hub. Because K_4 contains no loops, $x \neq y$ is unnecessary.

The inverse is computed by:

$$\begin{aligned} f^{-1} &= (A_{zv} S_{xvy})^{-1} \\ &= S_{xvy}^{-1} A_{zv}^{-1} \\ &= D_v D_{vy} D_{xv} A_{xy} D_{zv} \end{aligned}$$

$$\sigma_{pre} = \sigma \quad = \text{distinct } x, y, z \in V, v \in V, xy \in E, d(z) = \max$$

$$\begin{aligned} \sigma^{-1} &= \text{distinct } v, x, y, z \in V, xv, vy \in E, xy \in E, d(z) = \max, \\ &\quad d(v) = 3 \end{aligned}$$

The floor remains constant. Figure 3-31 shows WHEEL^{-1} operating on a graph $G \in G_p$ and a graph $G \in G_p$.

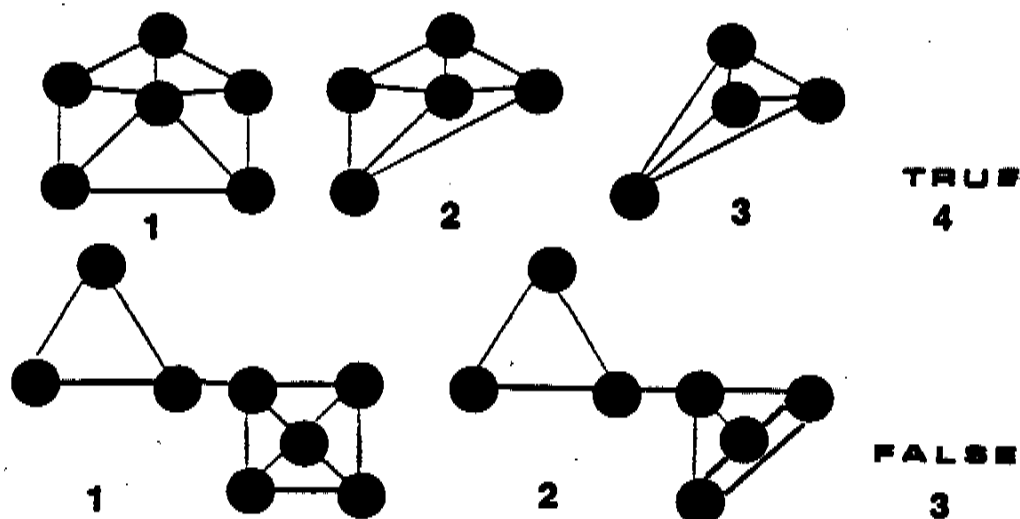


Figure 3-31: WHEEL^{-1} in Operation

WHEEL^{-1} contracts the rim of a wheel until the graph is isomorphic to K_4 . On a non-wheel, any vertices of degree other than 3 or $n - 1$ will remain untouched, with the wheel-like portions collapsing into K_4 . Thus WHEEL^{-1} is correct and WHEEL is complete.

3.7.8. Complete Graphs

A graph is *complete* if and only if $E = \{xy \mid x, y \in V, x \neq y\}$. The complete graph on k vertices is denoted K_k . Several examples of complete graphs appear in Figure 3-32.

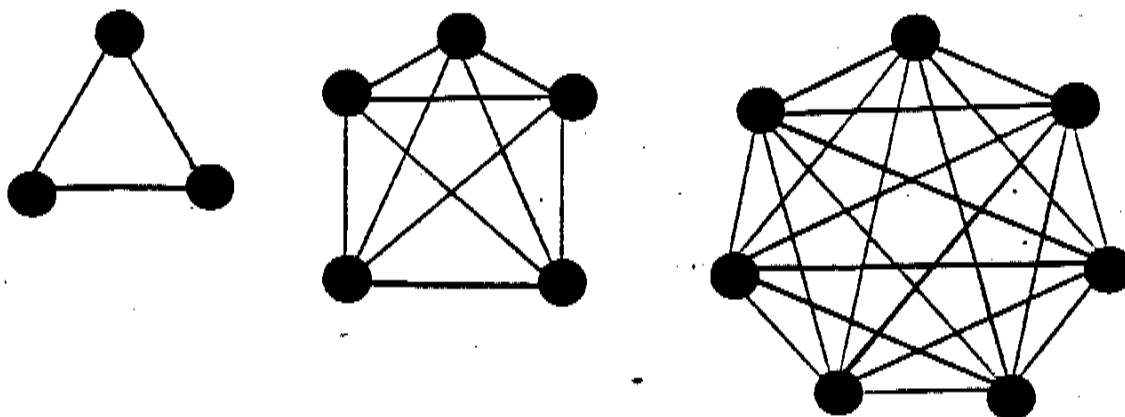


Figure 3-32: Some Complete Graphs

The R-property COMPLETE is

$$F_x^*(K_n) \text{ where } x \in V, \text{ distinct } v_1, v_2, \dots, v_n \in V, |\{v_1, v_2, \dots, v_n\}| = |V|$$

Figure 3-33 shows the iterative steps in a sample run of COMPLETE. Since $F_x =$

$A_{xv_1} A_{xv_2} \dots A_{xv_n} A_x$, the floor for complete graphs is $\langle P_1, L_1, \Sigma_g \rangle$.

COMPLETE is correct: it connects a new vertex to every vertex currently in the graph.

The inverse is computed by:

$$f^{-1} = F_x^{-1}$$

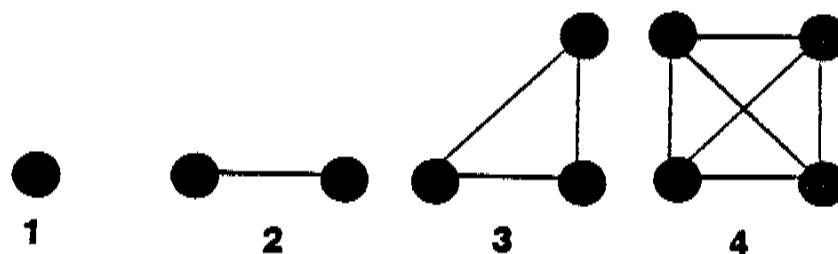


Figure 3-33: A Sample Run of COMPLETE

$$\begin{aligned}
 &= (A_{xv_1} A_{xv_2} \dots A_{xv_n} A_x)^{-1} \\
 &= A_x^{-1} A_{xv_n}^{-1} A_{xv_{n-1}}^{-1} \dots A_{xv_1}^{-1} \\
 &= D_x D_{xv_n} D_{xv_{n-1}} \dots D_{xv_1} \\
 \sigma_{\text{pre}} = \sigma &= x \in V, \text{ distinct } v_1, v_2, \dots, v_n \in V, |\{v_1, v_2, \dots, v_n\}| = |V| \\
 \sigma^{-1} &= \text{distinct } x, v_1, v_2, \dots, v_n \in V, |\{x, v_1, v_2, \dots, v_n\}| = |V|, d(x) = n
 \end{aligned}$$

The floor shifts to $\langle P_2, L_1, \Sigma_5 \rangle$. Figure 3-34 shows COMPLETE^{-1} operating on a graph $G \in G_p$ and a graph $G \in G_p$.

COMPLETE^{-1} deletes only fully connected vertices from a graph. If G initially has n vertices, COMPLETE^{-1} must delete $\sum_{i=1}^{n-1} i = n(n-1)/2 = \binom{n}{2}$ distinct edges to be successful; thus G must have been complete. COMPLETE^{-1} is correct and COMPLETE is complete.

3.7.9. Graphs with an Even Number of Vertices

Several examples of graphs with an even number of vertices appear in Figure 3-35. The R-property EVEN-N is

$$\begin{aligned}
 &(A_{xy} + A_w A_z)^*(E_2) \text{ where } x, y \in V, \\
 &w, z \in V, w \neq z
 \end{aligned}$$

Figure 3-36 shows the iterative steps in a sample run of EVEN-N. In L_1 , the graph E_2 has the characterization of most edgeless graphs, $E = 0$. In L_2 the characterization remains the same. In L_3 , however, the characterization is $E = 0$ and $\underline{E} \cap 1 < \underline{E} \cap 1$. These properties imply $n(n-1)/2 < n$ which requires $n < 3$. Since

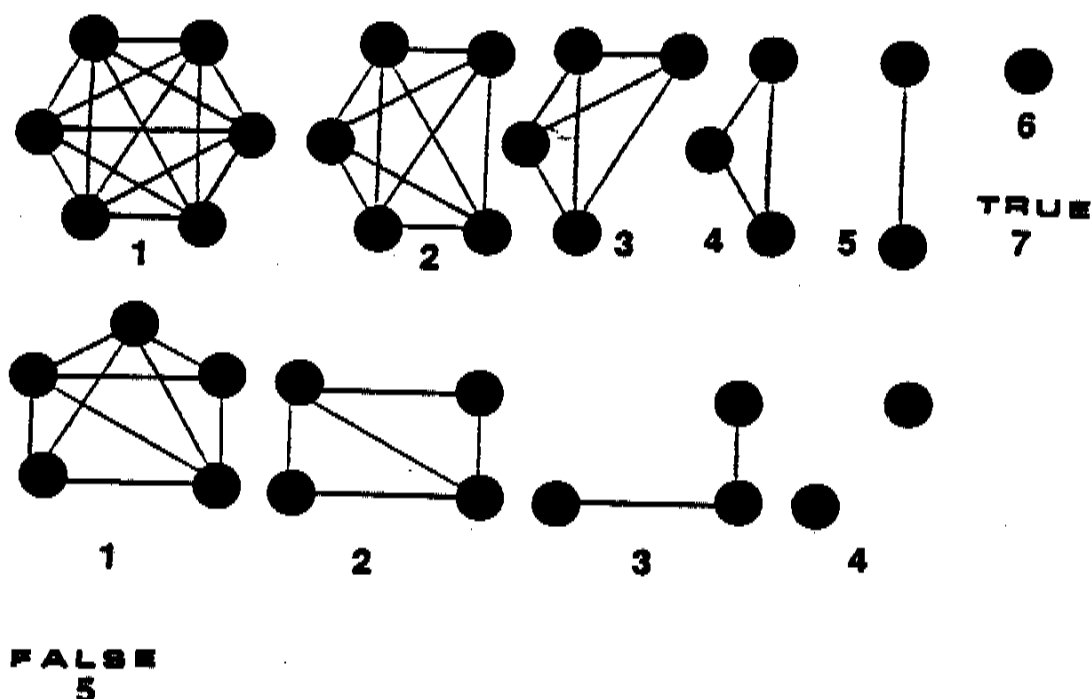
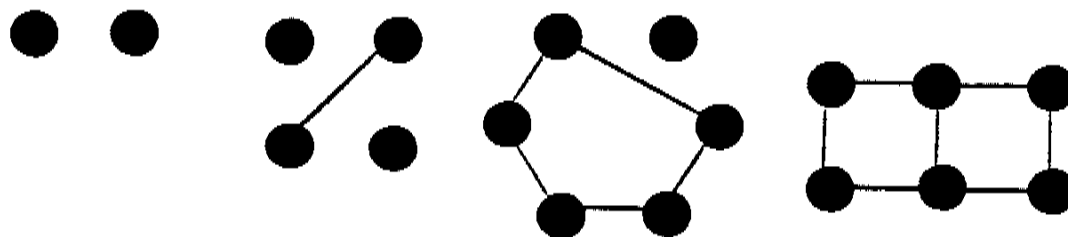
Figure 3-34: COMPLETE^{-1} in Operation

Figure 3-35: Some Graphs with an Even Number of Vertices

all the other graphs for which $n < 3$ have characterizations different from E_2 's, L_3 describes $\{E_2\}$ uniquely. Alternatively, in L_{1n} , E_2 is characterized by $E = 0$ and $n = 2$. The floors for graphs with an even number of vertices are therefore $\langle P_1, L_3, \Sigma_2 \rangle$ and $\langle P_1, L_{1n}, \Sigma_2 \rangle$.

EVEN-N is correct; it adds arbitrary legal edges singly and vertices two at a time.

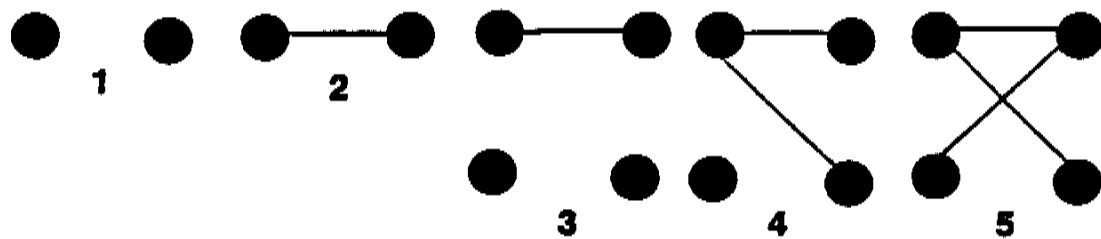
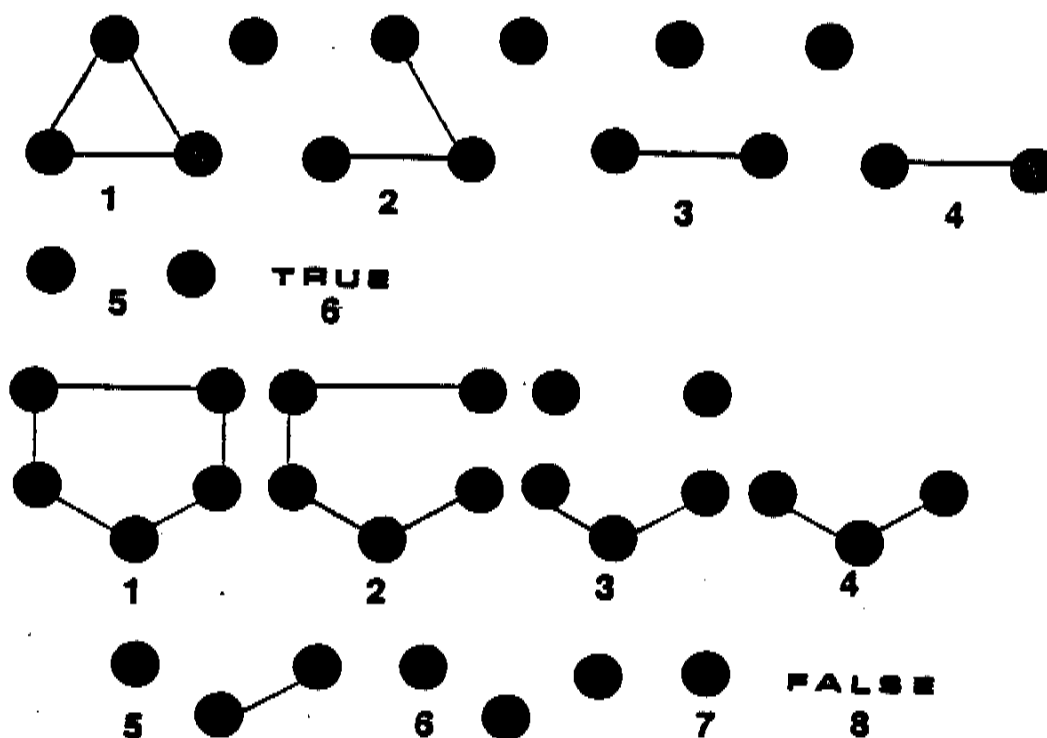


Figure 3-36: A Sample Run of EVEN-N

The inverse is computed by:

$$\begin{aligned}
 f^{-1} &= (A_{xy} + A_w A_z)^{-1} \\
 &= A_{xy}^{-1} + (A_w A_z)^{-1} \\
 &= A_{xy}^{-1} + A_z^{-1} A_w^{-1} \\
 &= D_{xy} + D_z D_w \\
 \sigma_{pre} &= x, y \in V, xy \notin V \\
 &\quad w, z \in V, w \neq z, \\
 \sigma^{-1} &= x, y \in V, xy \in E \\
 &\quad w, z \in V, w \neq z, d(w) = 0, d(z) = 0
 \end{aligned}$$

There is a shift in the floors to $\langle P_2, L_3, \Sigma_3 \rangle$ and $\langle P_2, L_{1n}, \Sigma_3 \rangle$. Figure 3-37 shows EVEN-N^{-1} operating on a graph $G \in G_p$ and a graph $G \notin G_p$. EVEN-N^{-1} deletes the edges of a graph with an even number of vertices and removes the isolated vertices two at a time until the graph is isomorphic to E_2 . A graph with an odd number of vertices will go from E_3 to E_1 and then fail. Thus EVEN-N^{-1} is correct and EVEN-N is complete.

Figure 3-37: $\text{EVEN-}N^{-1}$ in Operation

3.7.10. Graphs with an Odd Number of Vertices

Several examples of graphs with an odd number of vertices appear in Figure 3-38.

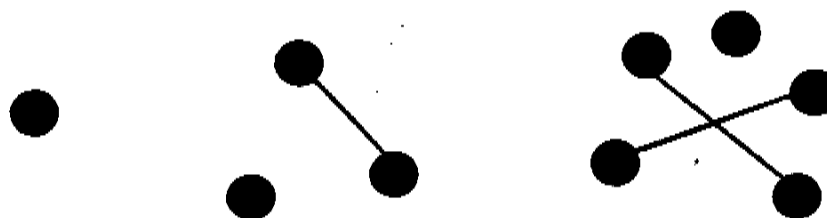


Figure 3-38: Some Graphs with an Odd Number of Vertices

The R-property $\text{ODD-}N$ is

$$(A_{xy} + A_w A_z)^*(K_1) \text{ where } x, y \in V, \\ w, z \in V, w \neq z$$

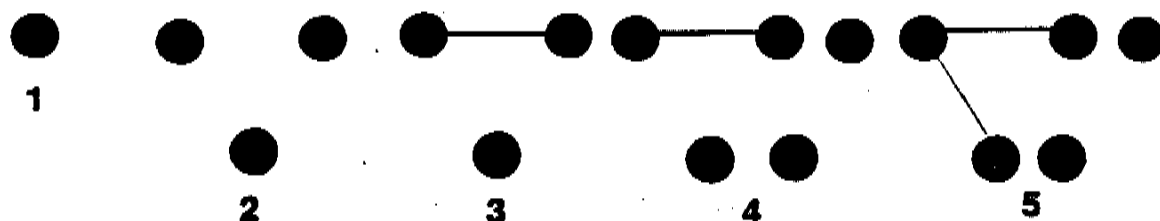


Figure 3-39: A Sample Run of ODD-N

Figure 3-39 shows the iterative steps in a sample run of ODD-N. The floor for graphs with an odd number of vertices is $\langle P_1, L_1, \Sigma_2 \rangle$, lower than that for graphs with an even number of vertices because of the simpler seed graph.

ODD-N is correct; it adds single arbitrary legal edges and vertices two at a time. The inverse for ODD-N has exactly the same f^{-1} and σ^{-1} as those for EVEN-N, namely,

$$\begin{aligned} f^{-1} &= D_{xy} + D_z D_w \\ \sigma^{-1} &= x, y \in V, xy \in E \\ &\quad w, z \in V, w \neq z, d(w) = 0, d(z) = 0 \end{aligned}$$

The floor shifts to $\langle P_2, L_1, \Sigma_3 \rangle$. Figure 3-40 shows ODD-N⁻¹ operating on a graph $G \in G_p$ and a graph $G \in G_p$. ODD-N⁻¹ is correct; it deletes the edges of a graph with an odd number of vertices and removes the isolated vertices two at a time until the graph is isomorphic to K_1 . A graph with an even number of vertices will go from E_2 to $\langle \emptyset, \emptyset \rangle$ and then fail. Thus ODD-N is complete.

3.7.11. Graphs with an Even Number of Edges

Several examples of graphs with an even number of edges appear in Figure 3-41. The R-property EVEN-M is

$$(A_x + A_{yz} A_{vw})^*(K_1) \text{ where } v, w, y, z \in V, yz, vw \in E, yz \neq vw$$

Figure 3-42 shows the iterative steps in a sample run of EVEN-M. The floor for graphs with an even number of edges is $\langle P_1, L_1, \Sigma_2 \rangle$.

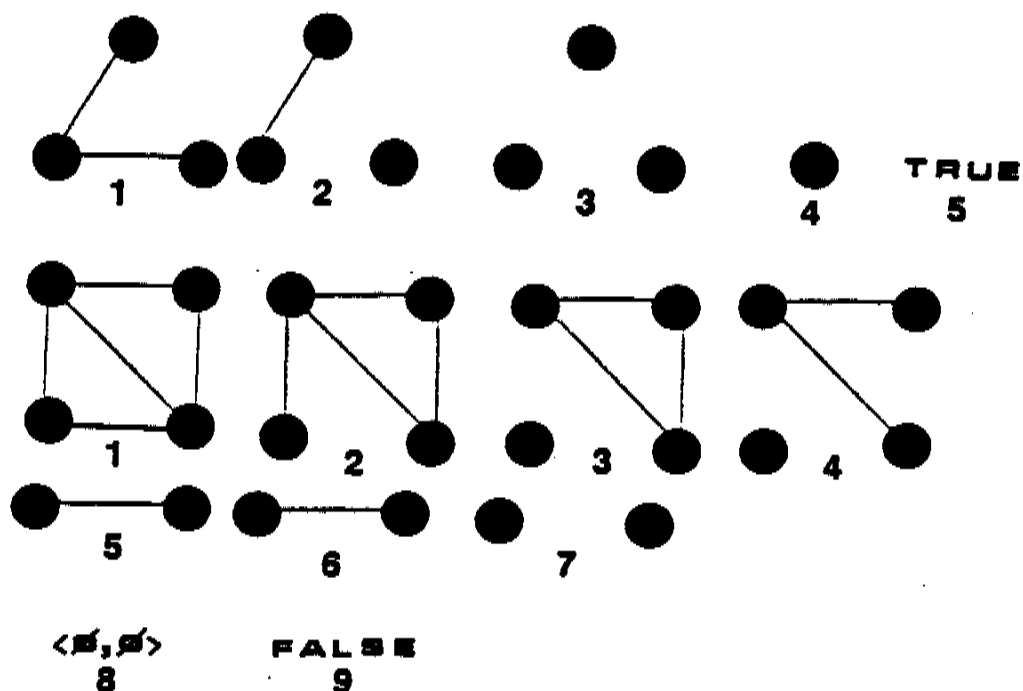
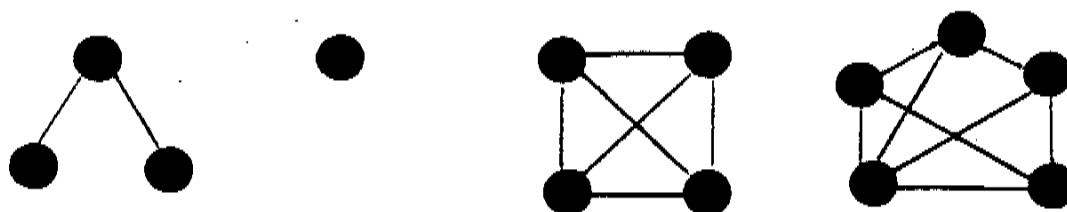
Figure 3-40: ODD- N^{-1} in Operation

Figure 3-41: Some Graphs with an Even Number of Edges

EVEN-M is correct; it adds vertices singly and legal edges two at a time. The inverse is computed by:

$$\begin{aligned}
 f^{-1} &= (A_x + A_{yz}A_{vw})^{-1} \\
 &= A_x^{-1} + (A_{yz}A_{vw})^{-1} \\
 &= A_x^{-1} + A_{vw}^{-1}A_{yz}^{-1} \\
 &= D_x + D_{vw}D_{yz}
 \end{aligned}$$

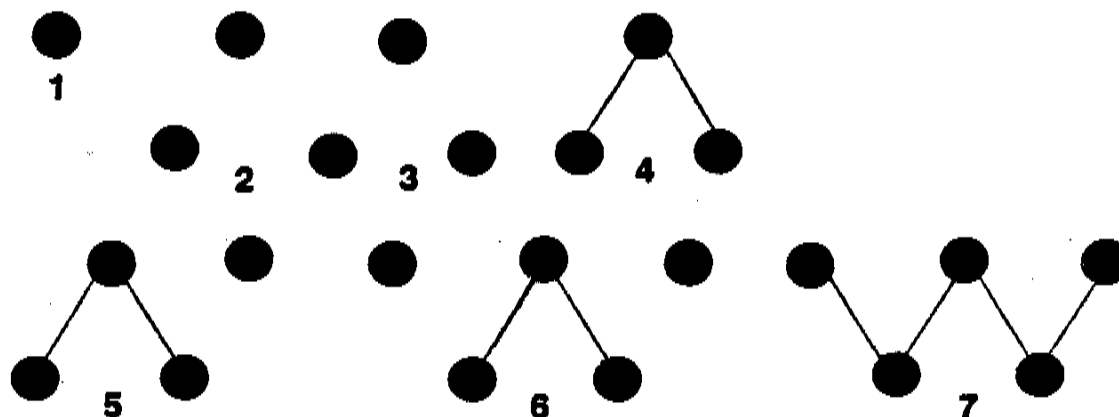


Figure 3-42: A Sample Run of EVEN-M

$$\begin{aligned}
 \sigma_{\text{pre}} &= x \in V \\
 &v, w, y, z \in V, yz, vw \in E, yz \neq vw \\
 \sigma^{-1} &= x \in V, d(x) = 0 \\
 &v, w, y, z \in V, yz, vw \in E, yz \neq vw
 \end{aligned}$$

The floor shifts to $\langle P_2, L_1, \Sigma_3 \rangle$. Figure 3-43 shows EVEN-M^{-1} operating on a graph $G \in G_p$ and a graph $G \in G_p$. EVEN-M^{-1} deletes singly the isolated vertices of a graph with an even number of edges and removes the edges two at a time until the graph is isomorphic to K_1 . A graph with an odd number of edges will reduce to K_2 and fail. Thus EVEN-M^{-1} is correct and EVEN-M is complete.

3.7.12. Graphs with an Odd Number of Edges

Several examples of graphs with an odd number of edges appear in Figure 3-44. The R-property ODD-M is

$$(A_x + A_{yz}A_{vw})^*(K_2) \text{ where } v, w, y, z \in V, yz, vw \in E, yz \neq vw$$

Figure 3-45 shows the iterative steps in a sample run of ODD-M . The floors for graphs with an odd number of edges are $\langle P_1, L_{1n}, \Sigma_2 \rangle$ and $\langle P_1, L_3, \Sigma_2 \rangle$.

ODD-M is correct; it adds vertices singly and legal edges two at a time. The inverse for ODD-M has exactly the same f^{-1} and σ^{-1} as those for EVEN-M .

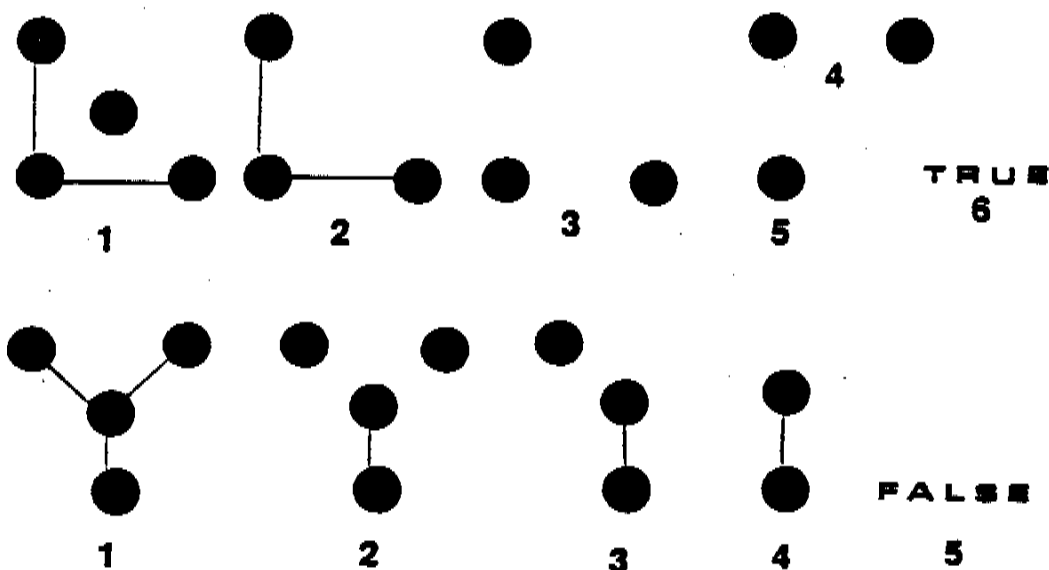
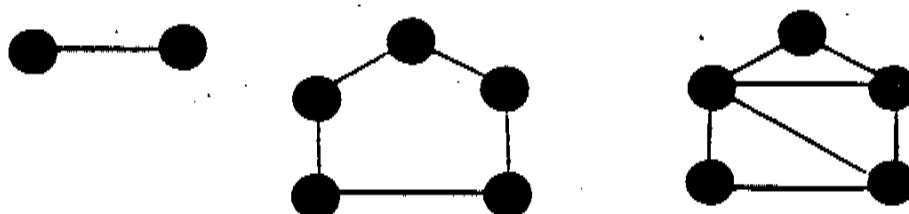
Figure 3-43: EVEN-M^{-1} in Operation

Figure 3-44: Some Graphs with an Odd Number of Edges

namely

$$f^{-1} = D_x + D_{vw} D_{yz}$$

$$\sigma^{-1} = \{x \in V, d(x) = 0\}$$

$$v, w, y, z \in V, yz, vw \in E, yz \neq vw$$

The floors shift to $\langle P_2, L_1, \Sigma_3 \rangle$ and $\langle P_2, L_3, \Sigma_3 \rangle$. Figure 3-46 shows ODD-M^{-1} operating on a graph $G \in G_p$ and a graph $G \in G_p$. ODD-M^{-1} deletes singly the isolated vertices of a graph with an odd number of edges and removes the edges two at a time until the graph is isomorphic to K_2 . A graph with an even number of edges will reduce to a graph isomorphic to $\langle \emptyset, \emptyset \rangle$ and then fail. Thus ODD-M^{-1} is correct and ODD-M is complete.

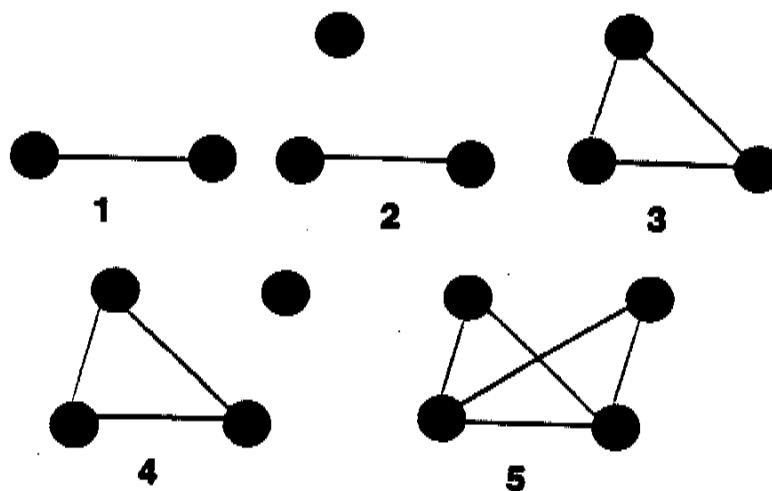
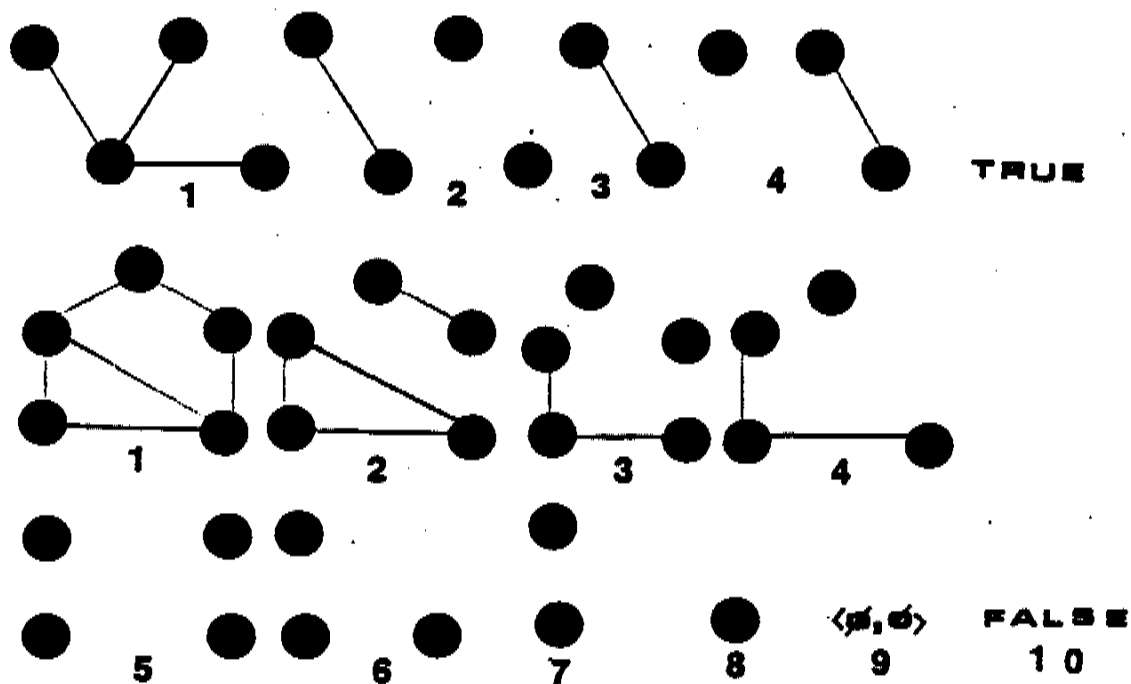


Figure 3-45: A Sample Run of ODD-M

Figure 3-46: $ODD-M^{-1}$ in Operation

3.7.13. Eulerian Graphs

A walk is a *trail* if all its edges are distinct and a *path* if all its vertices are distinct. A closed walk which traverses each edge of a graph exactly once and passes through every vertex at least once is called an *Eulerian walk*. An *Eulerian graph* is one for which an Eulerian walk exists. Several examples of Eulerian graphs appear in Figure 3-47.

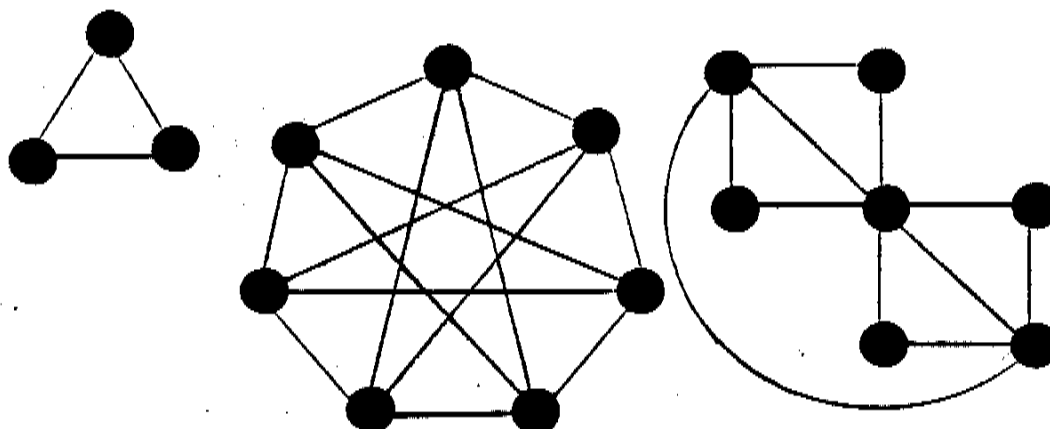


Figure 3-47: Some Eulerian Graphs

The R-property EULERIAN is

$$(S_{wvz} + Y_{v_1 v_2 \dots v_k})^*(K_3) \text{ where } w, z \in V, v \in V, wz \in E,$$

$$|\{v_1 v_2 \dots v_k\} \cap V| \geq 1,$$

$$v_i v_k \in E, v_i v_{i+1} \in E, i = 1, 2, \dots, k-1, \text{ distinct } v_i, i = 1, 2, \dots, k, k \geq 3$$

Figure 3-48 shows the iterative steps in a sample run of EULERIAN.

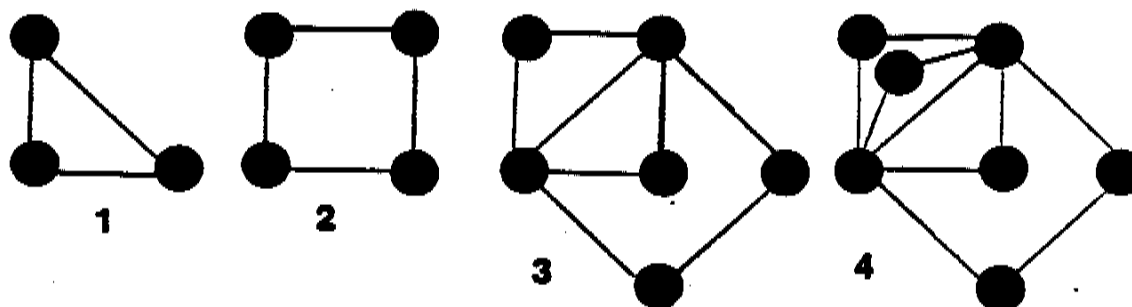


Figure 3-48: A Sample Run of EULERIAN

The algorithm either subdivides an existing edge or appends a cycle $C_{v_1 v_2 \dots v_k}$. This

cycle must have at least one vertex and no edges in common with the current graph. Since $Y_{v_1 v_2 \dots v_k} = A_{v_1 v_2} A_{v_2 v_3} \dots A_{v_{k-1} v_k} A_{v_1 v_k} A_{v_1} A_{v_2} \dots A_{v_k}$ and K_3 is describable in L_{1n} or L_2 , the floors for Eulerian graphs are $\langle P_2, L_{1n}, \Sigma_5 \rangle$ and $\langle P_2, L_2, \Sigma_5 \rangle$.

Given a graph G which is Eulerian, there is an Eulerian walk for it. Where the edge wz occurred in that walk, we substitute wv, v, vz and the walk will remain Eulerian. In any one location where $v_i \in V$ occurred in the Eulerian walk, we substitute the cycle

$$v_i v_{i+1} v_{i+2} \dots v_n v_1 v_2 \dots v_{i-1} v_i$$

and the walk will remain Eulerian. Thus cycle additions are closed path "detours" appended to the original Eulerian walk. We have shown that EULERIAN is correct.

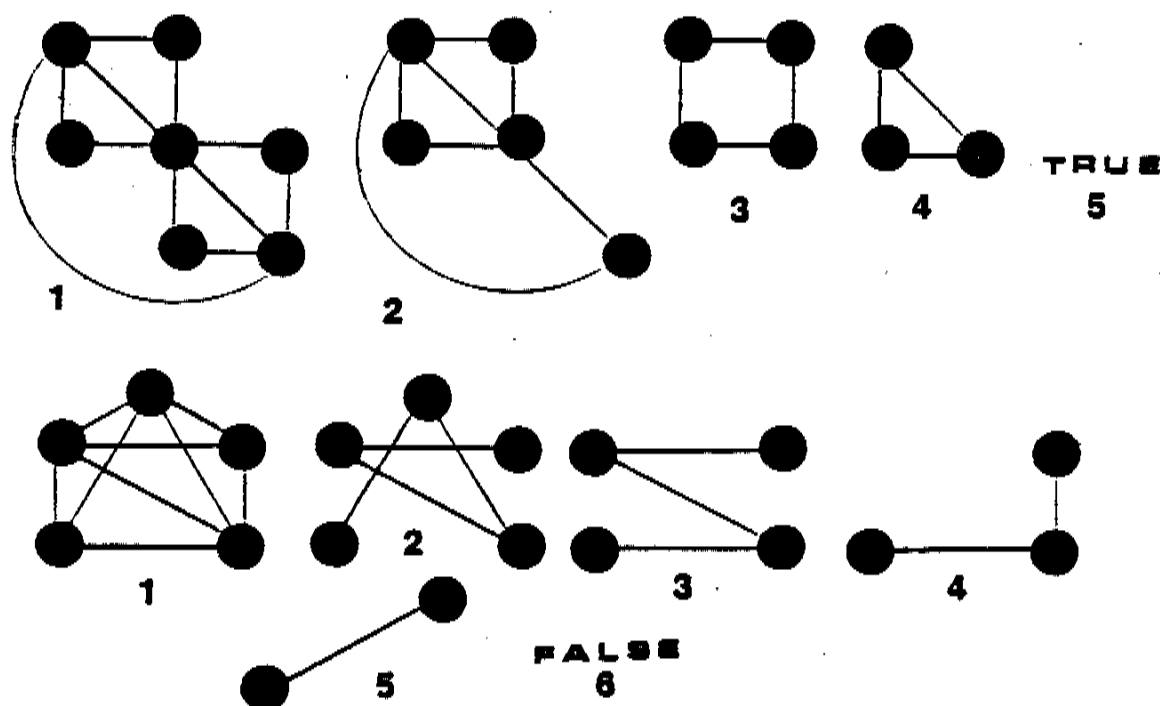
The inverse is computed by:

$$\begin{aligned} f^{-1} &= (S_{wvz} + Y_{v_1 v_2 \dots v_k})^{-1} \\ &= S_{wvz}^{-1} + Y_{v_1 v_2 \dots v_k}^{-1} \\ &= S_{wvz}^{-1} + Y_{v_1 v_2 \dots v_k} \\ &= D_v D_w D_z A_{xv} + Y_{v_1 v_2 \dots v_k} \end{aligned}$$

$$\begin{aligned} \sigma_{pre} = \sigma &= w, z \in V, wz \in E \\ &|\{v_1 v_2 \dots v_k\} \cap V| \geq 1, v_1 v_k \in E, \\ &v_i v_{i+1} \in E, i = 1, 2, \dots, k-1, \text{ distinct } v_i, i = 1, 2, \dots, k, \\ &k \geq 3 \end{aligned}$$

$$\begin{aligned} \sigma^{-1} &= \text{distinct } v, w, z \in V, wv, vz \in E, wz \in E, d(v) = 2 \\ &\text{distinct } v_1, v_2, \dots, v_k \in V, \\ &v_i v_{i+1} \in E, i = 1, 2, \dots, k-1, v_1 v_k \in E, k \geq 3, \\ &\text{not all } v_1, v_2, \dots, v_k \text{ of degree } \leq 2 \end{aligned}$$

The floors shift to the less powerful $\langle P_2, L_{1n}, \Sigma_3 \rangle$ and $\langle P_2, L_2, \Sigma_3 \rangle$. Figure 3-49 shows $EULERIAN^{-1}$ operating on a graph $G \in G_p$ and a graph $G \in G_p$. Note our first application of tentative deletion (Rule 4) in f^{-1} via $Y_{v_1 v_2 \dots v_k}$. It is not clear whether all (or even any) of the vertices $v_1, v_2, \dots, v_k$ were present before the cycle was added. After the edges of the cycle are deleted, however, those vertices of

Figure 3-49: $EULERIAN^{-1}$ in Operation

degree zero will clearly be the ones introduced by the cycle and deleted afterwards. If all of $v_1, v_2, \dots, v_k$ were introduced to the graph by the cycle addition, they would all be of degree two on its completion, hence the "not all" statement.

On an Eulerian graph $EULERIAN^{-1}$ detaches closed path "detours" from the underlying Eulerian walk without ever deleting the entire graph when it is a cycle, thanks to "not all $v_1, v_2, \dots, v_k$ of degree ≤ 2 ." Once the Eulerian graph being tested has no further closed path "detours", it is a simple cycle, to be contracted by S_{wvz}^{-1} until it is isomorphic to K_3 . On a non-Eulerian graph, $EULERIAN^{-1}$ also behaves correctly. There is a well known theorem in graph theory: "A graph is Eulerian if and only if each vertex is of even degree." A non-Eulerian graph will contain at least one vertex v of odd degree. Since the deletion of a cycle reduces the degree of each vertex by two, and the contraction of a subdivision deletes a degree two vertex, leaving all other degrees unchanged, v will remain of odd degree and a non-Eulerian graph will never become isomorphic to K_3 under repeated application of $EULERIAN^{-1}$. Thus $EULERIAN^{-1}$ is correct and $EULERIAN$ is complete.

3.7.14. Graphs with K Vertices

Several examples of graphs with $k = 3$ vertices appear in Figure 3-50.

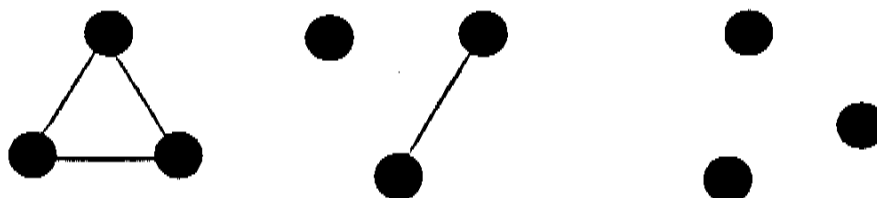


Figure 3-50: Some Graphs with 3 Vertices

The R-property K-VERTICES is:

$$A_{xy}^*(E_k) \text{ where } x, y \in V$$

Figure 3-51 shows the iterative steps in a sample run of K-VERTICES for $k = 5$.

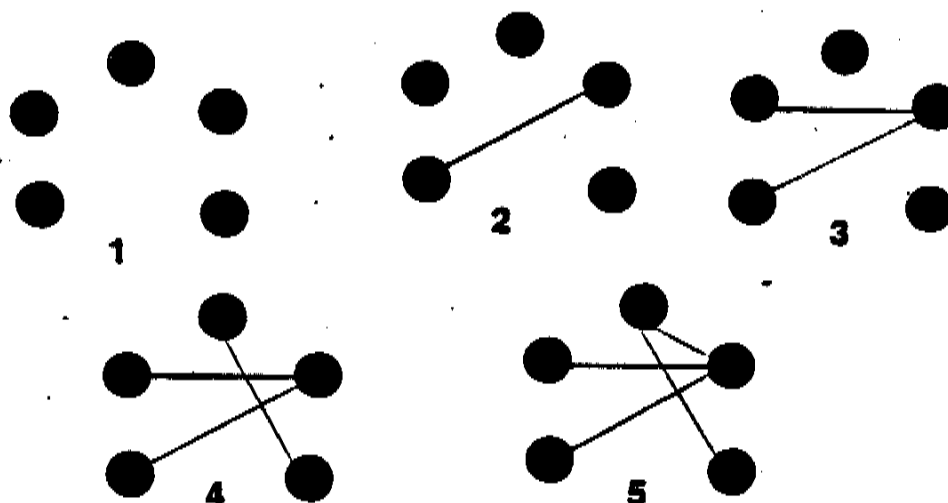


Figure 3-51: 5-VERTICES in Operation

L_1 , L_2 and L_3 would characterize E_k and any other edgeless graph as $E = 0$. A precise description of E_k is first available in L_{1n} as $E = 0$ and $n = k$. The floor for graphs with k vertices is $\langle P_1, L_{1n}, \Sigma_1 \rangle$.

K-VERTICES is correct: it only adds edges and cannot change n on any iteration. This algorithm is capable of running indefinitely, although eventually its additions are likely to be repetitive, since a graph on k vertices has at most k^2

edges.

The inverse is computed by:

$$\begin{aligned}
 f^{-1} &= A_{xy}^{-1} \\
 &= D_{xy} \\
 \sigma_{pre}^{-1} &= x, y \in V, xy \in E \\
 \sigma^{-1} &= x, y \in V, xy \in E
 \end{aligned}$$

The floor shifts to $\langle P_{2,L_1}, \Sigma_1 \rangle$. Figure 3-52 shows $K\text{-VERTICES}^{-1}$ operating on a graph $G \in G_p$ and a graph $G \in G_p$ for $k = 4$.

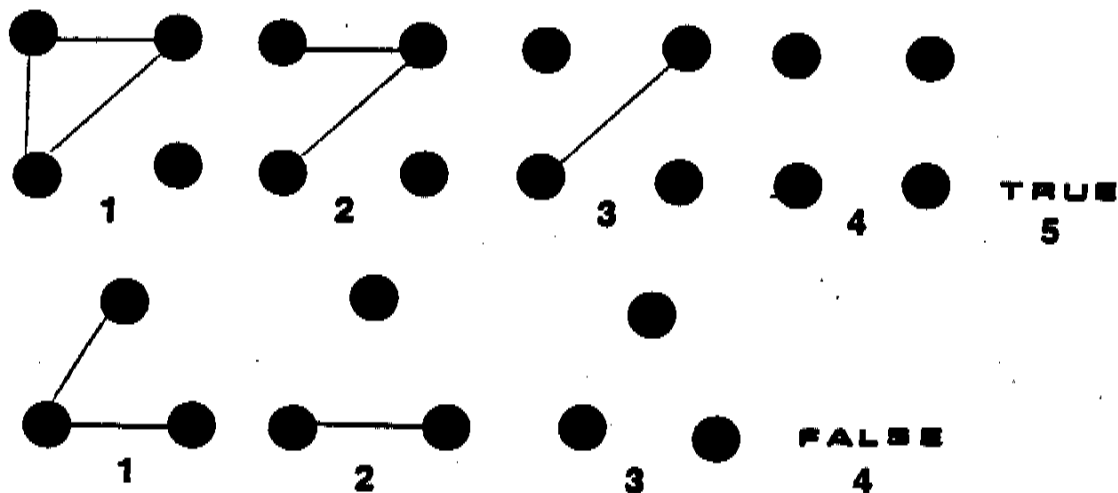


Figure 3-52: 4-VERTICES^{-1} in Operation

$K\text{-VERTICES}^{-1}$ is correct; it deletes all edges and then tests for isomorphism between two sets of isolated vertices. Thus $K\text{-VERTICES}$ is complete.

3.7.15. Graphs with K Edges

Several examples of graphs with a fixed number of edges $k = 3$ appear in Figure 3-53. We define $M_k = \langle \{v_1, v_2, \dots, v_{2k}\}, \{v_1v_2, v_3v_4, \dots, v_{2k-1}v_{2k}\} \rangle$ to be the matching graph on $2k$ vertices. M_k consists of $2k$ vertices and k edges such that each vertex is "matched" via an edge with exactly one other vertex. The R-property $K\text{-EDGES}$ is:

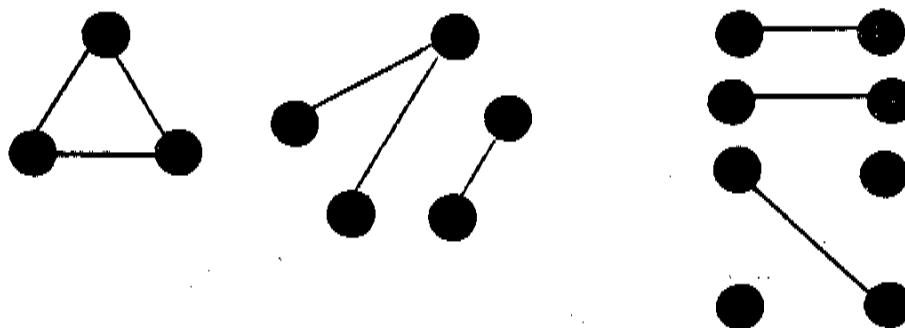


Figure 3-53: Some Graphs with 3 Edges

$$(A_x + I_{yz})^*(M_k) \text{ where distinct } y, z \in V, yz \in E, d(y) > 0, d(z) > 0, \\ \sim [w \in V, yw, zw \in E]$$

Figure 3-54 shows the iterative steps in a sample run of K-EDGES for $k = 5$. For $k > 2$, $\{M_k\}$ will not be uniquely describable in any language but L_Ω . Thus the floor for graphs with k edges is $\langle P_4, L_\Omega, \Sigma_5 \rangle$.

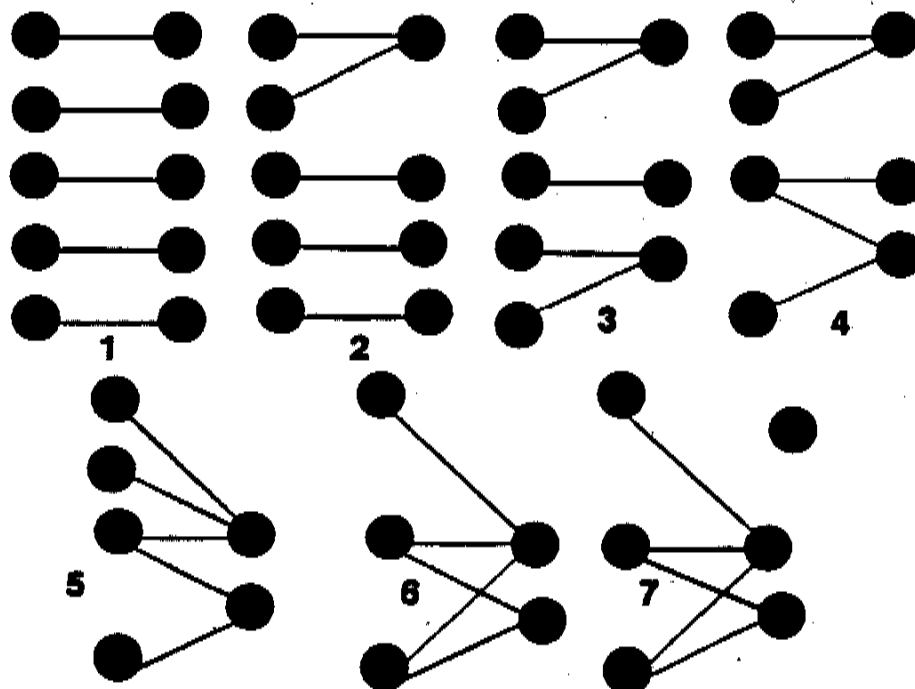


Figure 3-54: 5-EDGES in Operation

K-EDGES is correct; it adds isolated vertices and merges unisolated ones. An edge could be lost during such a merger only if the vertices being merged had a common neighbor; "there does not exist $w \in V$ such that $yw, zw \in E$ " prevents this. We prevent the transformation of an edge into a loop by " $yz \in E$ ".

Although we have never required it explicitly, a seed graph has been employed until now as a minimal case of a property p . Thus far this minimality has been directed to the values m and n . In K-EDGES, however, an intuitive incremental approach such as:

$$(A_x + A_{yz} D_{vw})^*(S)$$

for some S with minimal n will not be readily invertible, because it will be possible for the inverse

$$(D_x + A_{vw} D_{yz})$$

to churn in place, exchanging edges, indefinitely. In the formulation of an R-property we must strive for a monotonic function on the graph sequence to insure termination of the inverse. In the case of K-EDGES, this function is decreasing in the number of connected components, a topic to be defined and discussed later in this chapter.

The inverse is computed by:

$$\begin{aligned} f^{-1} &= (A_x + I_{yz})^{-1} \\ &= A_x^{-1} + I_{yz}^{-1} \\ &= D_x + F_{yzv_1 \dots v_k} \end{aligned}$$

$$\begin{aligned} \sigma_{pre} &= x \in V \\ &\quad \text{distinct } y, z \in V, d(y) > 0, d(z) > 0, yz \in E, \\ &\quad \sim [w \in V, yw, zw \in E] \end{aligned}$$

$$\begin{aligned} \sigma^{-1} &= x \in V, d(x) = 0 \\ &\quad y \in V, z \in V, yw_i \in E, i = 1, 2, \dots, k; 2 \leq d(y), d(y) > k \end{aligned}$$

The floor remains constant. The use of the fragmentation technique $F_{yzv_1 \dots v_k}$ requires some explanation. An arbitrary vertex y is subdivided into two vertices, y and z , which between them share all edges previously belonging to y without

duplication. (In particular z gets the adjacencies to $v_1, \dots, v_k$.) Normally, merger and fragmentation are inadequate inverses for each other because of the loss of information discussed in 3.4. In this instance previously merged vertices are identifiable by their degree, and merger is invertible. Figure 3-55 shows $K\text{-EDGES}^{-1}$ operating on a graph $G \in G_p$ and a graph $G \notin G_p$ for $k = 4$.

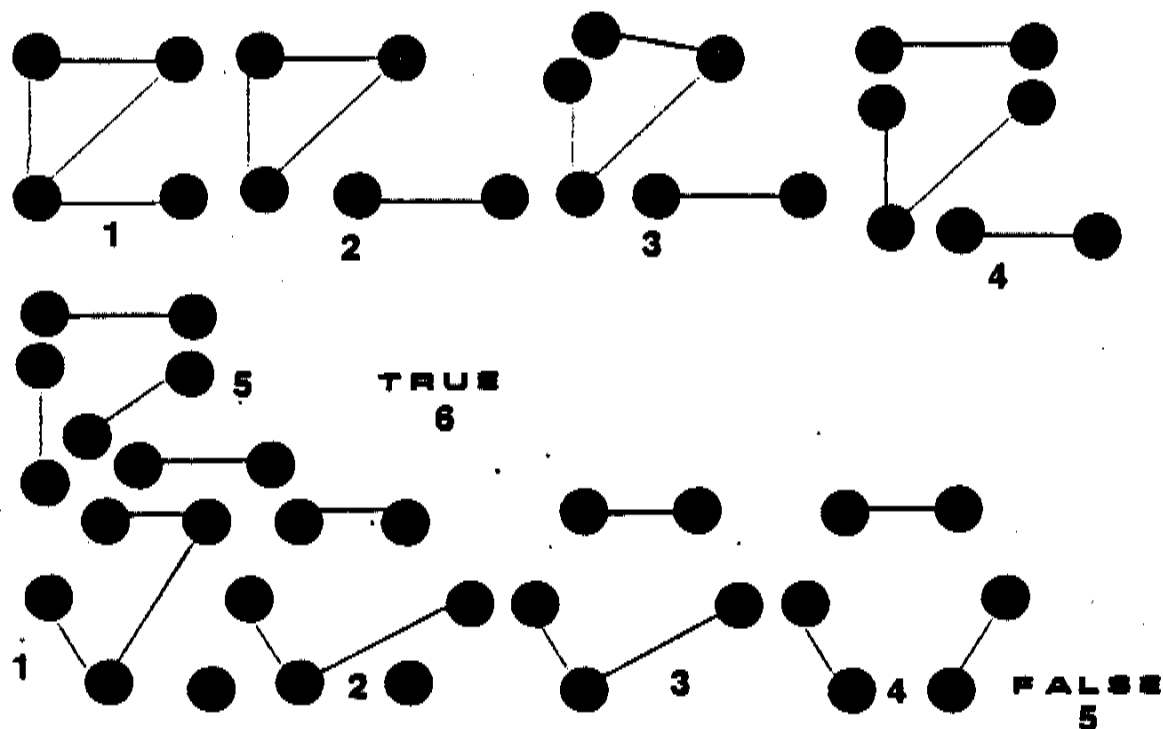


Figure 3-55: 4-EDGES^{-1} in Operation

On a graph with k edges, $K\text{-EDGES}^{-1}$ deletes isolated vertices and detaches the edges from one another until reaching M_k . On a graph with $m \neq k$ edges, $K\text{-EDGES}^{-1}$ will create M_m and then fail. Thus $K\text{-EDGES}^{-1}$ is correct and $K\text{-EDGES}$ is complete.

3.7.16. Graphs of Minimum Degree K

The *minimum degree* of a graph is the smallest degree of any of its vertices. Several examples of graphs with minimum degree $k = 3$ appear in Figure 3-56.

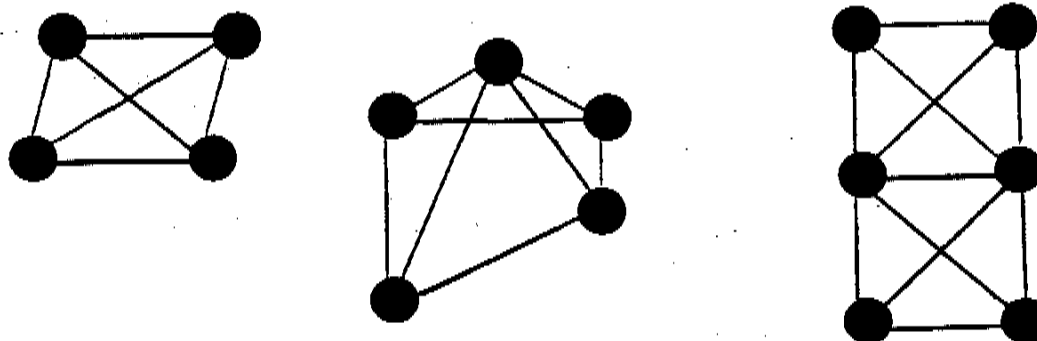


Figure 3-56: Some Graphs of Minimum Degree 3

The R-property MIN-K is:

$$(A_{xy} + A_{vx_1}A_{vx_2} \dots A_{vx_k}A_v)^*(K_{k+1}) \text{ where distinct } x, y, w \in V, d(w) = k$$

$$\text{distinct } x_1, x_2, \dots, x_k \in V, v \in V$$

The purpose of vertex w is to reserve at least one vertex which is always exactly of degree k . Figure 3-57 shows the iterative steps in a sample run of MIN-K for $k = 2$. Although $\{K_{k+1}\}$ is uniquely describable for $k = 1$, $k = 2$, and $k = 3$ in L_1 , L_3 and L_2 , respectively, for $k > 3$ $\{K_{k+1}\}$ is precisely describable only in L_{1n} . Thus the floor for graphs with minimum degree k is $\langle P_1, L_{1n}, \Sigma_3 \rangle$.

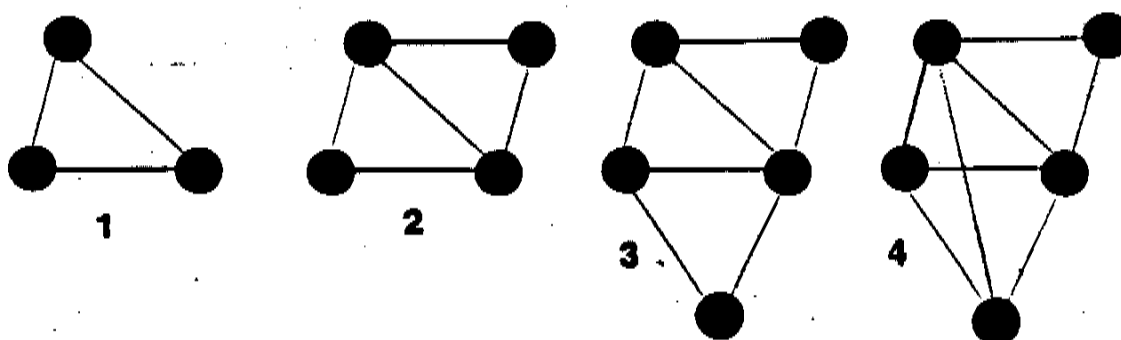


Figure 3-57: MIN-2 in Operation

MIN-K is correct: it adds only superfluous (degree-increasing) edges or new vertices of degree k to a graph. Vertex w prevents at least one vertex of degree k from a degree increase by a new edge.

The inverse is computed by:

$$\begin{aligned}
 f^{-1} &= (A_{xy} + A_{vx_1} A_{vx_2} \dots A_{vx_k} A_v)^{-1} \\
 &= A_{xy}^{-1} + (A_{vx_1} A_{vx_2} \dots A_{vx_k} A_v)^{-1} \\
 &= A_{xy}^{-1} + A_v^{-1} A_{vx_k}^{-1} A_{vx_{k-1}}^{-1} \dots A_{vx_1}^{-1} \\
 &= D_{xy} + D_v D_{vx_k} D_{vx_{k-1}} \dots D_{vx_1} \\
 \sigma_{pre} &= \text{distinct } x, y, w \in V, xy \in E, d(w) = k \\
 &\quad \text{distinct } w, x_1, x_2, \dots, x_k \in V, v \in V \\
 \sigma^{-1} &= \text{distinct } x, y, w \in V, xy \in E, d(x) > k, d(y) > k, d(w) = k \\
 &\quad \text{distinct } v, x_1, x_2, \dots, x_k \in V, xv_1, xv_2, \dots, xv_k \in E, d(v) = k
 \end{aligned}$$

The floor shifts to $\langle P_2, L_{1n}, \Sigma_3 \rangle$. Figure 3-58 shows MIN-K^{-1} operating on a graph $G \in G_p$ and a graph $G \in G_p$ for $k = 4$.

On a graph with minimum degree k , MIN-K^{-1} deletes vertices of exactly degree k preserving a single degree k vertex and reduces the degree of vertices of degree larger than k , until only K_{k+1} remains. If a graph has a vertex z of degree less than k , MIN-K^{-1} cannot delete z nor increase its degree, and the graph will never be isomorphic to K_{k+1} . If a graph has all vertices of degree greater than k , MIN-K^{-1} can delete neither a vertex nor an edge, because no correct v or $w \in V$ can be found. Thus MIN-K^{-1} is correct and MIN-K is complete.

3.7.17. Graphs of Maximum Degree K

The *maximum degree* of a graph is the largest degree of any of its vertices. Several examples of graphs with maximum degree $k = 3$ appear in Figure 3-59. The R-property MAX-K is

$$(A_{xy} + A_z)^*(K_{1,k}) \text{ where distinct } x, y \in V, d(x) < k, d(y) < k$$

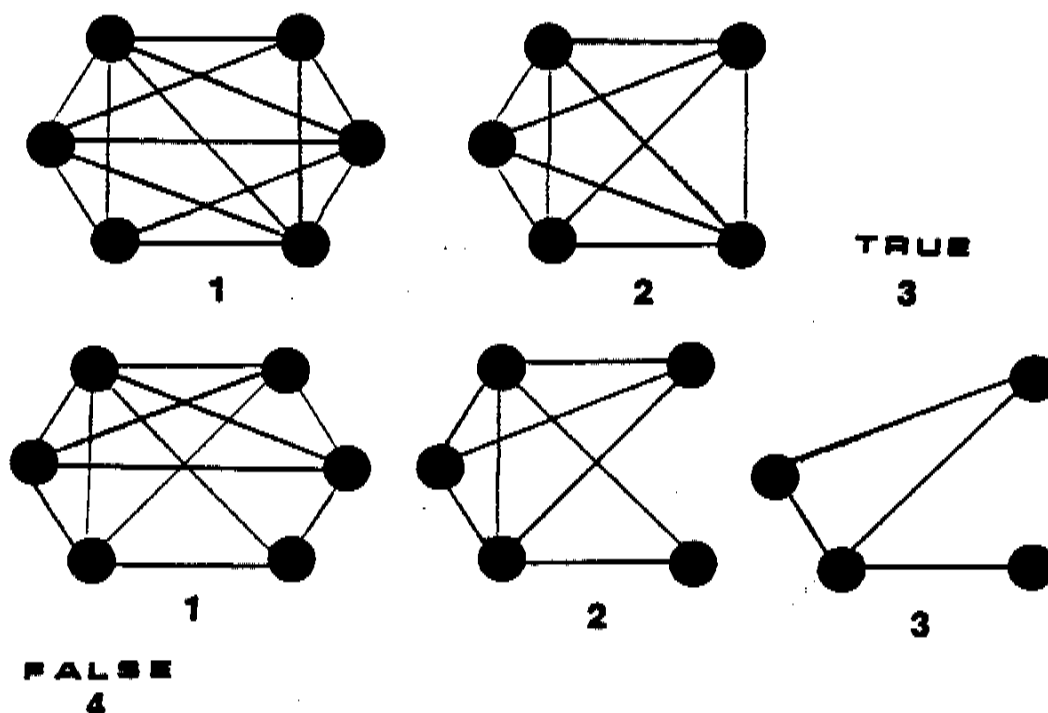


Figure 3-58: MIN-4^{-1} in Operation

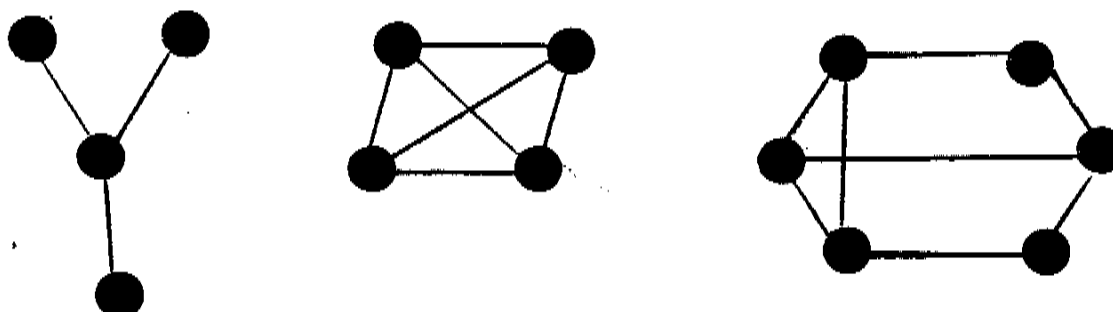


Figure 3-59: Some Graphs of Maximum Degree 3

Note that the seed graph is a star. Figure 3-60 shows the iterative steps in a sample run of MAX-K for $k = 5$. The floor for graphs with maximum degree k is $\langle P_1, L_2, \Sigma_3 \rangle$.

MAX-K is correct: it adds only isolated vertices z and edges xy which will not increase the degree of vertex x or y beyond k .

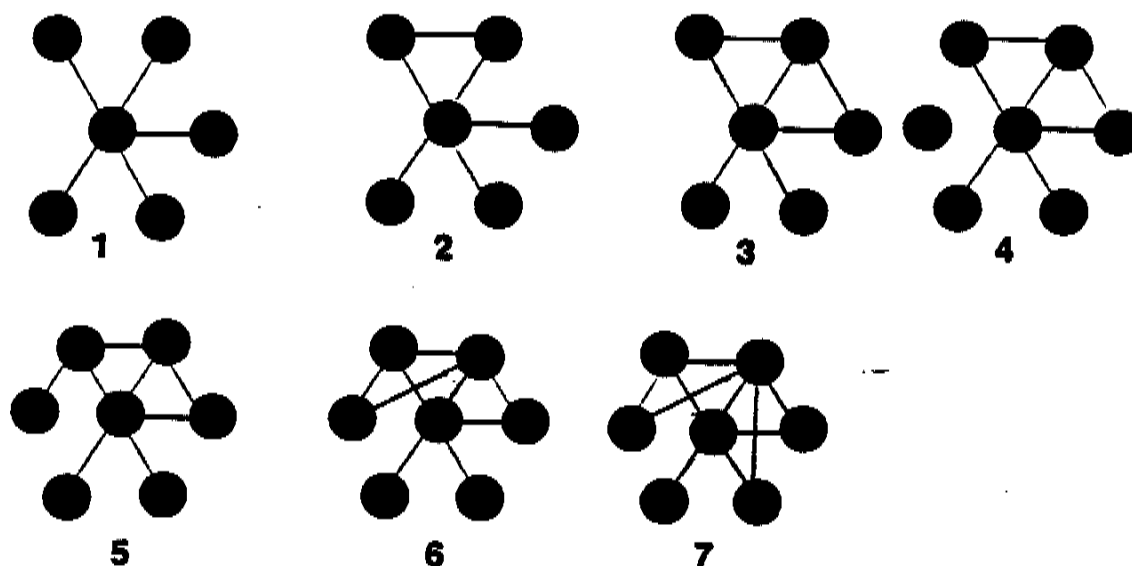


Figure 3-60: MAX-5 in Operation

The inverse is computed by:

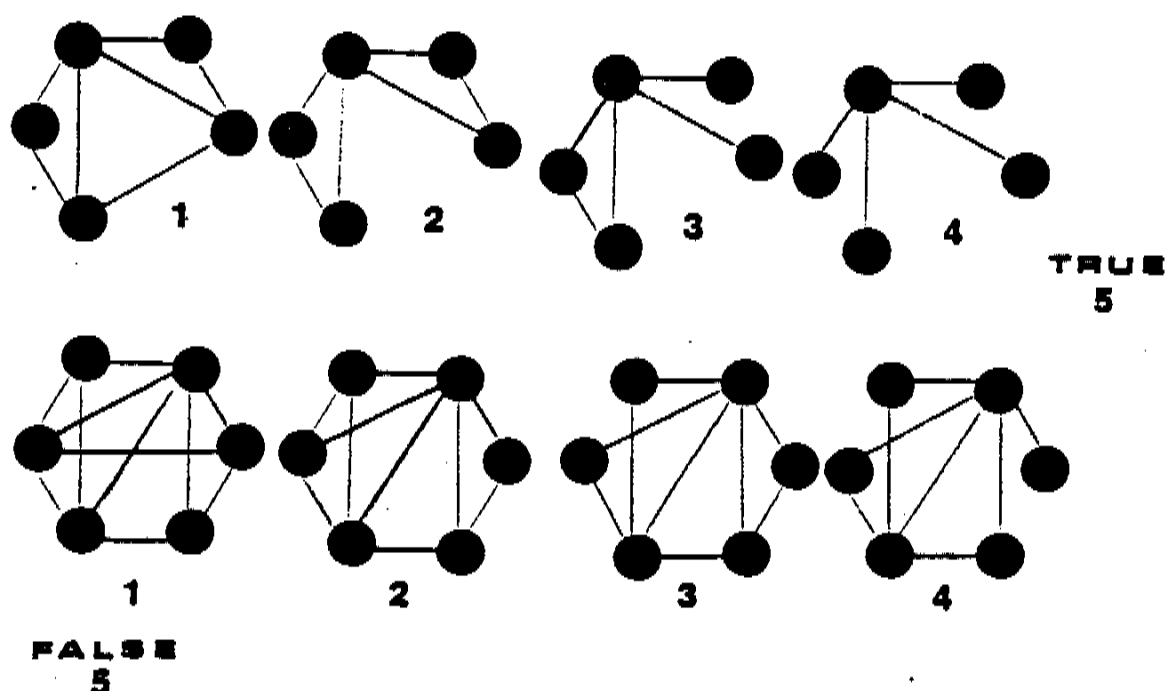
$$\begin{aligned}
 f^{-1} &= (A_{xy} + A_z)^{-1} \\
 &= A_{xy}^{-1} + A_z^{-1} \\
 &= D_{xy} + D_z
 \end{aligned}$$

$$\begin{aligned}
 \sigma_{pre} &= \text{distinct } w, x, y \in V, xy \in E, \\
 &\quad d(x) < k, d(y) < k, z \in V, d(w) = k \\
 &\quad z \in V
 \end{aligned}$$

$$\begin{aligned}
 \sigma^{-1} &= \text{distinct } w, x, y \in V, xy \in E, d(x) \leq k, d(y) \leq k, d(w) = k \\
 &\quad z \in V, d(z) = 0
 \end{aligned}$$

The floor shifts to $\langle P_2, L_0, \Sigma_3 \rangle$. Note that σ_{pre} incorporates properties of $K_{1,k}$ preserved under f to deduce that some vertex of degree k will always be present. Figure 3-61 shows $\text{MAX-}K^{-1}$ operating on a graph $G \in G_p$ and a graph $G \in G_p$ for $k = 4$.

On a graph with maximum degree k , $\text{MAX-}K^{-1}$ deletes isolated vertices and reduces the degree of vertices only of degree no larger than k , preserving a single

Figure 3-61: MAX-4^{-1} in Operation

degree k vertex until only $K_{1,k}$ remains. If a graph has a vertex z of degree greater than k , $\text{MAX-}K^{-1}$ cannot delete z or even reduce its degree, and the graph will never be isomorphic to $K_{1,k}$. If a graph has all vertices of degree less than k , no vertex degree will ever increase under $\text{MAX-}K^{-1}$ and isomorphism to $K_{1,k}$ will never occur. Thus $\text{MAX-}K^{-1}$ is correct and MAX-K is complete.

3.7.18. Pinwheels on Hubs of Size h

A *pinwheel* $W_{h,r}$ is a graph in which, for $r \geq 3$

$$V = \{v_1, v_2, \dots, v_h, w_1, w_2, \dots, w_r\}$$

$$E = \{v_i v_j \mid i \neq j, i, j = 1, 2, \dots, h\} \cup \{w_i w_{i+1} \mid i = 1, 2, \dots, r-1\} \cup \{w_1 w_r\} \cup \{v_i w_j \mid i = 1, 2, \dots, h; j = 1, 2, \dots, r\}$$

that is, a pinwheel is composed of a r/m ($C_{w_1 w_2 \dots w_r}$) and a complete graph (the *hub*) on $v_1, v_2, \dots, v_h$. Each vertex on the rim is adjacent to every hub vertex. A wheel is a special case of a pinwheel, where $h = 1$. A cycle may be thought of as a special case of a pinwheel where $h = 0$. Several examples of pinwheels appear

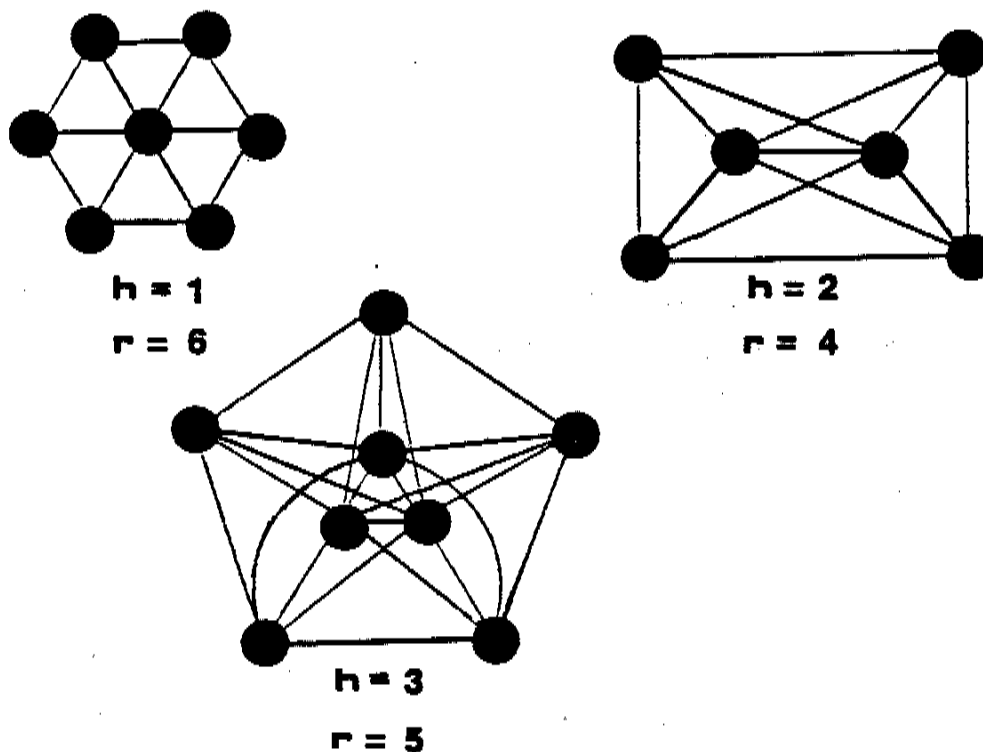


Figure 3-62: Some Pinwheels

in Figure 3-62. The R-property PINWHEEL is

$$(A_{w_1} A_{w_2} \dots A_{w_k} S_{xv})^* (W_{k,3}) \text{ where distinct } x, y, v_1, v_2, \dots, v_k \in V, v \notin V, xy \in E, \\ d(x) = k + 2, d(y) = k + 2, d(v_i) = \max, i = 1, 2, \dots, k$$

The seed graph is the smallest possible pinwheel on a hub of size k , one with a triangular rim. Figure 3-63 shows the iterative steps in a sample run of PINWHEEL for $k = 4$. Pinwheels are generated by gradually increasing the rim. The floor for pinwheels on hubs of size k is $\langle P_2, L_2, \Sigma_5 \rangle$, because $\{W_{k,3}\}$ is not precisely describable in any of our lower languages.

PINWHEEL is correct; it replaces any rim edge with a chain of two edges, connecting the new vertex v to each vertex in the hub. The inverse is computed by:

$$\begin{aligned}
 f^{-1} &= (A_{w_1} A_{w_2} \dots A_{w_k} S_{xv})^{-1} \\
 &= S_{xv}^{-1} A_{w_k}^{-1} A_{w_{k-1}}^{-1} \dots A_{w_1}^{-1} \\
 &= D_{v v_1} D_{v_1 v_2} \dots D_{v_{k-1} v_k} A_{xv} D_{v v_k} D_{v v_{k-1}} \dots D_{v v_1}
 \end{aligned}$$

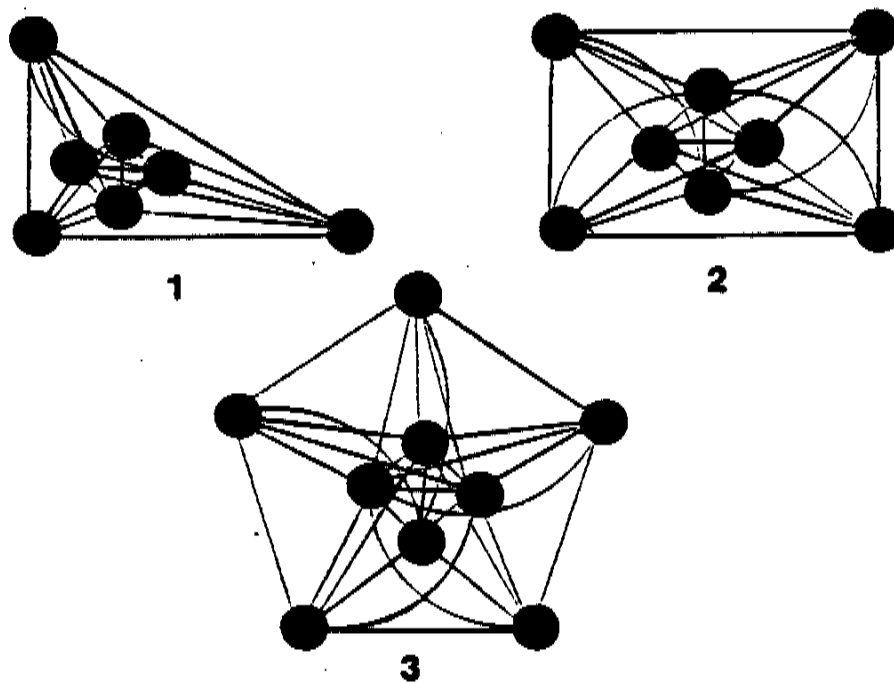
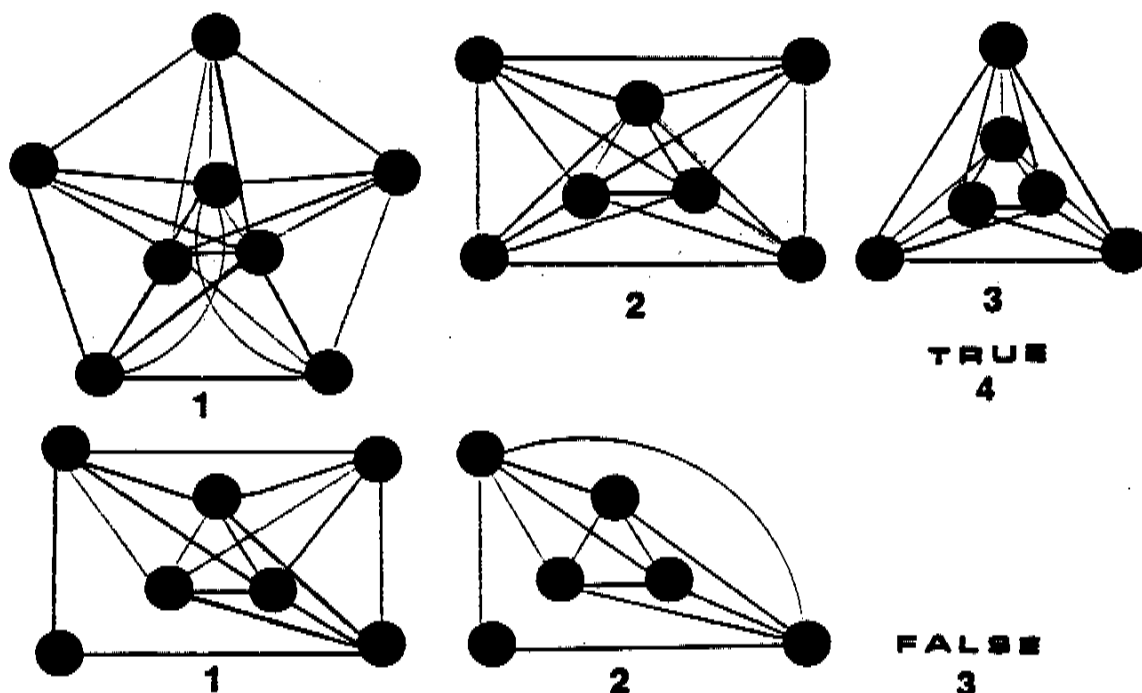


Figure 3-63: PINWHEEL in Operation

$$\begin{aligned}
 \sigma_{pre} = \sigma &= \text{distinct } x, y, v_1, v_2, \dots, v_k \in V, v \in V, xy \in E, \\
 &d(x) = k + 2, d(y) = k + 2, d(v_i) = \max, i = 1, 2, \dots, k \\
 \sigma^{-1} &= \text{distinct } x, y, v, v_1, v_2, \dots, v_k \in V, xy \in E, xv, vy \in E, \\
 &d(x) = k + 2, d(y) = k + 2, d(v) = k + 2, \\
 &d(v_i) = \max, i = 1, 2, \dots, k
 \end{aligned}$$

The floor remains constant. Figure 3-64 shows PINWHEEL^{-1} operating on a graph $G \in G_p$ and a graph $G \in G_p$ for $k = 3$.

On a pinwheel of k hubs, PINWHEEL^{-1} contracts the rim until the graph is isomorphic to $W_{k,3}$. On a graph which is not a pinwheel on k hubs, any vertices of degree other than $k + 2$ or $\max$ will remain untouched, with the pinwheel-like portions collapsing into $W_{k,2}$. An inadequate (incomplete) hub will remain so and never become isomorphic to $W_{k,3}$. Thus PINWHEEL^{-1} is correct and PINWHEEL is complete.

Figure 3-64: PINWHEEL⁻¹ in Operation

3.7.19. Graphs with K Components

If for every pair of vertices $x, y \in V$ there exists a path in E between them, the graph $G = \langle V, E \rangle$ is *connected*. A graph $G' = \langle V', E' \rangle$ is a *subgraph* of the graph $G = \langle V, E \rangle$ if and only if $V' \subseteq V$ and $E' \subseteq E$. A maximal connected subgraph of G is a *connected component* of G (or merely a *component*). Any graph G may be partitioned into its connected components $\langle V_1, S_1 \rangle, \langle V_2, S_2 \rangle, \dots, \langle V_k, S_k \rangle$ such that

$$G = \langle \bigcup_{i=1}^k V_i, \bigcup_{i=1}^k S_i \rangle$$

where the V_i 's partition V and the S_i 's partition E . We then say that there are k components in G . Several examples of graphs with $k = 3$ components appear in Figure 3-65. The R-property K-COMPONENTS is

$$(B_{wz} + (A_{xy} + A_{xv_1} A_{xv_2} \dots A_{xv_t} + A_{xy} A_{xv_1} A_{xv_2} \dots A_{xv_t}) F_{xyv_1 \dots v_t})^*(E_k)$$

where $w \in V, z \in V$

$$\text{distinct } x, v_1, v_2, \dots, v_t \in V, y \in V, d(x) > 0, x v_i \in E, i=1, 2, \dots, t$$

Recall that E_k is the edgeless graph on k vertices. There are two options. The first is a simple branch from vertex w . The second option begins when assigning x 's

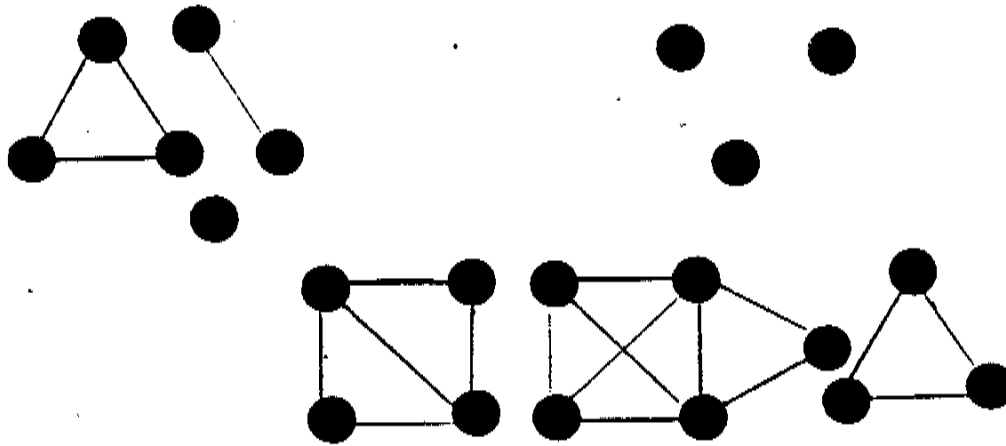


Figure 3-65: Some Graphs with Three Components

adjacency to $v_1, v_2, \dots, v_t$ to y instead. K-COMPONENTS fragments vertex x into vertices x and y . Then K-COMPONENTS either connects x and y or permits them to both be adjacent to $v_1, v_2, \dots, v_t$, or both. Figure 3-66 shows the iterative steps in a sample run of K-COMPONENTS for $k = 2$.

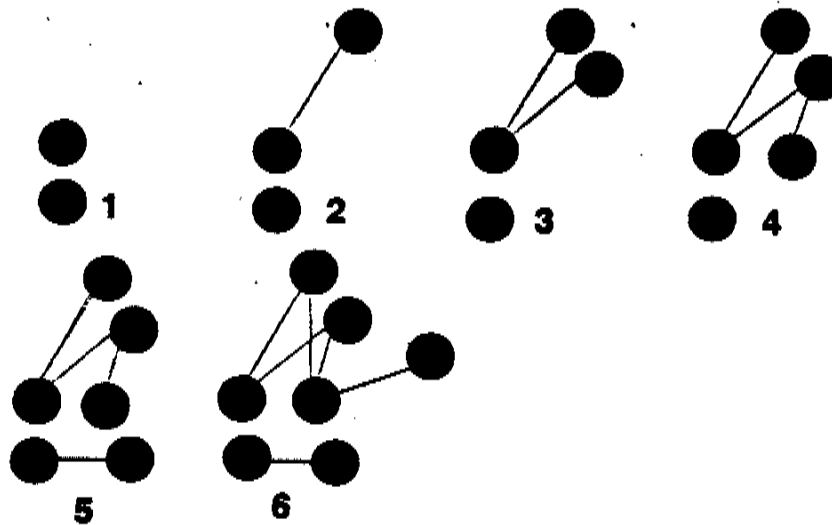


Figure 3-66: 2-COMPONENTS in Operation

The floor for graphs with k components is $\langle P_4, L_{1n}, \Sigma_3 \rangle$.

We now demonstrate that K-COMPONENTS is correct. Let G be a connected graph and apply one iteration of K-COMPONENTS to it, fragmenting vertex x into vertex x and vertex y . Let v be another vertex other than x and y , in G . We must

show that there exists an xv path and a yv path. Prior to fragmentation there was an xv path $xw...v$. After fragmentation either xw or yw is in E , say yw . It remains to construct an xw path. If A_{xy} were part of the iteration, then $xyw...v$ will be a valid xv path, else there is some v_i which is adjacent to both x and y , so that $xv_i yw...v$ is a valid xv path. In either case, G is connected and K -COMPONENTS is correct.

The inverse is computed by:

$$\begin{aligned}
 f^{-1} &= (B_{wz} + (A_{xy} + A_{xv_1} A_{xv_2} \dots A_{xv_t} + A_{xy} A_{xv_1} A_{xv_2} \dots A_{xv_t}) \\
 &\quad F_{xyv_1 \dots v_t})^{-1} \\
 &= B_{wz}^{-1} + F_{xyv_1 \dots v_t}^{-1} (A_{xy} + A_{xv_1} A_{xv_2} \dots A_{xv_t} + A_{xy} A_{xv_1} A_{xv_2} \dots \\
 &\quad A_{xv_t})^{-1} \\
 &= B_{wz}^{-1} + F_{xyv_1 \dots v_t}^{-1} (A_{xy}^{-1} + (A_{xv_1} A_{xv_2} \dots A_{xv_t})^{-1} + (A_{xy} A_{xv_1} A_{xv_2} \dots \\
 &\quad A_{xv_t})^{-1}) \\
 &= B_{wz}^{-1} + F_{xyv_1 \dots v_t}^{-1} (A_{xy}^{-1} + A_{xv_t}^{-1} A_{xv_{t-1}}^{-1} \dots A_{xv_1}^{-1} + A_{xv_t}^{-1} A_{xv_{t-1}}^{-1} \dots \\
 &\quad A_{xv_1}^{-1} A_{xy}^{-1}) \\
 &= D_z D_{wz} + I_{xy} (D_{xy} + D_{xv_t} D_{xv_{t-1}} \dots D_{xv_1} + D_{xv_t} D_{xv_{t-1}} \dots \\
 &\quad D_{xv_1} D_{xy}) \\
 \sigma_{pre} = \sigma &= w \in V, z \in V \\
 &\quad \text{distinct } x, v_1, v_2, \dots, v_t \in V, y \in V, d(x) > 0, \\
 &\quad xv_i \in E, i = 1, 2, \dots, t \\
 \sigma^{-1} &= \text{distinct } w, z \in V, wz \in E, d(z) = 1 \\
 &\quad \text{distinct } x, y, v_1, v_2, \dots, v_t \in V, d(x) > 0, d(y) > 0, \\
 &\quad xy \in E \text{ or } xv_i, yv_i \in E, i = 1, 2, \dots, t
 \end{aligned}$$

The floor remains constant. (Although "or" is not in Σ_3 , an equivalent, more complex notation in Σ_3 can convey the same list of possible bindings.) Figure 3-67 shows K -COMPONENTS $^{-1}$ operating on a graph $G \in G_p$ and a graph $G \in G_p$ for $k = 1$. On any graph, K -COMPONENTS $^{-1}$ merges vertices which are adjacent or have a common neighbor, until each component is contracted into a single isolated vertex.

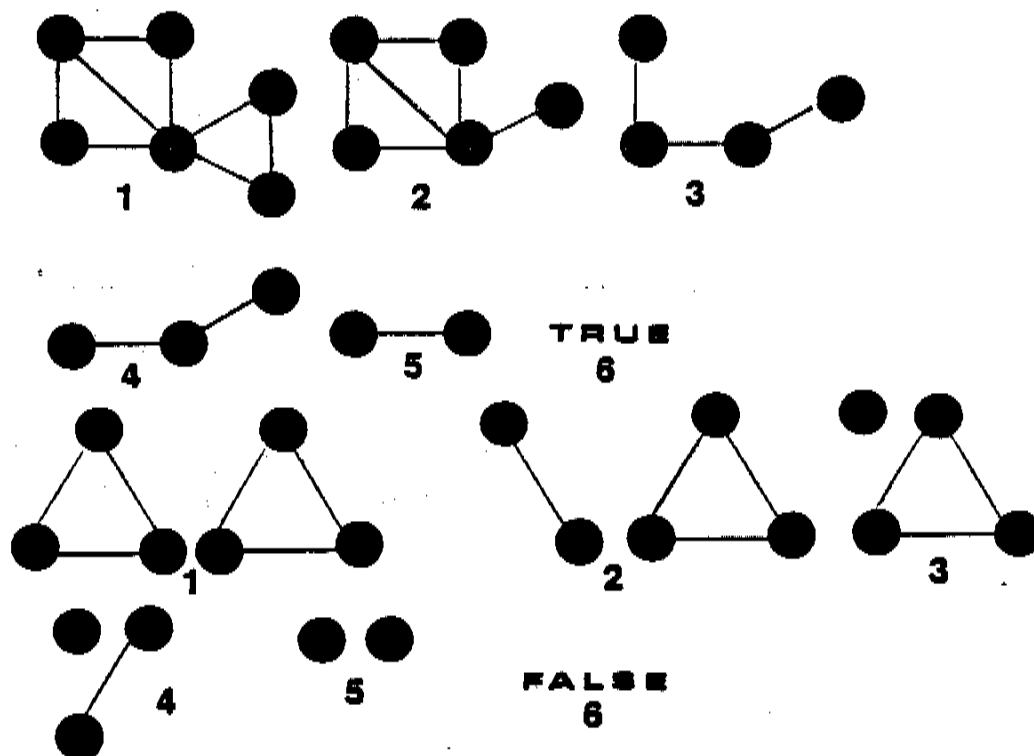


Figure 3-67: 1-COMPONENTS⁻¹ in Operation

Thus K-COMPONENTS⁻¹ is correct and K-COMPONENTS is complete.

A graph is *connected* if it consists of a single connected component. Thus connectedness is an R-property, a special case of K-COMPONENTS, for $k = 1$:

$$((A_{xy} + A_{yv_1} A_{v_1v_2} \dots A_{v_{t-1}v_t} + A_{xy} A_{v_1v_2} \dots A_{v_{t-1}v_t}) F_{xy})^*(K_1)$$

where distinct $x, v_1, v_2, \dots, v_t \in V$, $y \in V$, $d(x) > 0$, $xv_i \in E$, $i=1,2,\dots,t$

We will see another formulation for connectedness later in this chapter.

3.7.20. Regular Graphs

If every vertex of a graph is of the same fixed degree k , the graph is said to be *k-regular* or simply *regular*. Several examples of regular graphs appear in Figure 3-68 for varying k values. This property must deal with k even and k odd in separate algorithms. To simplify the notation we introduce some new composite operators,

$$EM_{xv_1 \dots v_k}, \quad OM_{vw_1 \dots v_{k-1} w_1 \dots w_{k-1}}, \quad OM^2_{vw_1 \dots v_{k+1} w_1 \dots w_{k-1}},$$

$$OM_{vw_1 \dots v_{k+1} w_1 \dots w_{k-1}}, \quad CM_{v_1 \dots v_{k+1}}, \quad \text{and} \quad FR_{v_1 \dots v_k}.$$

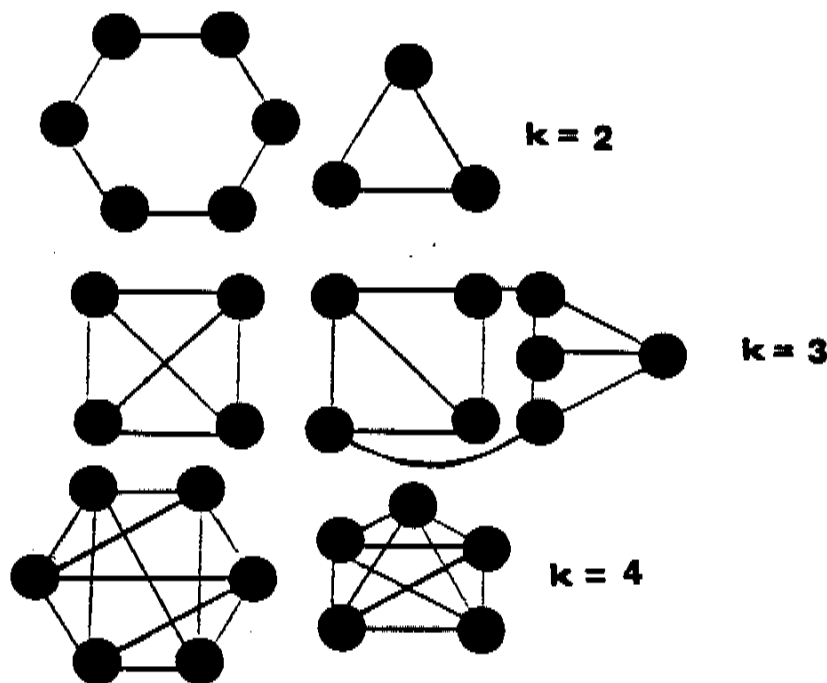


Figure 3-68: Some Regular Graphs

$EM_{xv_1 \dots v_k}$ introduces an even degree vertex x while maintaining the degree of every vertex to which it is adjacent. $EM_{xv_1 \dots v_k}$ deletes $k/2$ edges among k distinct vertices, and adds k new edges to x , i.e.,

$$EM_{xv_1 \dots v_k} = A_{xv_1} \dots A_{xv_k} D_{v_1 v_2} \dots D_{v_{k-1} v_k} A_x$$

$OM_{vw_1 \dots v_{k+1} w_1 \dots w_{k-1}}$ introduces two odd degree vertices v and w while maintaining the degree of every vertex to which they become adjacent. There are two ways this can be done. $OM^1_{vw_1 \dots v_{k+1} w_1 \dots w_{k-1}}$ deletes $(k-1)/2$ edges among $k-1$ distinct vertices for each of v and w and adds $k-1$ edges to each of v and w plus the edge vw . $OM^2_{vw_1 \dots v_{k+1} w_1 \dots w_{k-1}}$ deletes $(k+1)/2$ edges for v among $k+1$ vertices, permitting exactly one vertex (v_{k+1}) to repeat, deletes $(k-1)/2$ edges among $k-1$ distinct vertices for w , and "gives" the adjacency of v_{k+1} to w also. Thus we have:

$$OM^1_{vw_1 \dots v_{k+1} w_1 \dots w_{k-1}} = A_{vw_1} \dots A_{vw_{k-1}} D_{v_1 v_2} \dots D_{v_{k-2} v_{k-1}} A_{vw_1} \dots A_{vw_{k-1}} D_{w_1 w_2} \dots D_{w_{k-2} w_{k-1}} A_{vw} A_v A_w$$

$$\begin{aligned}
 OM_{vw_1 \dots v_{k+1} w_1 \dots w_{k-1}}^2 &= A_{vw_1} \dots A_{vw_k} D_{v_1 v_2} \dots D_{v_k v_{k+1}} A_{ww_1} \dots A_{ww_{k-1}} D_{w_1 w_2} \dots \\
 &\quad D_{w_{k-2} w_{k-1}} A_{ww_{k+1}} A_{vw} \\
 OM_{vw_1 \dots v_{k+1} w_1 \dots w_{k-1}} &= OM_{vw_1 \dots v_{k-1} w_1 \dots w_{k-1}}^1 + OM_{vw_1 \dots v_{k+1} w_1 \dots w_{k-1}}^2
 \end{aligned}$$

$CM_{v_1 \dots v_{k+1}}$ adds an entire copy of K_{k+1} on $v_1, \dots, v_{k+1}$ to the graph, i.e.,

$$CM_{v_1 \dots v_{k+1}} = A_{v_1 v_2} A_{v_1 v_3} \dots A_{v_1 v_k} A_{v_2 v_3} \dots A_{v_{k-1} v_k} A_{v_1 v_{k+1}} A_{v_2 v_{k+1}} \dots A_{v_k v_{k+1}}$$

$CM_{v_1 \dots v_k}$ adds k vertices of degree k and $(k+1)k/2$ edges to the graph, without changing the degrees of any previously existing vertices.

$FR_{v_1 \dots v_k}$ "fractures" the degree k vertex v_1 into a set of k vertices, $v_1, \dots, v_k$, which form among themselves a complete K_k subgraph. Each vertex assumes one of the adjacencies previously assigned to v_1 , i.e., if the neighbors of v_1 were $x_1, \dots, x_k$, then

$$FR_{v_1 \dots v_k} = A_{v_2 x_2} D_{v_1 x_2} A_{v_3 x_3} D_{v_1 x_3} \dots A_{v_k x_k} D_{v_1 x_k} CM_{v_1 \dots v_k}$$

Also for notational convenience we define the inverses:

$$EM_{xv_1 \dots v_k}^{-1} = D_x A_{v_{k-1} v_k} \dots A_{v_1 v_2} D_{xv_k} \dots D_{xv_1}$$

$$\begin{aligned}
 (OM_{vw_1 \dots v_{k-1} w_1 \dots w_{k-1}}^1)^{-1} &= D_w D_v D_{vw} A_{w_{k-2} w_{k-1}} \dots A_{w_1 w_2} D_{ww_{k-1}} \dots \\
 &\quad D_{ww_1} A_{v_{k-2} v_{k-1}} \dots A_{v_1 v_2} D_{w_{k-1}} \dots D_{w_1}
 \end{aligned}$$

$$\begin{aligned}
 (OM_{vw_1 \dots v_{k+1} w_1 \dots w_{k-1}}^2)^{-1} &= D_w D_v D_{vw_{k+1}} A_{w_{k-2} w_{k-1}} \dots A_{w_1 w_2} D_{ww_{k-1}} \dots \\
 &\quad D_{ww_1} A_{v_k v_{k+1}} \dots A_{v_1 v_2} D_{w_k} \dots D_{w_1}
 \end{aligned}$$

$$\begin{aligned}
 OM_{vw_1 \dots v_{k+1} w_1 \dots w_{k-1}}^{-1} &= (OM_{vw_1 \dots v_{k-1} w_1 \dots w_{k-1}}^1)^{-1} + \\
 &\quad (OM_{vw_1 \dots v_{k+1} w_1 \dots w_{k-1}}^2)^{-1}
 \end{aligned}$$

$$CM_{v_1 \dots v_{k+1}}^{-1} = D_{v_{k+1}} \dots D_{v_1} D_{v_k v_{k+1}} \dots D_{v_1 v_2}$$

$$FR_{v_1 \dots v_k}^{-1} = CM_{v_1 \dots v_k}^{-1} A_{v_1 x_k} D_{v_k x_k} A_{v_1 x_{k-1}} D_{v_{k-1} x_{k-1}} \dots A_{v_1 x_2} D_{v_2 x_2}$$

Now the R-property for EVEN-REGULAR is

$$(EM_{xv_1 \dots v_k} + A_{y_2 y_1} A_{y_1 y_2} D_{y_1 y_2} CM_{y_1 \dots y_{k+1}} D_{pq} + FR_{z_1 \dots z_k})^*(Q_{k+1}) \text{ where}$$

distinct $v_1, v_2, \dots, v_k \in V$, $x \in V$, $v_{2i-1}v_{2i} \in E$, $i=1, 2, \dots, k/2$

distinct $p, q \in V$, distinct $y_1, y_2, \dots, y_{k+1} \in V$, $pq \in E$

distinct $x_1, x_2, \dots, x_k, z_1 \in V$, distinct $z_2, z_3, \dots, z_k \in V$, $z_i x_i \in E$, $i = 1, 2, \dots, k$

The seed set Q_{k+1} requires some explanation. Let $K_{k+1}^1, K_{k+1}^2, \dots, K_{k+1}^t$ be t distinct copies of a complete graph on $k+1$ vertices. Take

$$Q_{k+1} = \{u_{i=1}^t K_{k+1}^i \mid t=1, 2, \dots\}$$

in other words, each element of Q_{k+1} is a t component graph, composed of t disjoint copies of K_{k+1} . Clearly $K_{k+1} \in Q_{k+1}$. As EVEN-REGULAR iterates on G , the number of connected components may or may not reduce, depending upon whether or not the v_i 's lie in the same component. Figure 3-69 shows the iterative steps in a sample run of EVEN-REGULAR for $k = 4$.

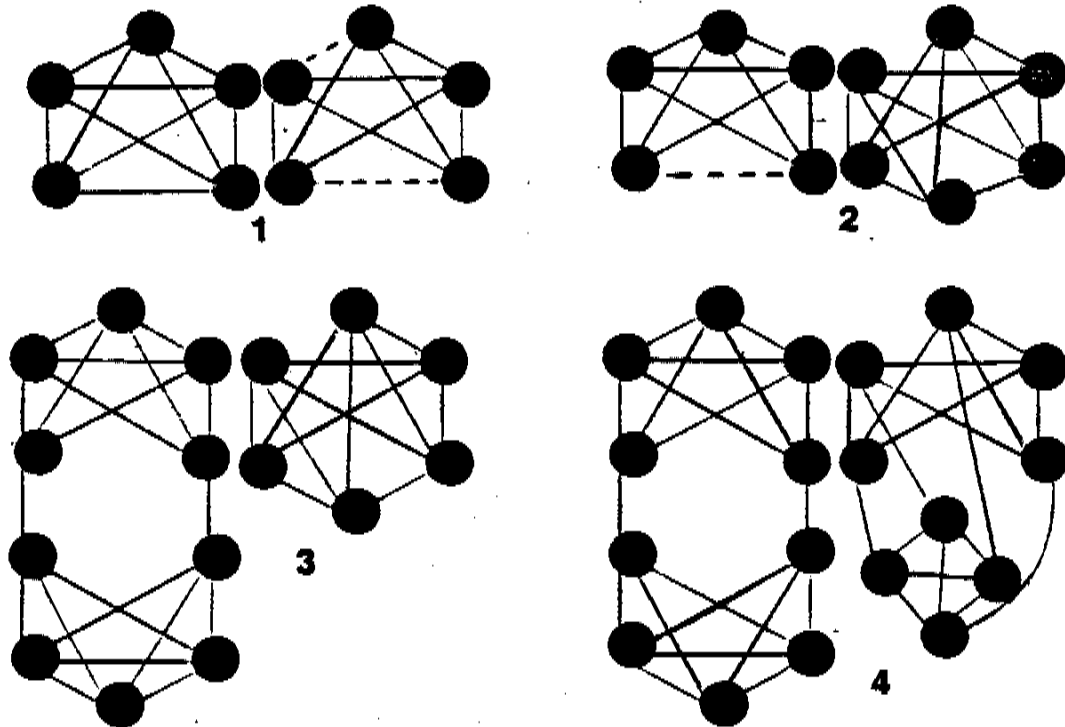


Figure 3-69: EVEN-REGULAR in Operation for $k = 4$

The edges to be used for the next iteration appear dotted in the figure. The floor for k -regular graphs with k even is $\langle P_2, L_0, \Sigma_2 \rangle$. EVEN-REGULAR has three options. The first adds vertex x , replacing $k/2$ edges with k edges. Each affected vertex v_i loses one edge and gains another, leaving $d(v_i)$ unchanged. The new vertex x is

constructed to have $d(x) = k$. The second option deletes an edge between two old vertices (p and q), adds a copy of K_{k+1} , deletes an edge from the copy ($y_1 y_2$) and connects the copy to the original graph with two edges ($y_1 p$ and $y_2 q$), restoring the degree of all four vertices to k . In total, $k+1$ new vertices and $(k+1)k/2$ new edges are added to the graph. The third option fragments a vertex (z_1) while maintaining the degrees of its neighbors. The vertices in the fragmentation each have degree k ($k-1$ edges among each other and one "external" edge to a previous neighbor of z_1). There are $k-1$ new vertices created, each of degree k . Thus EVEN-REGULAR is correct.

The inverse is computed by:

$$\begin{aligned}
 f^{-1} &= (EM_{xv_1 \dots v_k} + A_{y_2 q} A_{y_1 p} D_{y_1 y_2} CM_{y_1 \dots y_{k+1}} D_{pq} + \\
 &\quad FR_{z_1 \dots z_k})^{-1} \\
 &= EM_{xv_1 \dots v_k}^{-1} + (A_{y_2 q} A_{y_1 p} D_{y_1 y_2} CM_{y_1 \dots y_{k+1}} D_{pq})^{-1} + \\
 &\quad FR_{z_1 \dots z_k}^{-1} \\
 &= EM_{xv_1 \dots v_k}^{-1} + A_{pq} CM_{y_1 \dots y_{k+1}} A_{y_1 y_2} D_{y_1 p} D_{y_2 q} + \\
 &\quad FR_{z_1 \dots z_k}^{-1} \\
 \sigma_{pre} &= \text{distinct } v_1, v_2, \dots, v_k \in V, x \in V, v_{2i-1} v_{2i} \in E, \\
 &\quad i=1, 2, \dots, k/2, d(v_j) = k, j = 1, 2, \dots, k \\
 &\quad \text{distinct } p, q \in V, \text{ distinct } y_1, y_2, \dots, y_{k+1} \in V, pq \in E, \\
 &\quad d(p) = k, d(q) = k \\
 &\quad \text{distinct } x_1, x_2, \dots, x_k, z_1 \in V, \text{ distinct } z_2, z_3, \dots, z_k \in V, \\
 &\quad z_1 x_i \in E, i = 1, 2, \dots, k, d(z_1) = k \\
 \sigma^{-1} &= \text{distinct } x, v_1, v_2, \dots, v_k \in V, xv_i \in E, i=1, 2, \dots, k, \\
 &\quad v_{2i-1} v_{2i} \in E, i=1, 2, \dots, k/2, d(x) = k, d(v_i) = k, i=1, 2, \dots, k \\
 &\quad \text{distinct } p, q, y_1, y_2, \dots, y_{k+1} \in V, pq \in V, y_i y_j \in E, d(y_i) = k, \\
 &\quad i = 1, 2, \dots, k+1, j = 1, 2, \dots, k+1 \text{ except } y_1 y_2; \\
 &\quad y_1 p, y_2 q \in E \\
 &\quad \text{distinct } x_i, z_i \in V, x_i z_i \in E, d(x_i) = k, d(z_i) = k, i = 1, 2, \dots, k; \\
 &\quad x_i z_j \in E, j = 2, 3, \dots, k; z_r z_s \in E, r, s = 1, 2, \dots, k
 \end{aligned}$$

The floor shifts to $\langle P_2, L_2, \Sigma_3 \rangle$. Figure 3-70 shows EVEN-REGULAR^{-1} operating on a graph $G \in G_p$ and a graph $G \notin G_p$ for $k = 2$. The edges to be used for the next iteration appear dotted in the figure. Any vertex of degree other than k will be unmodifiable and isomorphism will eventually fail in a graph containing such a vertex. Under the first option, EVEN-REGULAR^{-1} removes a degree k vertex x and inserts $k/2$ edges maintaining the regularity of the v_i 's, until K_{k+1} is reached in each connected component. Under the second option, a copy of $K_{k+1} - xy$ (the complete graph on $k+1$ vertices missing a single edge) attached to two non-adjacent vertices (p and q) is deleted, removing $k+1$ vertices of degree k without changing the degree of any other vertex in the graph. Under the third option a copy of K_k , each of whose vertices has one different neighbor not in K_k , is replaced with a single vertex (z_i) of degree k without changing the degree of any other vertex in the graph. EVEN-REGULAR^{-1} is clearly correct for $k = 2$ (the last two options are instances of edge subdivisions) and we believe it to be correct for $k > 2$, although a formal proof will be offered elsewhere. At this writing no $G \in G_p$ has been shown inaccessible and its behavior on $G \notin G_p$ is correct, so we will postulate EVEN-REGULAR^{-1} to be correct and thus EVEN-REGULAR to be complete.

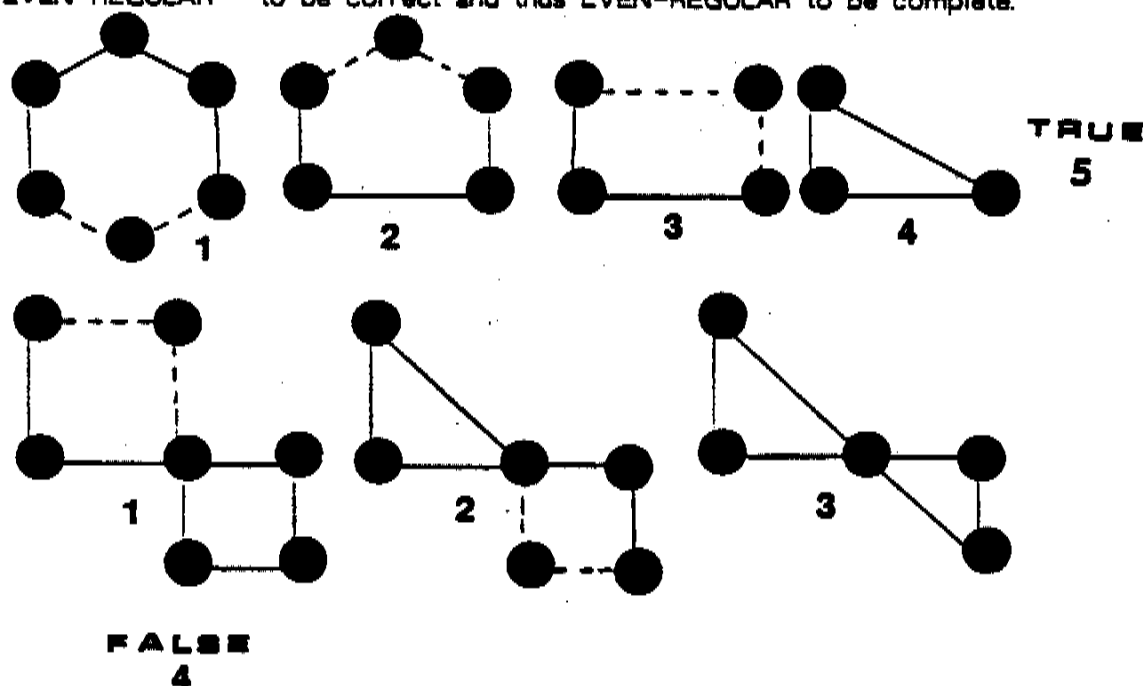


Figure 3-70: EVEN-REGULAR^{-1} in Operation for $k = 2$

The R-property for ODD-REGULAR is

$$(OM_{vwv_1 \dots v_{k+1} w_1 \dots w_{k-1}} + A_{y_2 q} A_{y_1 p} D_{y_1 y_2} CM_{y_1 \dots y_{k+1}} D_{pq} + FR_{z_1 \dots z_k})^*$$

(Q_{k+1}) where distinct $v, w \in V$, distinct $v_1, v_2, \dots, v_k \in V$,

distinct $w_1, w_2, \dots, w_{k-1} \in V$, $v_{2i-1} v_{2i}, w_{2j-1} w_{2j} \in E$,

$i=1, 2, \dots, (k+1)/2$; $j=1, 2, \dots, (k-1)/2$

distinct $p, q \in V$, distinct $y_1, y_2, \dots, y_{k+1} \in V$, $pq \in E$

distinct $x_1, x_2, \dots, x_k, z_1 \in V$, distinct $z_2, z_3, \dots, z_k \in V$, $z_i x_i \in E$, $i=1, 2, \dots, k$

ODD-REGULAR appends two vertices at a time to G because an odd regular graph must have n even. Figure 3-71 shows the iterative steps in a sample run of ODD-REGULAR for $k=5$.

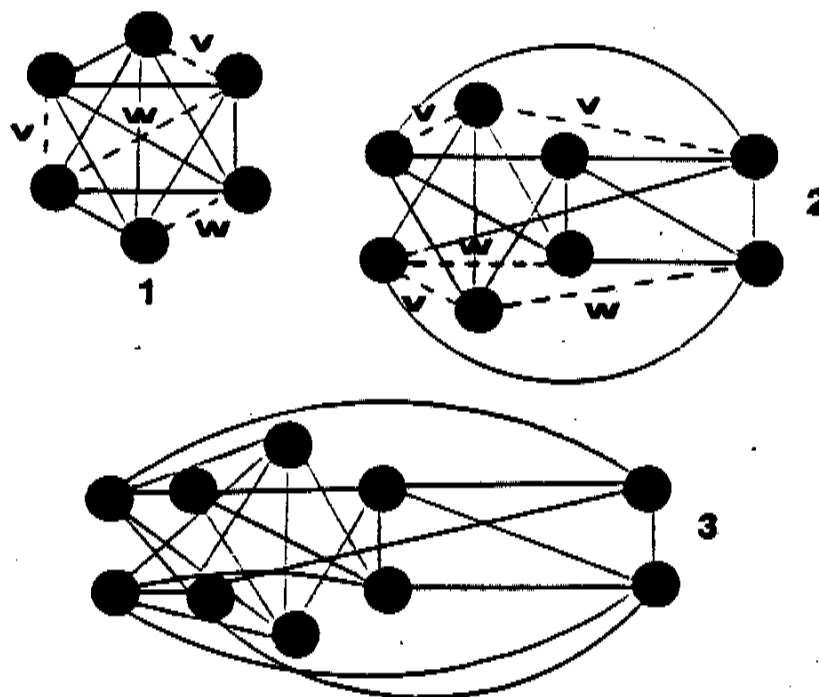


Figure 3-71: ODD-REGULAR in Operation for $k=5$

The edges to be used for the next iteration appear dotted in the figure and are labelled v or w indicating their relationship to the new vertices. The floor for k -regular graphs with k odd is also $\langle P_2, L_Q, \Sigma_2 \rangle$. ODD-REGULAR has three options. The first maintains the degree of every vertex in G while appending two vertices of odd degree. The new vertices v and w are constructed to have $d(v) = d(w) = k$. The second and third options are identical to those for EVEN-REGULAR and

maintain the degree of all previously-existing vertices while adding $k+1$ or $k-1$ vertices, respectively. Thus ODD-REGULAR is correct.

The inverse is computed by:

$$\begin{aligned}
 f^{-1} &= (OM_{vw_1 \dots v_{k+1} w_1 \dots w_{k-1}} + \\
 &\quad A_{y_2 q} A_{y_1 p} D_{y_1 y_2} CM_{y_1 \dots y_{k+1}} D_{pq} + FR_{z_1 \dots z_k})^{-1} \\
 &= OM_{vw_1 \dots v_{k+1} w_1 \dots w_{k-1}}^{-1} + \\
 &\quad (A_{y_2 q} A_{y_1 p} D_{y_1 y_2} CM_{y_1 \dots y_{k+1}} D_{pq})^{-1} + FR_{z_1 \dots z_k}^{-1} \\
 &= OM_{vw_1 \dots v_{k+1} w_1 \dots w_{k-1}}^{-1} + \\
 &\quad A_{pq} CM_{y_1 \dots y_{k+1}} A_{y_1 y_2} D_{y_1 p} D_{y_2 q} + FR_{z_1 \dots z_k}^{-1} \\
 \sigma_{pre} &= \text{distinct } v, w \in V, \text{ distinct } v_1, v_2, \dots, v_{k+1} \in V, \\
 &\quad \text{distinct } x_1, x_2, \dots, x_{k-1} \in V, d(v_i) = d(x_i) = k, \\
 &\quad i = 1, 2, \dots, k; \text{ distinct } v_{2i-1} v_{2i}, w_{2j-1} w_{2j} \in E, \\
 &\quad i = 1, 2, \dots, (k+1)/2; j = 1, 2, \dots, (k-1)/2 \\
 &\quad \text{distinct } p, q \in V, \text{ distinct } y_1, y_2, \dots, y_{k+1} \in V, pq \in E, \\
 &\quad d(p) = k, d(q) = k \\
 &\quad \text{distinct } x_1, x_2, \dots, x_k, z_1 \in V, \text{ distinct } z_2, z_3, \dots, z_k \in V, \\
 &\quad z_1 x_i \in E, i = 1, 2, \dots, k, d(z_i) = k \\
 \sigma^{-1} &= \text{distinct } v, v_1, v_2, \dots, v_k \in V, \text{ distinct } w, w_1, w_2, \dots, w_{k-1} \in V \\
 &\quad vw, vv_i, ww_j \in E, i = 1, 2, \dots, k+1; j = 1, 2, \dots, k-1 \\
 &\quad v_{2i-1} v_{2i}, w_{2j-1} w_{2j} \in E, i = 1, 2, \dots, (k+1)/2; \\
 &\quad j = 1, 2, \dots, (k-1)/2, d(v) = k, d(w) = k, d(v_i) = k, \\
 &\quad i = 1, 2, \dots, k+1, d(w_j) = k, j = 1, 2, \dots, k-1 \\
 &\quad \text{distinct } p, q, y_1, y_2, \dots, y_{k+1} \in V, pq \in V, y_i y_j \in E, d(y_i) = k, \\
 &\quad i = 1, 2, \dots, k+1, j = 1, 2, \dots, k+1 \text{ except } y_1 y_2; \\
 &\quad y_1 p, y_2 q \in E \\
 &\quad \text{distinct } x_i, z_i \in V, x_i z_i \in E, d(x_i) = k, d(z_i) = k, i = 1, 2, \dots, k; \\
 &\quad x_i z_j \in E, j = 2, 3, \dots, k; z_r z_s \in E, r, s = 1, 2, \dots, k
 \end{aligned}$$

The floor shifts to $\langle P_2, L_0, \Sigma_3 \rangle$. Figure 3-72 shows ODD-REGULAR $^{-1}$ operating on a graph $G \in G_p$ and a graph $G \in G_p$ for $k = 3$. The edges to be used in the next

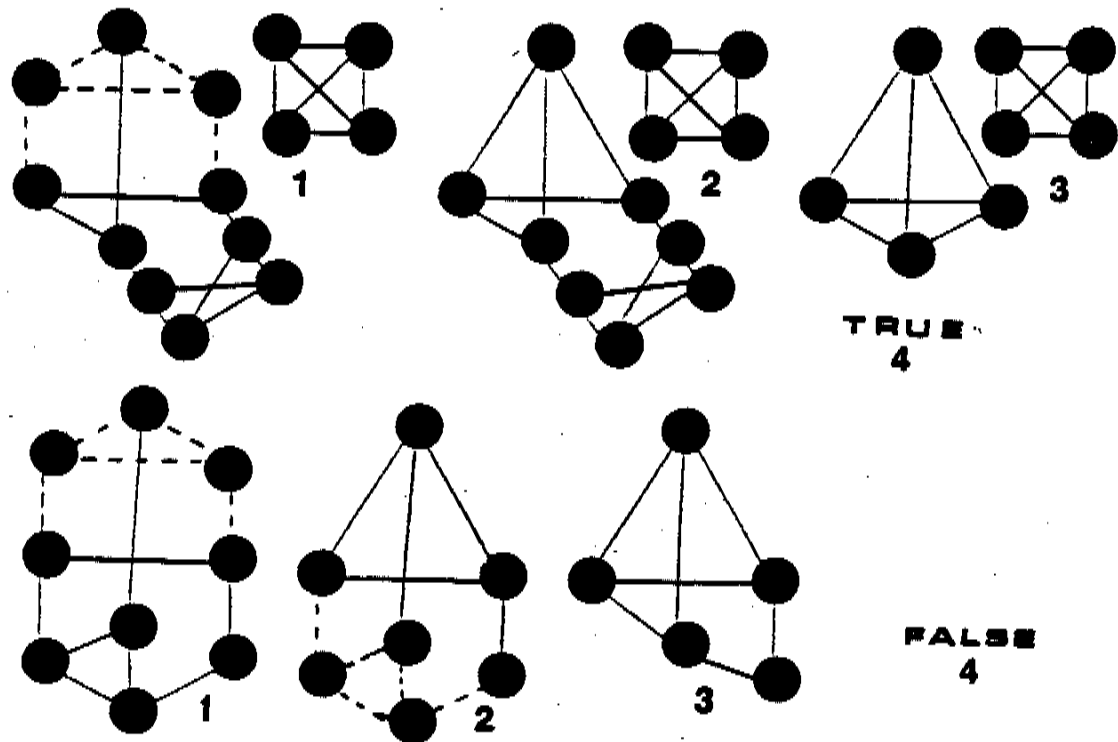


Figure 3-72: ODD-REGULAR^{-1} in Operation for $k = 3$

iteration appear dotted. Any vertex of degree other than k will be unmodifiable and isomorphism will eventually fail in a graph containing such a vertex. Under the first option, ODD-REGULAR^{-1} removes pairs of degree k vertices v and w and replaces $2k-1$ edges with $k-1$ edges, maintaining the regularity of the v_i 's and the w_i 's, until K_{k+1} is reached in each connected component. The second and third options behave as they did in EVEN-REGULAR^{-1} , maintaining the degree of all other vertices while deleting $k+1$ or $k-1$ vertices, respectively. ODD-REGULAR^{-1} is clearly correct for $G \in G_p$ and has been confirmed correct for $k = 3$ against [Statman 82]. We believe it to be correct for $k > 3$ although a formal proof will be offered elsewhere. At this writing no $G \in G_p$ has been shown inaccessible, and we will postulate ODD-REGULAR^{-1} to be correct and thus ODD-REGULAR to be complete.

3.7.21. Connected Graphs

A graph $G = \langle V, E \rangle$ is *connected* if for every pair of vertices in V there is a path constructable between them using only edges in E . Several examples of connected graphs appear in Figure 3-73.

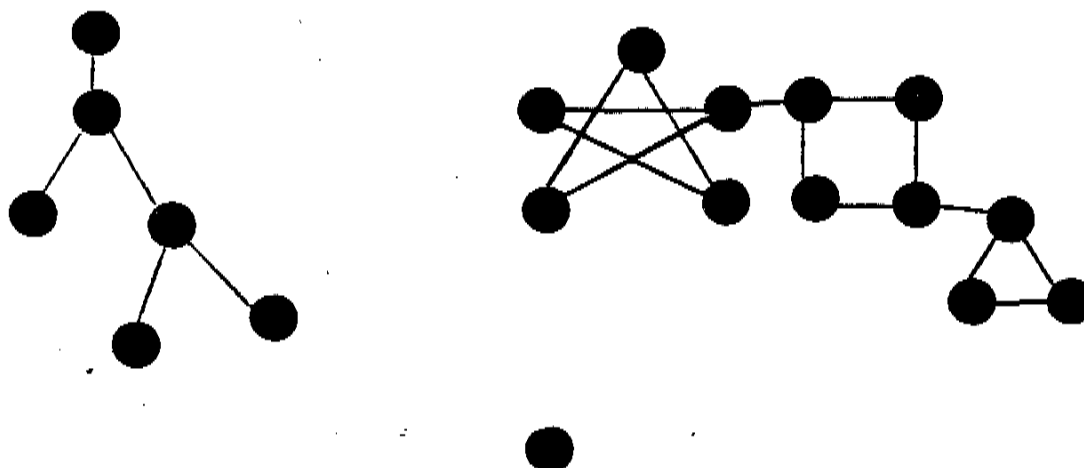


Figure 3-73: Some Connected Graphs

There are many ways to write an R-property for connectivity. One reasonably obvious form is

$$B_{xy}^*(K_1) \text{ where } x \in V$$

We prefer a more complex statement which will relate connectivity to other properties. Our formulation for the R-property CONNECTED is

$$((N + A_{xv_1}) \dots (N + A_{xv_r}) A_{xy} F_{xv_1 \dots v_r})^*(K_1)$$

where distinct $x, v_i \in V$, $y \in V$, $xv_i \in E$, $0 \leq r \leq d(x)$

Figure 3-74 shows the iterative steps in a sample run of CONNECTED. The floor is $\langle P_4, L_1, \Sigma_5 \rangle$.

An iteration of CONNECTED begins by fragmenting vertex x into vertices x and y and forces y adjacent to x . The iteration requires y to assume r (≥ 0) of x 's adjacencies to the v_i 's and permits x to retain any or all of those adjacencies as

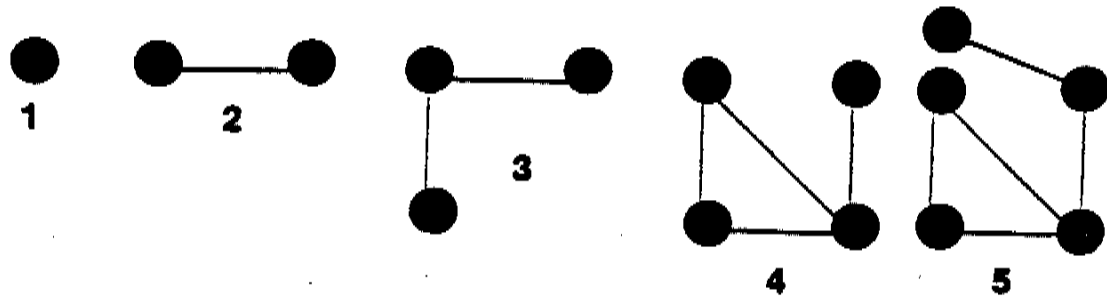


Figure 3-74: CONNECTED in Operation

well. Because x retains all its previous adjacencies through E or through the edge xy , and because each newly-introduced vertex y has access to the remainder of the graph via the edge xy to its originating vertex x , CONNECTED is correct.

The inverse CONNECTED^{-1} is computed by:

$$\begin{aligned}
 f^{-1} &= ((N + A_{xv_1}) \dots (N + A_{xv_r}) A_{xy} F_{xyv_1 \dots v_r})^{-1} \\
 &= F_{xyv_1 \dots v_r}^{-1} A_{xy}^{-1} (N + A_{xv_r})^{-1} \dots (N + A_{xv_1})^{-1} \\
 &= I_{xy} D_{xy} (N + D_{xv_r}) \dots (N + D_{xv_1}) \\
 \sigma_{\text{pre}} = \sigma &= \text{distinct } x, v_i \in V, y \in V, xv_i \in E, 0 \leq r \leq d(x) \\
 \sigma^{-1} &= x, y, v_i \in V, xy, yv_1, \dots, yv_r \in E, 0 \leq r \leq d(x)
 \end{aligned}$$

The floor remains constant. Figure 3-75 shows CONNECTED^{-1} operating on a graph $G \in G_p$ and a graph $G \in G_p$. CONNECTED^{-1} collapses each connected component of a graph into a single vertex. If G is connected, the result will be K_1 and success; if G is not connected the result will be a set of at least two isolated vertices, which will fail. Thus CONNECTED^{-1} is correct and CONNECTED is complete.

3.7.22. Biconnected Graphs

A graph $G = \langle V, E \rangle$ is *biconnected* if there is no vertex in V whose deletion (with all its associated edges) disconnects the graph. Several examples of biconnected graphs appear in Figure 3-76. There are several ways to write an R-property for biconnectivity. We choose one closely related to the formulation for connectivity. The R-property BICONNECTED is

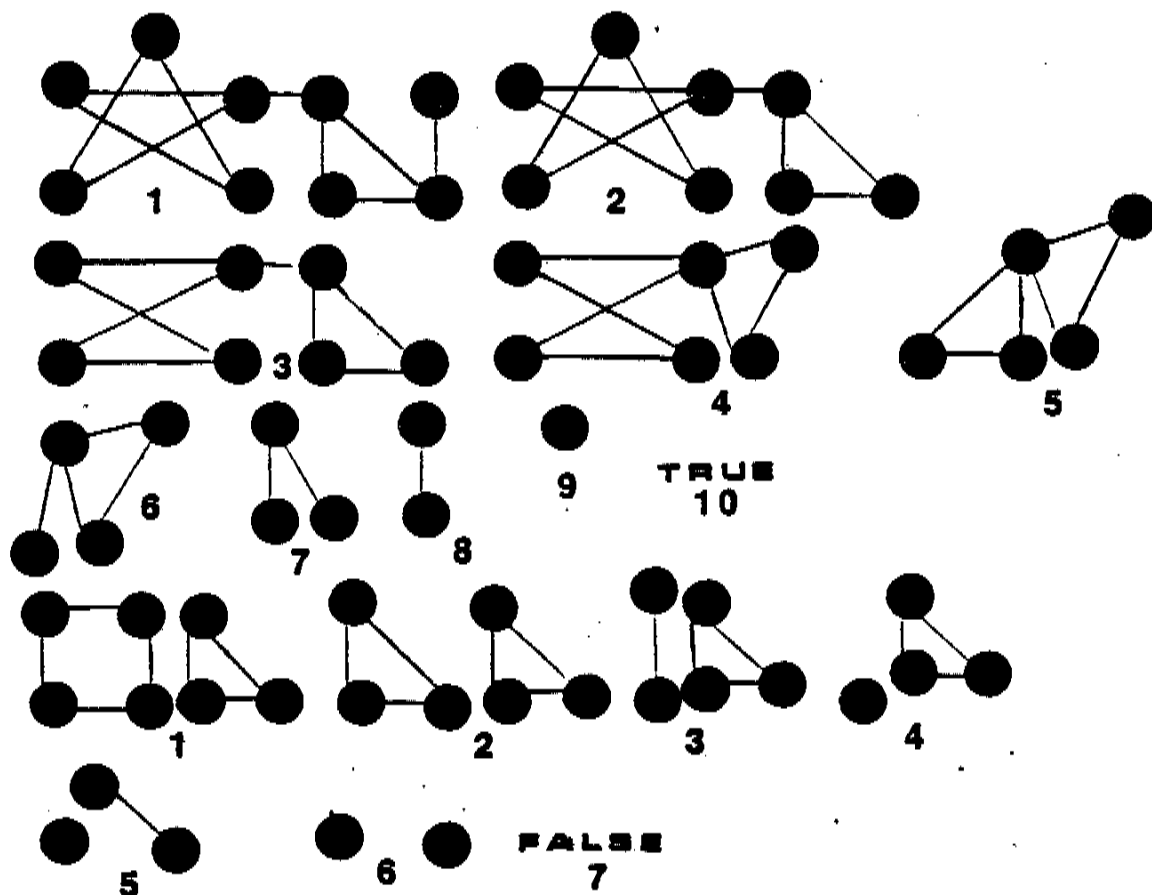
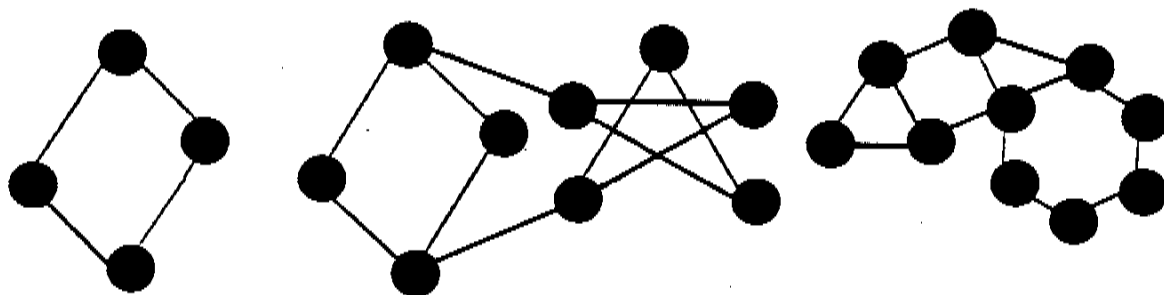
Figure 3-75: CONNECTED^{-1} in Operation

Figure 3-76: Some Biconnected Graphs

$$((N + A_{xv_1}) \dots (N + A_{xv_r}) A_{xy} F_{xyv_1 \dots v_r} + (N + A_{wt_1}) \dots (N + A_{wt_s}) A_{wt_{s+1}} A_{wz} F_{wzt_1 \dots t_{s+1}})^*(K_3)$$

where distinct $x, v_i \in V$; $y \in V$, $xv_i \in E$, $1 \leq i \leq d(x)-1$

distinct $w, t_i \in V$, $z \in V$, $wt_i \in E$, $s = d(w)-1$

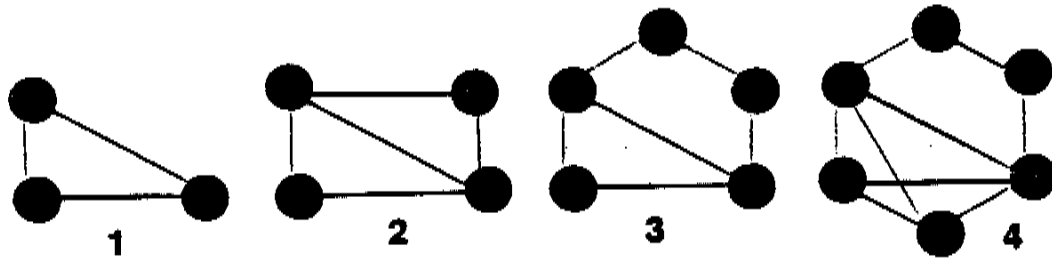


Figure 3-77: BICONNECTED in Operation

Figure 3-77 shows the iterative steps in a sample run of BICONNECTED. The floors are $\langle P_4, L_2, \Sigma_5 \rangle$ and $\langle P_4, L_{1n}, \Sigma_5 \rangle$.

An iteration of BICONNECTED behaves exactly like an iteration of CONNECTED, except that the new vertex must receive at least one of the old vertex's neighbors in the fragmentation, and the old vertex must retain at least one of its neighbors. (This is indicated by BICONNECTED's two options; the first leaves some neighbor of x untouched, while the second shares a neighbor t_{s+1} between w and z). A graph with $n > 2$ is biconnected if and only if every pair of vertices lie on a common cycle. After an iteration of BICONNECTED, if edge xv_1 is not present, xv_1 may be substituted to produce all the previously existing cycles. Thus the graph will still be biconnected and BICONNECTED is correct.

The inverse BICONNECTED⁻¹ is computed similarly to that for CONNECTED:

$$\begin{aligned}
 f^{-1} &= ((N + A_{xv_1}) \dots (N + A_{xv_r}) A_{xy} F_{xyv_1 \dots v_r} + (N + A_{wt_1}) \dots \\
 &\quad (N + A_{wt_s}) A_{wt_{s+1}} A_{wz} F_{wzt_1 \dots t_{s+1}})^{-1} \\
 &= ((N + A_{xv_1}) \dots (N + A_{xv_r}) A_{xy} F_{xyv_1 \dots v_r})^{-1} + ((N + A_{wt_1}) \dots \\
 &\quad (N + A_{wt_s}) A_{wt_{s+1}} A_{wz} F_{wzt_1 \dots t_{s+1}})^{-1} \\
 &= I_{xy} D_{xy} (N + D_{xv_r}) \dots (N + D_{xv_1}) + I_{wz} D_{wz} D_{wt_{s+1}} (N + D_{wt_s}) \dots \\
 &\quad (N + D_{wt_1}) \\
 \sigma_{pre} &= \text{distinct } x, v_i \in V, y \in V, xv_i \in E, 1 \leq r \leq d(x) - 1, \\
 &\quad d(x) > 1, d(v_i) > 1, i=1, 2, \dots, r \\
 &\quad \text{distinct } w, t_i \in V, z \in V, wt_i \in E, s = d(w) - 1, d(w) > 1,
 \end{aligned}$$

$$\begin{aligned}
& d(t_i) > 1, \quad i=1,2,\dots,s \\
\sigma^{-1} & = \text{distinct } x, y, v_i \in V, \quad xy, yv_1, \dots, yv_r \in E, \quad 1 \leq r, \\
& d(x) > 1, \quad d(y) > 1, \quad d(v_i) > 1, \\
& |\{pv_i \mid p \in V, \quad pv_i \in E, \quad p \neq x, \quad p \neq y\}| > 0, \\
& i=1,2,\dots,r \\
& \text{distinct } w, z, t_i \in V, \quad wz, zt_1, \dots, zt_{s+1}, wt_{s+1} \in E, \quad d(w) > 1, \\
& d(z) > 1, \quad d(t_i) > 1, \quad s = d(w) - 1, \\
& |\{pt_i \mid p \in V, \quad pt_i \in E, \quad p \neq x, \quad p \neq y\}| > 0, \quad i=1,2,\dots,s+1
\end{aligned}$$

The floor remains constant. The inversion procedure has noted that the degree of each $v_i(t_i)$ which $y(z)$ acquires will be at least two, and each $v_i(t_i)$ will be adjacent to some vertex other than x and y (w and z). Figure 3-78 shows BICONNECTED^{-1} operating on a graph $G \in G_p$ and a graph $G \in G_p$. BICONNECTED^{-1} collapses each block of a graph into K_3 . A disconnected graph will continue to reduce to a set of disjoint chains, a connected but not biconnected graph to a chain of length two, and a biconnected graph to K_3 . Thus BICONNECTED^{-1} is correct and BICONNECTED is complete.

3.7.23. k -Connected Graphs

A graph $G = \langle V, E \rangle$ is k -connected if there is no set of $k-1$ vertices in V whose deletion (with all their associated edges) disconnects the graph. Two examples of 5-connected graphs appear in Figure 3-79. Note that 1-connected is equivalent to connected and that 2-connected is equivalent to biconnected. The somewhat awkward constructions of the two previous sections are now seen to be special cases of the R-property K -CONNECTED:

$$\begin{aligned}
& ((N + A_{xv_1}) \dots (N + A_{xv_r}) A_{xy} F_{xv_1 \dots v_r} + (N + A_{wt_1}) \dots (N + A_{wt_{s-k+1}}) A_{wt_{s-k+2}} \dots \\
& A_{wt_{s+1}} A_{wz} F_{wzt_1 \dots t_{s+1}})^*(K_{k+1}) \\
& \text{where distinct } x, v_i \in V, \quad y \in V, \quad xv_i \in E, \quad k-1 \leq r \leq d(x) - k+1 \\
& \text{distinct } w, t_i \in V, \quad z \in V, \quad wt_i \in E, \quad s = d(w) - k+1
\end{aligned}$$

Figure 3-80 shows the iterative steps in a sample run of K -CONNECTED for $k = 4$. The floor for K -CONNECTED is $\langle P_4, L_{10}, \Sigma_5 \rangle$.

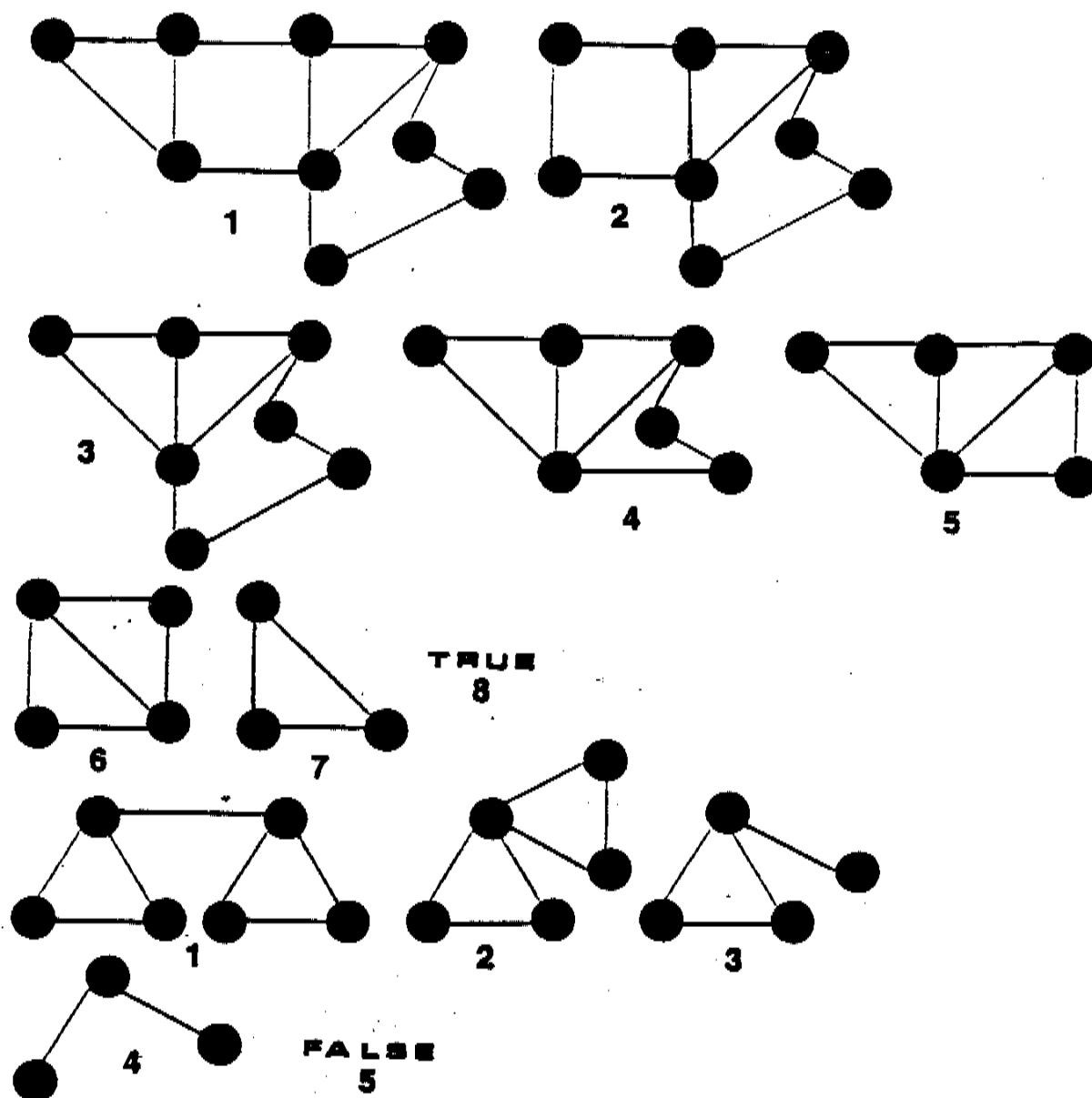


Figure 3-78: $BICONNECTED^{-1}$ in Operation

An iteration of K -CONNECTED behaves exactly like an iteration of $BICONNECTED$, except that the new vertex must receive at least $k-1$ of the old vertex's neighbors in the fragmentation, and the old vertex must retain at least $k-1$ of its neighbors. The removal of $k-1$ vertices from K_{k+1} leaves K_2 . Thus the seed

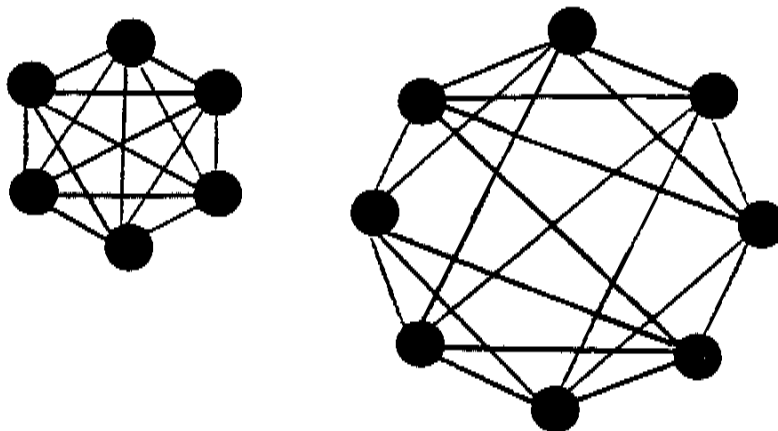


Figure 3-79: Some 5-Connected Graphs

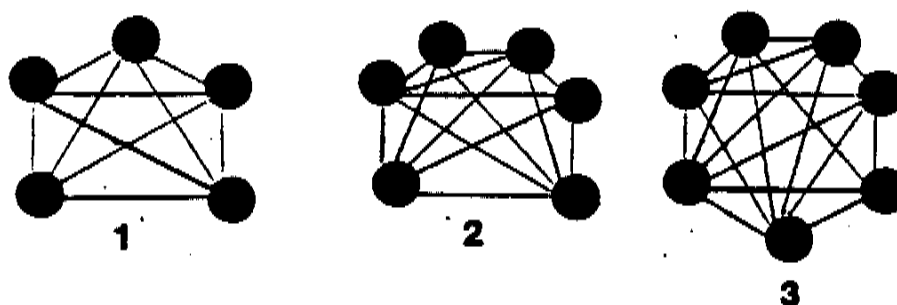


Figure 3-80: 4-CONNECTED in Operation

is k -connected. Suppose that G is a k -connected graph but a single iteration of K -CONNECTED on G results in G' which is not k -connected. Then G' must contain $k-1$ vertices $w_1, \dots, w_{k-1}$, whose deletion will disconnect G' . Certainly the new vertex is among them or the same vertices would have disconnected G . Thus there are vertices t and u in G' which have no path between them once the new vertex is removed. The deletion of the old vertex in G should have had the same effect, however. Thus our supposition is incorrect, G' is k -connected and K -CONNECTED is correct.

The inverse $K\text{-CONNECTED}^{-1}$ is computed similarly to that for $B\text{CONNECTED}$:

$$\begin{aligned}
 f^{-1} &= ((N + A_{xv_1}) \dots (N + A_{xv_r}) A_{xy} F_{xyv_1 \dots v_r} \\
 &\quad + (N + A_{wt_1}) \dots (N + A_{wt_{s-k+2}}) A_{wt_{s-k+3}} \dots A_{wt_{s+1}} \\
 &\quad A_{wz} F_{wzt_1 \dots t_{s+1}})^{-1}
 \end{aligned}$$

CHAPTER 4

ADVANCED TOPICS IN RECURSIVE LANGUAGES

The essential characteristic of reasoning by recurrence is that it contains, condensed, so to speak, in a single formula, an infinite number of syllogisms.

--Poincare

This chapter considers an assortment of advanced topics in recursive graph property languages. The first section extends R-properties by access to a register and contrasts Σ_2 with Σ_3 . The second section explores loop marking and contrasts Σ_3 with Σ_4 . The third section discusses loop labelling and demonstrates properties available with it. Subsequent sections are devoted to graphs with more elaborate labels, subsumption, merger and NP-completeness.

4.1. Extended Recursive Languages

By enlarging the input and slightly modifying the interpretation, this section extends R-properties to R^+ -properties, motivated by the calculation of n and m . An application to the selector languages Σ_2 and Σ_3 is given.

Imagine an algorithm, similar to the ones we used for R-properties, with a register which tallies the number of algorithmic iterations and outputs both the register value and the graph. Such algorithms will be for properties associated with an integer value. More formally, we define an R^+ -property as the following semantic interpretation of the triple $p = \langle f, S, o \rangle$ as a recursive algorithm, called on $(G, 0)$ for any graph G described by S :

$p(G, k)$ $= (G, k)$ if enough
 $= p(G', k+1)$ where $G' = f(g)$ using elements from G selected

by σ in order to apply f

At the end of each iteration, the graph G has the R^+ -property with value k . Note that k is incidental to p . The definition of an R^+ -property is independent of the value of k .

4.1.1. Calculating the Number of Vertices and Edges in a Graph

As a first example of an R^+ -property, we offer VERTICES to construct graphs with a known number of vertices. In section 3.7.14 we had K -VERTICES which operated for fixed k . Now we define VERTICES as:

$$(A_{xy}^* A_z^*)^*(\langle \emptyset, \emptyset \rangle, 0) \text{ where distinct } x, y \in V, z \in V$$

Figure 4-1 shows the iterative steps in a sample run of VERTICES.

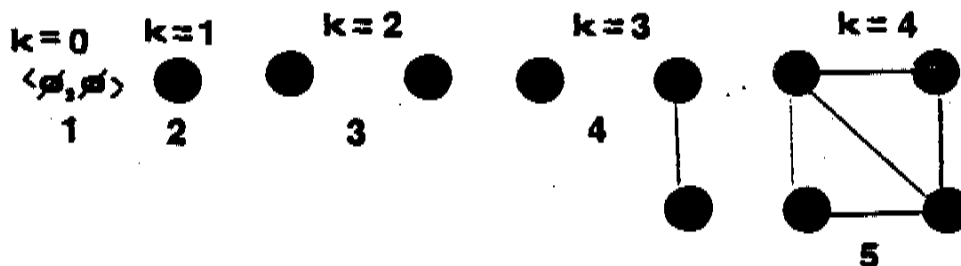


Figure 4-1: VERTICES in Operation

Note that on each iteration any number of edges (including zero) may be added and exactly one vertex must be added. The floor for VERTICES is $\langle P_1, L_1, \Sigma_1 \rangle$.

Similarly we can define the R^+ -property EDGES to construct graphs with a known number of edges. In section 3.7.15 we had K -EDGES which operated for fixed k . Now we define EDGES as:

$$(A_{yz}^* A_x^*)^*(K_1, 0) \text{ where distinct } y, z \in V, yz \in E$$

Figure 4-2 shows the iterative steps in a sample run of EDGES. Note that on each iteration any number of vertices may be added and exactly one edge must be added. The floor for EDGES is $\langle P_1, L_1, \Sigma_1 \rangle$ also.

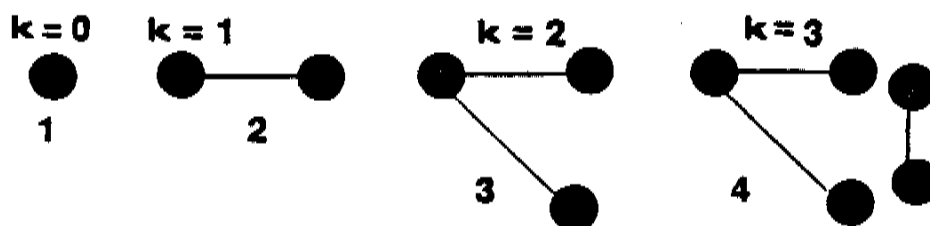


Figure 4-2: EDGES in Operation

Now we require an inverse for an R^+ -property. This inverse should be a tester which, given an input graph G and register value k , attempts to restore G to a seed graph in S and k to zero, counting its iterations in the register. More formally, a terminal R^+ -expression $p = \langle f^{-1}, S, \sigma^{-1} \rangle$ is said to be the inverse of another R^+ -expression $p = \langle f, S, \sigma \rangle$ if and only if the testing semantic interpretation returns (TRUE,0) on all outputs of the generator which is the R^+ -property defined by p , and (FALSE, k) on all other graphs. The testing semantic interpretation of $p^{-1} = \langle f^{-1}, S, \sigma^{-1} \rangle$ is the following recursive algorithm:

$p^{-1}(G,k)$ = (TRUE,0) if $k = 0$ and G is described by S
 = $f^{-1}(G',k-1)$ where $G' = f^{-1}(G)$ using
 elements from G selected by σ^{-1} in order to apply f^{-1}
 = (FALSE, k) if G is not described by S and
 σ^{-1} is not applicable
 = (FALSE, k) if G is not described by S and
 $k = 0$
 = (FALSE, k) if G is described by S and $k \neq 0$

First p^{-1} checks to see if G has returned to S and k is zero, in which case the algorithm terminates, returning (TRUE,0). Otherwise, p^{-1} attempts to iterate by finding suitable vertices and edges for Σ^{-1} . If successful termination and iteration are both impossible, p^{-1} terminates, returning (FALSE, k). Failure is caused by G in S with a non-zero k , by G not in S with a zero k , or by G not in S with an unmatchable σ^{-1} .

The automated calculation of an inverse for the sample R^+ -properties we have

shown is complicated by their $(f^*g)^*$ format. In our attempt to return to S, we may not iterate f enough, masking what should be successful results. Thus we will define an extreme superscript e , in order to force the most iterations of f^{-1} possible, i.e., $(f^{-1})^e$ will be interpreted as "do f^{-1} as many times as σ^{-1} will permit." This will avoid under-iterating f^{-1} . We therefore add a new rule to those already existing for R-property inversion:

RULE 6

The inverse of a function occurring an unknown number of times is as many iterations as possible of its inverse. $(f^*)^{-1} = (f^{-1})^e$

We construct inverses now for VERTICES and EDGES. For VERTICES^{-1} we have:

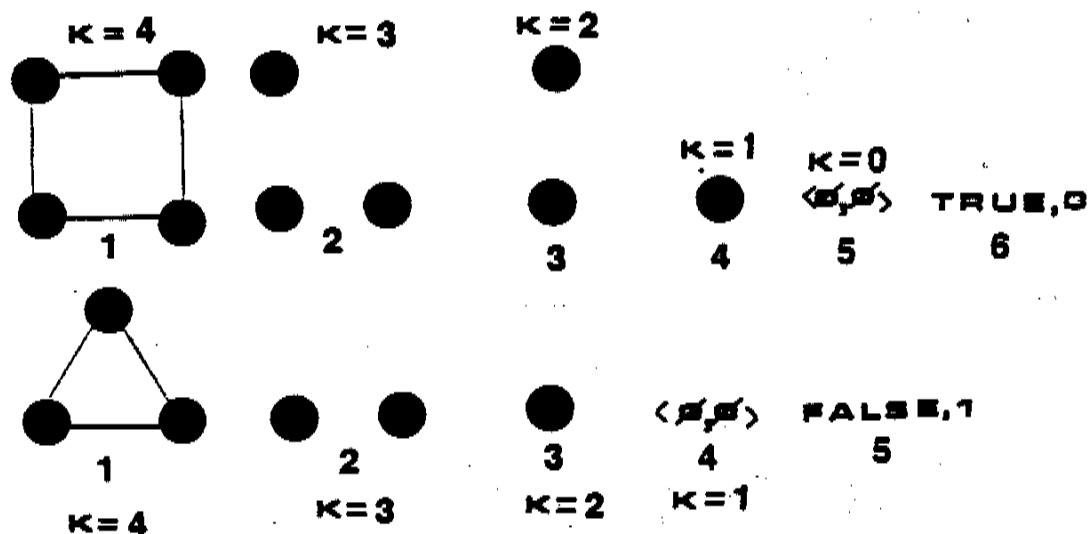
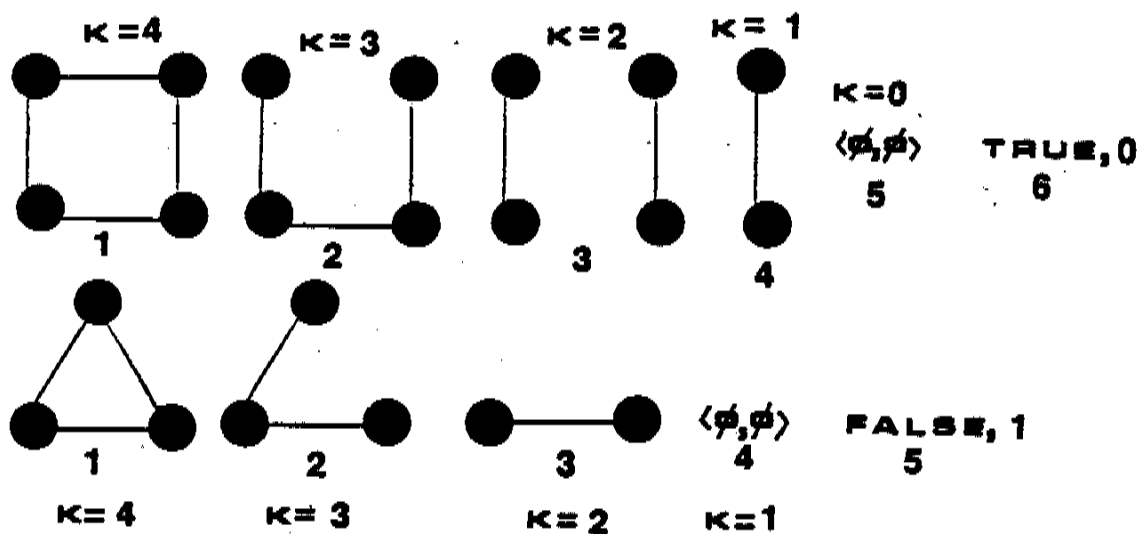
$$\begin{aligned}
 f^{-1} &= (A_{xy}^* A_z)^{-1} \\
 &= A_z^{-1} (A_{xy}^*)^{-1} \\
 &= A_z^{-1} (A_{xy}^{-1})^e \\
 &= D_z D_{xy}^e \\
 \sigma_{\text{pre}} &= \text{distinct } x, y \in V, z \in V, xy \in E \\
 \sigma^{-1} &= \text{distinct } x, y, z \in V, xy \in E, d(x) = 0
 \end{aligned}$$

The floor shifts to $\langle P_2, L_1, \Sigma_3 \rangle$. Figure 4-3 shows VERTICES^{-1} operating on two graphs, one with a correct k value and one with an incorrect k value.

For EDGES^{-1} we have:

$$\begin{aligned}
 f^{-1} &= (A_{yz} A_x^*)^{-1} \\
 &= (A_x^*)^{-1} A_{yz}^{-1} \\
 &= (A_x^{-1})^e A_{yz}^{-1} \\
 &= D_x^e D_{yz} \\
 \sigma_{\text{pre}} &= \text{distinct } y, z \in V, x \in V, yz \in E \\
 \sigma^{-1} &= \text{distinct } x, y, z \in V, yz \in E, d(x) = 0
 \end{aligned}$$

Again the floor shifts to $\langle P_2, L_1, \Sigma_3 \rangle$. Figure 4-4 shows EDGES^{-1} operating on two graphs, one with a correct k value and one with an incorrect k value.

Figure 4-3: VERTICES^{-1} in OperationFigure 4-4: EDGES^{-1} in Operation

4.1.2. Calculating the Degree of a Vertex

The R^+ -property "has vertex v of degree k " is the concept used to extend Σ_2 to Σ_3 . This property, DEGREE , may be stated as:

$$(A_{vw}(A_x + A_{yz})^*)^* \langle \{v\}, \phi, 0 \rangle \text{ where distinct } v, w, y, z \in V, x \in V, vw \in E$$

Figure 4-5 shows the iterative steps in a sample run of DEGREE . Note that on each iteration any number of vertices and edges may be added and exactly one

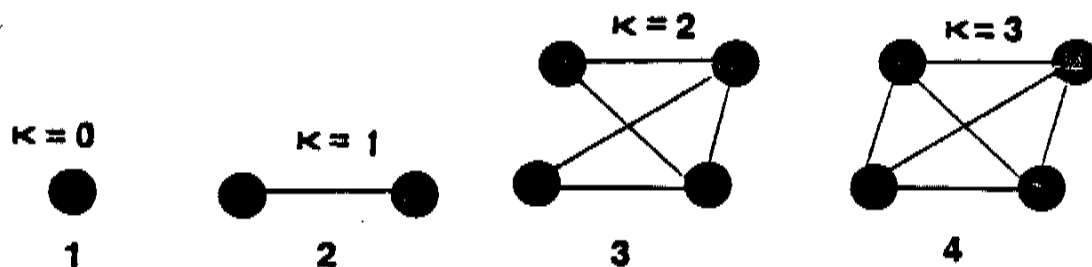


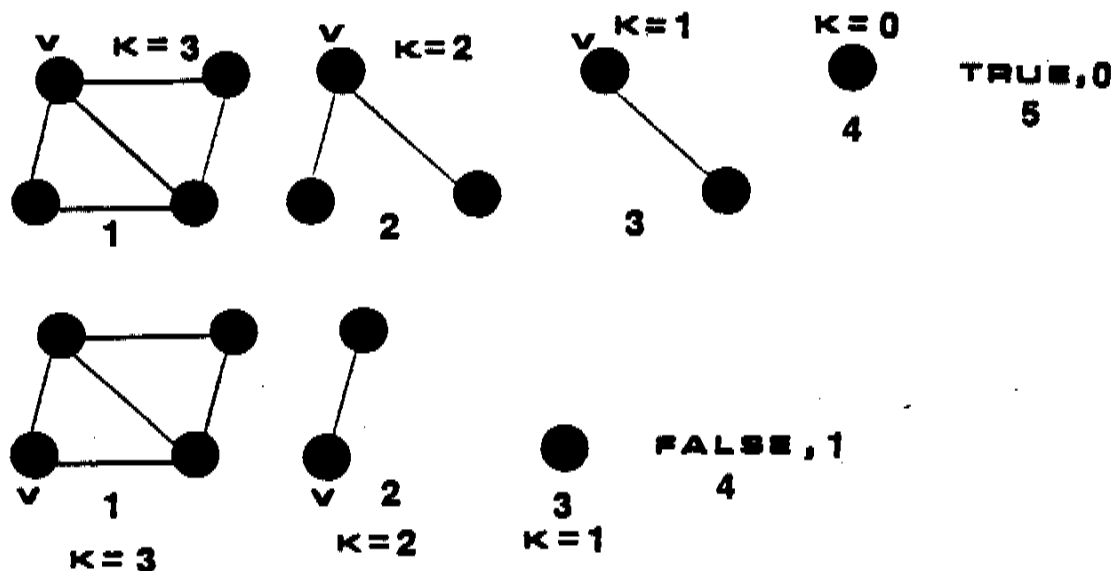
Figure 4-5: DEGREE in Operation

edge must involve vertex v . The floor for DEGREE is $\langle P_1, L_1, \Sigma_2 \rangle$. The inverse, DEGREE^{-1} , is computed by:

$$\begin{aligned}
 f^{-1} &= (A_{vw}(A_x + A_{yz})^*)^{-1} \\
 &= ((A_x + A_{yz})^*)^{-1} A_{vw}^{-1} \\
 &= ((A_x + A_{yz})^{-1})^* A_{vw}^{-1} \\
 &= (A_x^{-1} + A_{yz}^{-1})^* A_{vw}^{-1} \\
 &= (D_x + D_{yz})^* D_{vw} \\
 \sigma_{\text{pre}} = \sigma &= \text{distinct } v, w, y, z \in V, x \in V, yz, vw \in E \\
 \sigma^{-1} &= \text{distinct } x, v, w, y, z \in V, yz, vw \in E, d(x) = 0
 \end{aligned}$$

The floor shifts to $\langle P_2, L_1, \Sigma_3 \rangle$. Figure 4-6 shows DEGREE^{-1} operating on two graphs, one with a correct k value and one with an incorrect k value.

Clearly any $\sigma \in \Sigma_3 - \Sigma_2$ references the degree of a vertex v . Such a procedure may be thought of as calling DEGREE^{-1} on (G, n) and interpreting the output (FALSE, k) to mean that v had degree $n-k$. Of course such calls could be inefficient; it might be more economical to use $O(n)$ (one register for each vertex) storage and calculate the degree of all vertices in $O(n + m)$ iterations. As an R-property in $\langle P, L, \Sigma_3 \rangle$ iterates, it could update the degrees of the vertices at $O(i)$ cost, where i is the number of iterations. In the worst case, $O(m) = O(n^2)$ and $O(m + n) = O(n^2)$; in the best case, $m = 0$ and $O(m + n) = O(n)$. Thus an R-property whose floor requires Σ_3 rather than Σ_2 represents an additional complexity of between $O(ni)$ and $O(n^2i)$ for a graph on n vertices requiring i iterations for

Figure 4-6: DEGREE⁻¹ in Operation

generation or testing. The value of i is property-dependent.

4.2. The Loop as Marker

R^+ -properties were one way of extending our recursive formulation. In this section we explore a different extension, a marking technique using loops, motivated by the calculation of $\max$. A comparison of Σ_3 and Σ_4 is made.

4.2.1. Calculating the Maximum Vertex Degree in a Graph

All work in this segment is for *directed* graphs only. In order to apply this algorithm to an undirected graph G , transform every undirected edge between x and y into two directed edges, xy and yx . Any $\sigma \in \Sigma_4 - \Sigma_3$ references $\max$, the maximum vertex degree in the graph. In 3.7.17 we had MAX-K which operated for fixed k . Now we define the R^+ -property MAX as:

$$(A_x^*(D_{yy}A_{yz})^n L)^*(E_p, 0) \text{ where distinct } y, z \in V, yy \in E, yz \in E$$

On each iteration, MAX places a loop (L) on every vertex, *marking* those which have not yet had their out degrees increased. Then, n times, MAX selects a vertex y with a loop on it, adds an edge from y to some other vertex z , and removes y 's loop. (The selector operates *after* the application of L .) Finally, MAX adds zero or more vertices to the graph and increments k , completing a single iteration. Figure

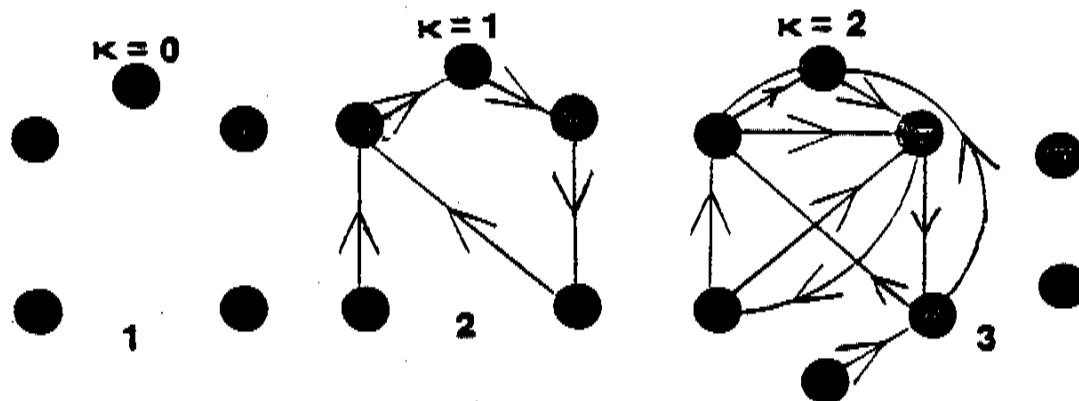


Figure 4-7: MAX in Operation

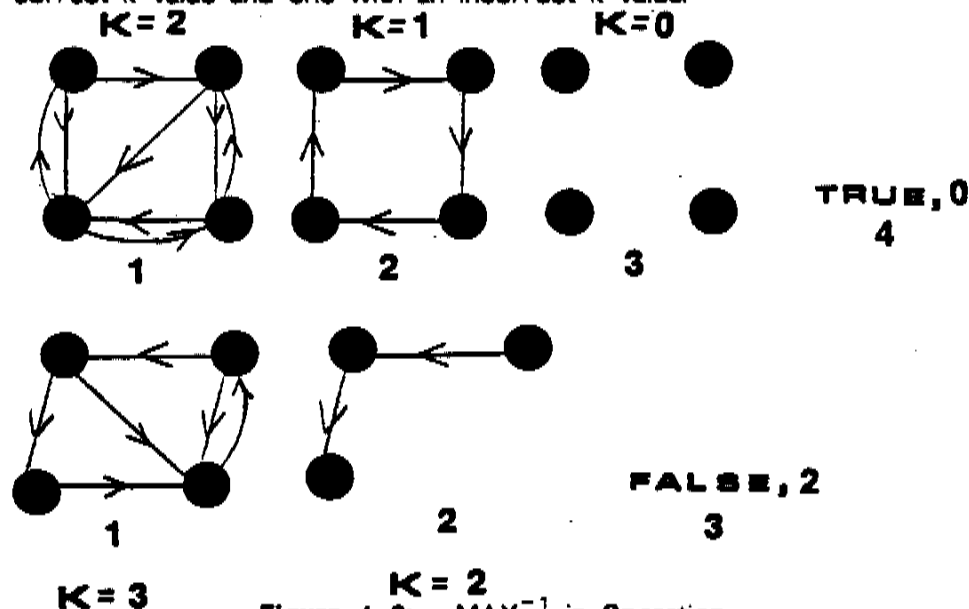
4-7 shows the iterative steps in a sample run of MAX with $p = 5$. After i iterations, each of the p vertices initially in the graph will have out-degree k . Those vertices added on the first iteration will be of out-degree $k-1$, those added on the second of out-degree $k-2$, and so on. The floor is $\langle P_3, L_1, \Sigma_2 \rangle$. This is our only algorithm in which f is dependent on the size of the graph. An alternative formulation without loops would be more difficult to follow. We therefore permit this construction, albeit with reservations.

We have already noted that the loops are used as uniform markers. "L" may be interpreted as "we are going to do this to every vertex." At any intermediate point, say after $(D_{yy} A_{yz})^i$ where $i < n$, those vertices with loops have not yet acquired a new out-edge. Thus the loop marks the vertex in a context known to the algorithm. Under inversion we expect loop markers to continue as adequate. The inverse MAX^{-1} is computed by:

$$\begin{aligned}
 f^{-1} &= (A_x^* (D_{yy} A_{yz})^n L)^{-1} \\
 &= L^{-1} ((D_{yy} A_{yz})^n)^{-1} (A_x^*)^{-1} \\
 &= L^{-1} ((D_{yy} A_{yz})^{-1})^n (A_x^{-1})^n \\
 &= L^{-1} (A_{yz}^{-1} D_{yy}^{-1})^n (A_x^{-1})^n \\
 &= L (D_{yz} A_{yy})^n D_x^* \\
 \sigma_{\text{pre}} &= \text{distinct } y, z \in V, x \in V, yy \in E, yz \in E \\
 \sigma^{-1} &= \text{distinct } x, y, z \in V, yy \in E, yz \in E, d(x) = 0
 \end{aligned}$$

By $d(x) = 0$, we mean both the in degree and the out-degree of x are zero. The

floor shifts to $\langle P_3, L_1, \Sigma_3 \rangle$. Figure 4-8 shows MAX^{-1} operating on two graphs, one with a correct k value and one with an incorrect k value.



In MAX^{-1} the loops mark those vertices which have already had an out-edge deleted until all (n) vertices have loops, at which time all loops are removed (L). Thus the contextual significance of a loop has changed (from "needs a new out-edge" in MAX to "has lost an out-edge" in MAX^{-1}). What remains constant is the loop (or absence of a loop) as a partitioning of the vertices into "already processed" and "to be processed". Any application of loops to the vertices of G creates a partition on V . Such a partition may be exploited in various ways. When all vertices are looped, and then gradually unlooped in a single iteration, we will say that we are *loop marking*.

An algorithm using $\sigma = \Sigma_4 - \Sigma_3$ clearly references $\max$. Such a procedure may be thought of as calling MAX^{-1} on (G, n) and interpreting the output $(FALSE, k)$ to mean that $\max = n - k + 1$. Of course such calls could be inefficient; it might be more economical to use $O(n)$ storage to maintain the number of vertices of each degree. For an initial set-up cost of $O(m)$ time, the value of $\max$ will be available as long as the algorithm executes. Σ_4 therefore represents an additional complexity

of $O(m + n)$ over Σ_3 .

4.3. The Loop as Label

Loops can be used as other than markers. This section demonstrates another extension to our recursive formulation, the use of loops as labels. Motivation is provided from bipartite graphs, and examples of other loop-labelled properties are provided.

4.3.1. Bipartite Graphs

Several examples of bipartite graphs appear in Figure 4-9.

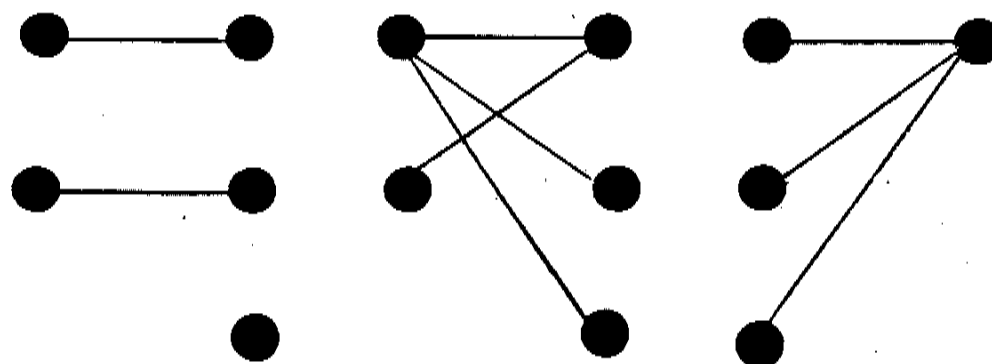


Figure 4-9: Some Bipartite Graphs

The R-property BIPARTITE is:

$$(A_x + A_{xx}A_x + A_{yz})^*(\langle\{1,2\},\{11\}\rangle) \text{ where } y,z \in V, x \in V, \\ |\{yy,zz\} \cap E| = 1$$

Figure 4-10 shows the iterative steps in a sample run of BIPARTITE. In L_1 , $\langle\{1,2\},\{11\}\rangle$ is characterized by $E \cap \underline{1} = 0$ and $E \cap \bar{1} \neq 0$, but so is any edgeless graph with some loops. In L_2 the seed is uniquely characterized as:

$$E \cap \underline{1} = 0 \\ E \cap \underline{1} \sim E \cap \bar{1} \\ E \cap \bar{1} \sim E \cap 1$$

In L_{1n} the seed is uniquely characterized as $E \cap \underline{1} = 0$, $E \cap \bar{1} \neq 0$ and $n = 2$. Thus the floors for bipartite graphs are $\langle P_1, L_{1n}, \Sigma_5 \rangle$ and $\langle P_1, L_2, \Sigma_5 \rangle$. The seed graph makes the partition of the vertices explicit: those vertices with loops are in one

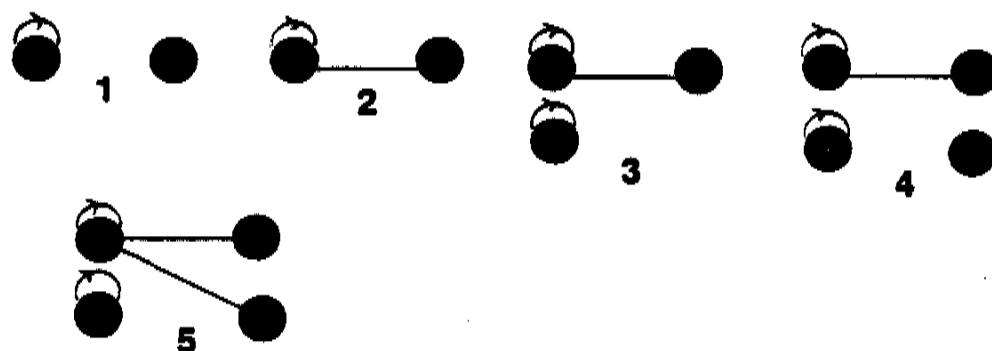


Figure 4-10: BIPARTITE in Operation

class, those vertices without loops in the other. New vertices have their class specified (by the presence or absence of a loop) when they are added to V (A_x or $A_{xx}A_x$). An edge may only be added between a vertex with a loop and vertex without a loop. The final output is a bipartite graph whose partition is clearly labelled by its loops. This *loop labelling* is different from the loop marking technique of the previous section. Loop marking is temporary, for uniform processing within an iteration. Loop labelling is retained from one iteration to the next.

The inverse $BIPARTITE^{-1}$ is computed by:

$$\begin{aligned}
 f^{-1} &= (A_x + A_{xx}A_x + A_{yz})^{-1} \\
 &= A_x^{-1} + (A_{xx}A_x)^{-1} + A_{yz}^{-1} \\
 &= A_x^{-1} + A_x^{-1}A_{xx}^{-1} + A_{yz}^{-1} \\
 &= D_x + D_xD_{xx} + D_{yz} \\
 \sigma_{pre} &= y, z \in V, x \in V, yz \in E, |\{yy, zz\} \cap E| = 1 \\
 \sigma^{-1} &= x, y, z \in V, yz \in E, |\{yy, zz\} \cap E| = 1, d(x) < 1, \\
 &\quad |\{p | p \in V, d(p)=0\}| > 1, \\
 &\quad |\{p | p \in V, pp \in E, d(p) < 1\}| > 1.
 \end{aligned}$$

The floors shift to $\langle P_2, L_{1n}, \Sigma_5 \rangle$ and $\langle P_2, L_2, \Sigma_5 \rangle$. Figure 4-11 shows $BIPARTITE^{-1}$ operating on a graph $G \in G_p$ and a graph $G \in G_p$. Notice that $BIPARTITE^{-1}$ will not accept just any graph when testing to see if it is bipartite. Each graph in the G_p produced by BIPARTITE is loop labelled, and the only input on which $BIPARTITE^{-1}$ will return "TRUE" is a correctly loop-labelled bipartite graph. On a non-bipartite

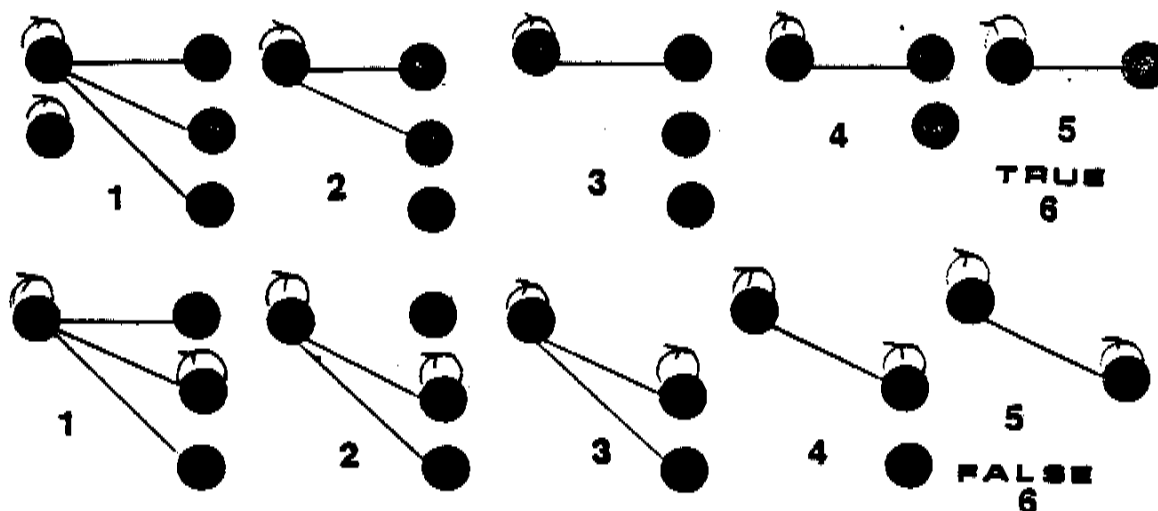


Figure 4-11: $BIPARTITE^{-1}$ in Operation

graph or an unlabelled bipartite graph or an incorrectly labelled bipartite graph, the tester will return "FALSE." If we imagine all possible non-trivial loop labellings of a graph (there are $2^{n-1} - 1$ such labellings) the tester could perform in parallel on all possible labellings of an unlabelled input graph in $O(m + n)$ time, but sequentially in $O(2^n)$ time. We will see this potential for parallelism throughout the labelling properties in this chapter. If labelling is required to construct a graph, then labelling will be required to test it. We suspect that properties of unlabelled graphs which can only be implemented by labelling are intrinsically different from those which do not require labelling. Each segment in the remainder of this section describes a specific graph property which appears to require loop labelling.

4.3.2. Complete Bipartite Graphs

A *complete bipartite graph* K_{n_1, n_2} is a bipartite graph $G = \langle V, E \rangle$ where V is partitioned into V_1 and V_2 , $|V_1| = n_1$, $|V_2| = n_2$ and all possible edges are present, i.e.,

$$E = \{ xy \mid x \in V_1, y \in V_2 \}$$

Several examples of complete bipartite graphs appear in Figure 4-12. K_{n_1, n_2} is not "complete" in the full sense of COMPLETE because of the bipartite restriction. The R-property COMPLETE-BIPARTITE is:

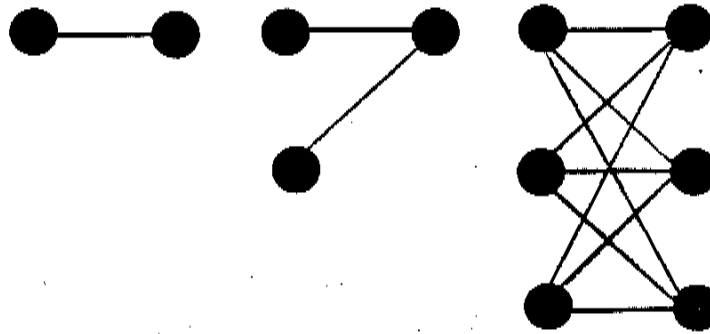


Figure 4-12: Some Complete Bipartite Graphs

$(A_{vx_1} \dots A_{vx_p} A_v + A_{wy_1} \dots A_{wy_q} A_w A_w)^* (\{1,2\}, \{11,12\})$ where
 distinct $x_i \in V, v \in V, x_i x_i \in E, i=1,2,\dots,p; |\{z \mid x \in V, xx \in E\}| = p$
 distinct $y_i \in V, w \in V, y_i y_i \in E, i=1,2,\dots,q; |\{z \mid y \in V, yy \in E\}| = q$

Figure 4-13 shows the iterative steps in a sample run of COMPLETE-BIPARTITE.

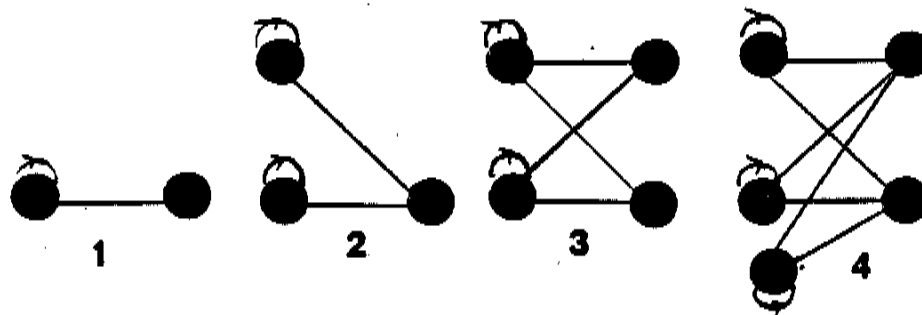


Figure 4-13: COMPLETE-BIPARTITE in Operation

The seed is described in L_1 as $\underline{E} \cap \underline{1} = 0$ and $E \cap 1 \neq 0$, as are all other graphs which contain some loops and all their non-loop edges. In L_{1n} the seed is described uniquely as $\underline{E} \cap \underline{1} = 0$, $E \cap 1 \neq 0$ and $n = 1$, and in L_2 the seed is described uniquely as

$$\underline{E} \cap \underline{1} = 0$$

$$E \cap 1 \sim E \cap \underline{1} \sim \underline{E} \cap 1$$

Thus the floors for complete bipartite graphs are $\langle P_1, L_{1n}, \Sigma_5 \rangle$ and $\langle P_1, L_2, \Sigma_5 \rangle$.

COMPLETE-BIPARTITE uses the same loop labelling as BIPARTITE to denote the

partition on V . It adds all the appropriate edges to G when it adds a vertex. Thus COMPLETE-BIPARTITE is correct. The inverse COMPLETE BIPARTITE⁻¹ is computed by:

$$\begin{aligned}
 f^{-1} &= (A_{vx_1} \dots A_{vx_p} A_v + A_{wy_1} \dots A_{wy_q} A_w A_w)^{-1} \\
 &= (A_{vx_1} \dots A_{vx_p} A_v)^{-1} + (A_{wy_1} \dots A_{wy_q} A_w A_w)^{-1} \\
 &= D_v D_{vx_p} \dots D_{vx_1} + D_w D_{wy_q} \dots D_{wy_1} \\
 \sigma_{pre} = \sigma &= \text{distinct } x_i \in V, v \in V, x_i x_i \in E, \\
 &\quad |\{z \mid x \in V, xx \in E\}| = p \\
 &\quad \text{distinct } y_i \in V, w \in V, y_i y_i \in E, \\
 &\quad |\{z \mid y \in V, yy \in E\}| = q \\
 \sigma^{-1} &= \text{distinct } v, x_i \in V, x_i x_i \in E, \\
 &\quad i=1,2,\dots,p; \\
 &\quad |\{z \mid z \in V, zz \in E\}| = p, \\
 &\quad |\{z \mid z \in V, zz \in E\}| > 1, d(v) = p \\
 &\quad \text{distinct } w, y_i \in V, y_i y_i \in E, \\
 &\quad i=1,2,\dots,q, \\
 &\quad |\{z \mid z \in V, zz \in E\}| = q, \\
 &\quad |\{z \mid z \in V, zz \in E\}| > 1, d(w) = q
 \end{aligned}$$

Again the floors shift to $\langle P_{2,L_1}, \Sigma_3 \rangle$ and $\langle P_{2,L_2}, \Sigma_3 \rangle$. Figure 4-14 shows COMPLETE-BIPARTITE⁻¹ operating on a graph $G \in G_p$ and a graph $G \in G_p$. On a complete, loop-labelled bipartite graph, COMPLETE-BIPARTITE⁻¹ will delete one vertex at a time (preserving one looped and one unlooped vertex) until G is the seed. On a graph which is incorrectly loop-labelled, some edge will be unremovable and the graph will eventually fail. On a graph which is correctly labelled as bipartite but is not a complete bipartite graph, the absence of some edge necessary for "completeness" will prevent the deletion of both vertices associated with it, and the graph will ultimately fail. Thus COMPLETE-BIPARTITE⁻¹ is correct and COMPLETE-BIPARTITE is complete.

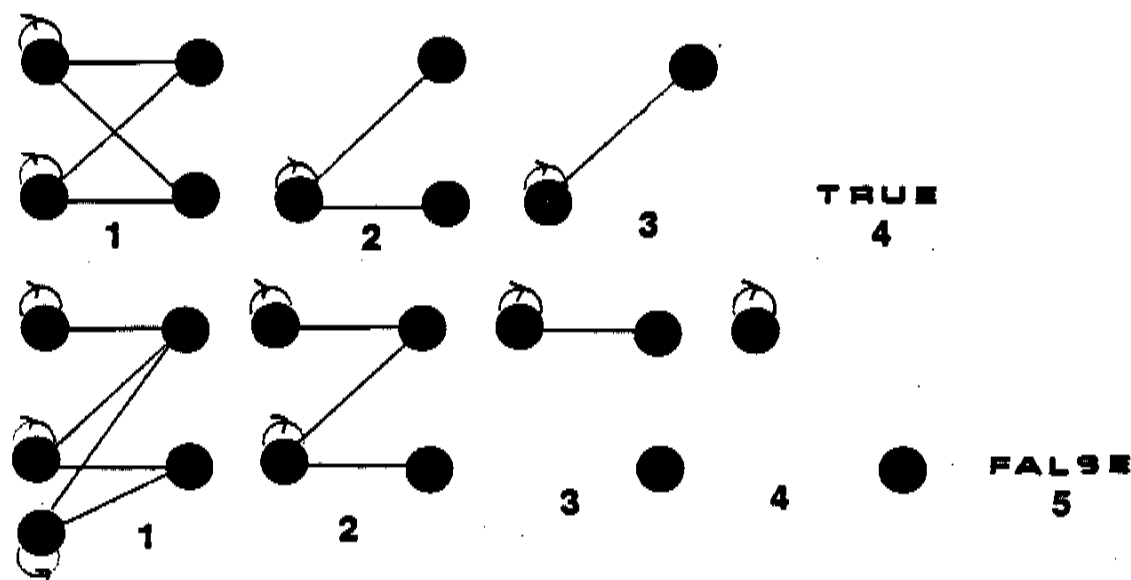


Figure 4-14: COMPLETE-BIPARTITE⁻¹ in Operation

4.3.3. K-Vertex-Covered Graphs

A vertex x covers an edge yz if x is y or z . Given a graph $G = \langle V, E \rangle$, a set of vertices $A \subseteq V$ is a *vertex cover* if for every edge yz in E either y is in A or z is in A or both. If A is a vertex cover for G and $|A| = k$, G is said to be *k-vertex-coverable*. If a graph G is *k-vertex-coverable*, A is its vertex cover and A is labelled, G is said to be *k-vertex-covered*. Several examples of 5-vertex-covered graphs appear in Figure 4-15.

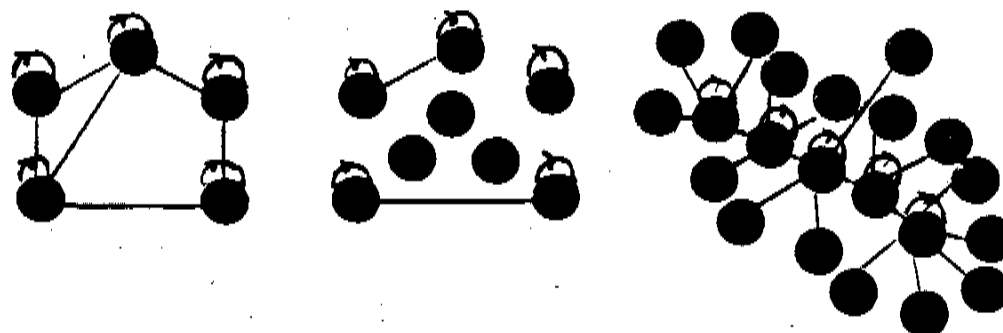


Figure 4-15: Some 5-Vertex-Covered Graphs

The R-property K-VERTEX-COVERED is

$$(A_x + A_{yz})^*(LE_k) \text{ where } y, z \in V, y \neq z, |\{yy, zz\} \cap E| \neq 0$$

Figure 4-16 shows the iterative steps in a sample run of K-VERTEX-COVERED for $k = 4$.

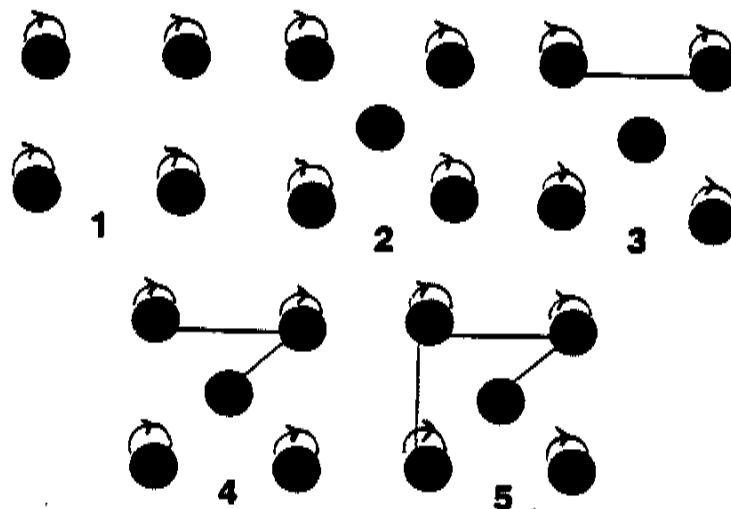


Figure 4-16: 4-VERTEX-COVERED in Operation

The seed is a graph on k vertices with no edges and all possible loops, which we have abbreviated as LE_k :

$$LE_k = \langle \{1, 2, \dots, k\}, \{11, 22, \dots, kk\} \rangle$$

The set of all LE_k 's is described in L_1 as

$$E \cap 1 = 0$$

and

$$\bar{E} \cap 1 = 0$$

but in order to distinguish a particular LE_k we require L_{1n} . Thus the floor for K-VERTEX-COVERED is $\langle P_1, L_{1n}, \Sigma_5 \rangle$. The loops label the vertex cover throughout the execution of the algorithm. No loops may be added and every edge is covered by at least one looped vertex. Thus K-VERTEX-COVERED is correct. There is no guarantee that the looped vertices form a minimal cover, merely a cover.

The inverse K-VERTEX-COVERED⁻¹ is computed from:

$$f^{-1} = (A_x + A_{yz})^{-1}$$

$$\begin{aligned}
 &= A_x^{-1} + A_{yz}^{-1} \\
 &= D_x + D_{yz} \\
 \sigma_{pre} &= x \in V \\
 & \quad y, z \in V, y \neq z, yz \in E, |\{yy, zz\} \cap E| \neq 0 \\
 \sigma^{-1} &= x \in V, xx \in E, d(x) = 0 \\
 & \quad y, z \in V, y \neq z, yz \in E, |\{yy, zz\} \cap E| \neq 0
 \end{aligned}$$

The floor shifts to $\langle P_2, L_{in}, \Sigma_S \rangle$. Figure 4-17 shows $K\text{-VERTEX-COVERED}^{-1}$ operating on a graph $G \in G_p$ and a graph $G \in G_p$ for $k = 3$.

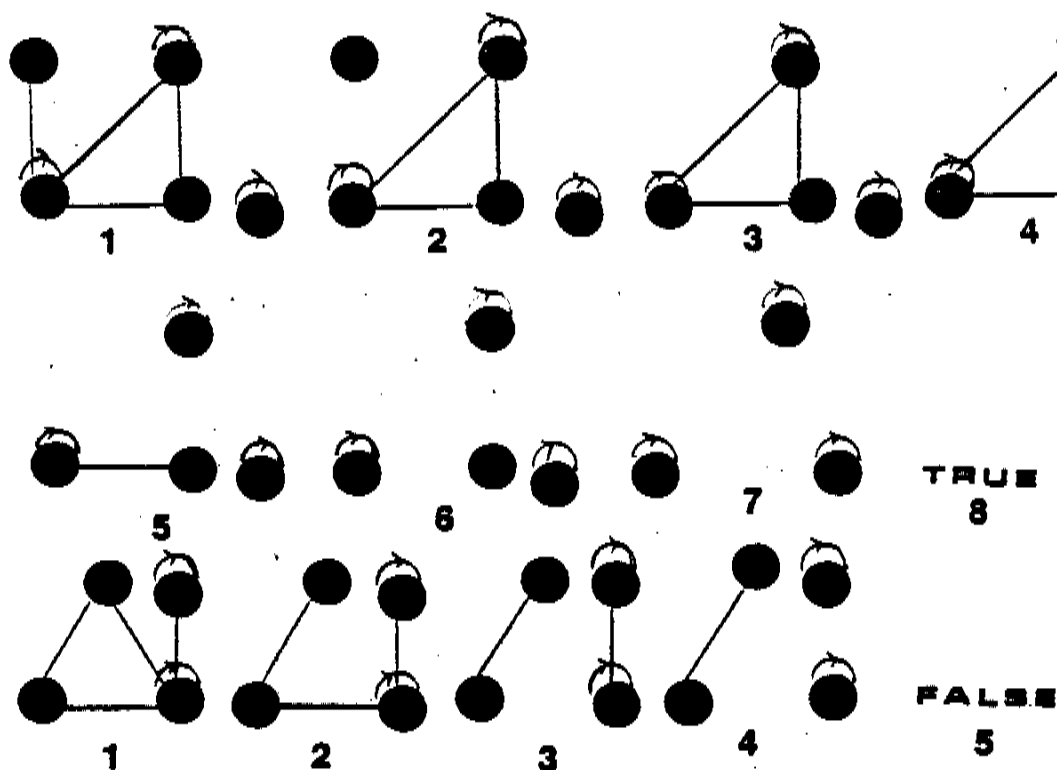


Figure 4-17: $3\text{-VERTEX-COVERED}^{-1}$ in Operation

Just as for BIPARTITE^{-1} , $K\text{-VERTEX-COVERED}^{-1}$ checks to see if a particular vertex labelling is in fact a vertex cover for the graph. On a correctly indicated cover, all edges will eventually be removed, as will all unlooped vertices, returning the graph to LE_k . On an incorrectly indicated cover some edges will remain or the final edgeless graph will contain the wrong number of looped vertices. Thus $K\text{-VERTEX-COVERED}^{-1}$ is correct and $K\text{-VERTEX-COVERED}$ is complete.

4.3.4. Graphs with K Independent Vertices

Given a graph $G = \langle V, E \rangle$, a set of vertices $A \subseteq V$ is *independent* if for any $x, y \in A$, $xy \notin E$, i.e., no two are adjacent. Several examples of graphs with 3 independent vertices appear in Figure 4-18.

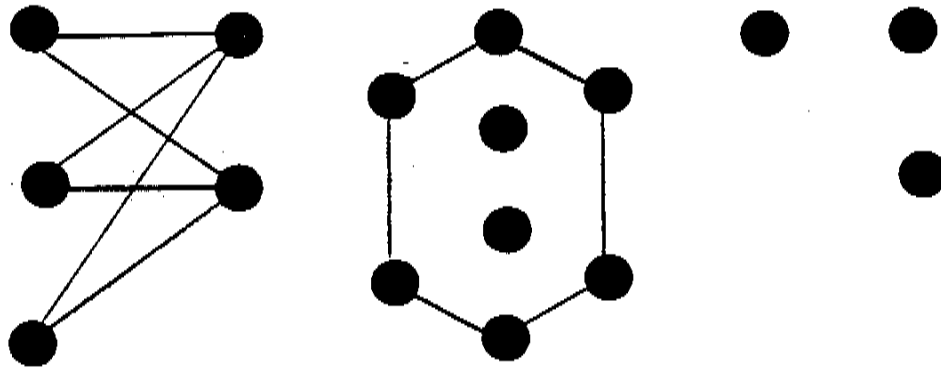


Figure 4-18: Some Graphs with 3 Independent Vertices

The R-property K-INDEPENDENT is

$$(A_x + A_{yz})^*(LE_k) \text{ where distinct } y, z \in V, |\{yy, zz\} \cap E| \leq 1$$

Figure 4-19 shows the iterative steps in a sample run of K-INDEPENDENT for $k = 3$.

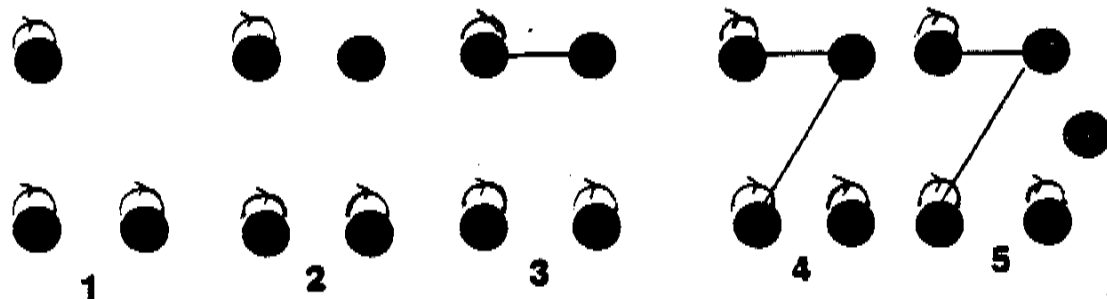


Figure 4-19: 3-INDEPENDENT in Operation

The seed is again a graph on k vertices, with all possible loops and no edges. The loops label the independent set throughout the algorithm. New, unlooped vertices may be added. Any edge may be added, as long as it is not between two looped (independent) vertices. There is no guarantee that the looped vertices form a

maximal independent set, merely an independent set.

The floor for K-INDEPENDENT is $\langle P_1, L_{1n}, \Sigma_5 \rangle$. Because no edges are ever added between the labelled vertices, K-INDEPENDENT is correct.

The inverse K-INDEPENDENT⁻¹ is computed from:

$$\begin{aligned}
 f^{-1} &= (A_x + A_{yz})^{-1} \\
 &= A_x^{-1} + A_{yz}^{-1} \\
 &= D_x + D_{yz} \\
 \sigma_{pre} &= x \in V \\
 &\quad \text{distinct } y, z \in V, yz \in E, |\{yy, zz\} \cap E| \leq 1 \\
 \sigma^{-1} &= x \in V, d(x) = 0 \\
 &\quad \text{distinct } y, z \in V, yz \in E, |\{yy, zz\} \cap E| \leq 1
 \end{aligned}$$

The floor shifts to $\langle P_2, L_{1n}, \Sigma_5 \rangle$. Figure 4-20 shows 2-INDEPENDENT⁻¹ for $k = 2$ operating on a graph $G \in G_p$ and a graph $G \in G_p$. On a graph from G_p all edges and unlooped vertices will be deleted. On a graph $G \in G_p$, either there are the wrong number of loops or loops on the wrong vertices. If there are incorrectly-placed loops, some edge will have two looped endpoints and will never be removed. If there are too few or too many loops, since loops are unremovable, the graph will never be isomorphic to LE_k . Thus K-INDEPENDENT⁻¹ is correct and K-INDEPENDENT is complete.

4.4. Labelling/Coloring Graphs

Abandoning loops for now, this section describes a substantial extension to our recursive formulation, labelling graphs. Properties which require labels by definition (such as coloring properties) and properties which are achievable via labels are considered.

Let $G = \langle V, E \rangle$ be a graph and let c be a function defined on V , i.e., $c(v)$ is defined and unique for each $v \in V$. Then we say that c is a *labelling* of G , that

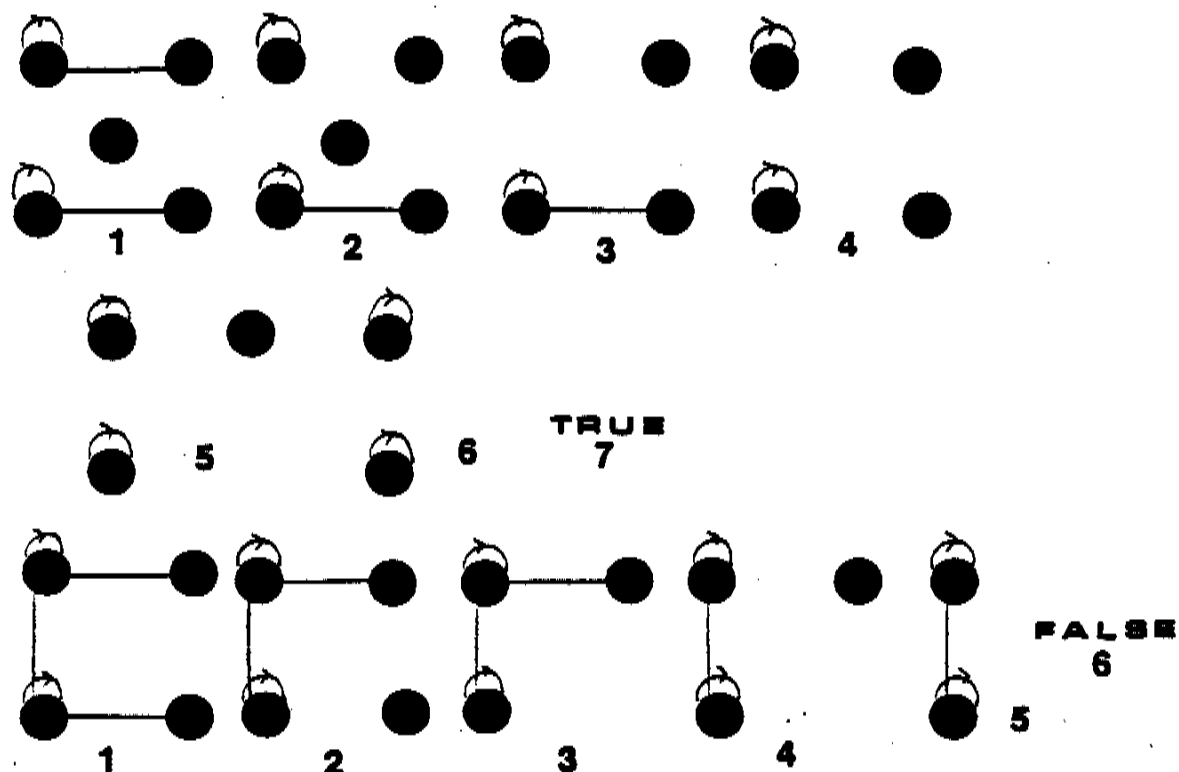


Figure 4-20: 2-INDEPENDENT⁻¹ in Operation

the range of c , $\Lambda = \{c(v) \mid v \in V\}$, is the set of *labels* for G , and that $G = \langle V, E, c, \Lambda \rangle$ is a *labelled graph*. It is important to distinguish the name of the vertex (v) from its label ($c(v)$). We use lower case Greek letters for labels. Vertex names are distinct; vertex labels need not be. As a matter of fact,

$$1 \leq |\Lambda| \leq |V|$$

A primitive form of labelling is the *loop*, where $|\Lambda| = 2$, i.e., the labels are "has a loop" or "has no loop." One helpful way to think about labels is to imagine them as colors in which the vertices may be painted, one color to a vertex. There are many graph properties which are described in terms of colors.

A labelling of G such that no two adjacent vertices have the same label is called a *coloring* of G . If $c: V \rightarrow \Lambda$ is a coloring and $|\Lambda| = k$, c is a *k-coloring*, and partitions V in to k classes. A graph G is *k-colorable* if there exist a k -coloring for G . A graph G is *k-colored* if it is k -colorable and c is a k -coloring defined on

it, i.e., it is appropriately labelled.

In order to represent coloring or labelling, we must extend the definition of an R-property. Recall that an R-property $p = \langle f, S, o \rangle$ had its origin in the ordered triple $\langle P, L, \Sigma \rangle$. We must provide first an operator to assign a label or color to a vertex. This coloring operator Z will take two arguments, a vertex v and a color α . $Z_{x\alpha}(G)$ will set $c(x)$ to α , leaving the remainder of the graph unchanged. More formally, we define the primitive operator sets:

$$P_{ic} = P_i \cup \{Z_{x\alpha}\} \text{ for } i = 1, 2, \dots, 5$$

and call any P_{ic} a P_c -language. When a vertex is added to a graph, it must always be labelled separately.

We must also provide L_c -languages in which labelled graphs may be specified. These languages have a "most-powerful" equivalent to L_0 which we call L_Z . L_Z is the language which precisely lists the vertices, edges and labels of a labelled graph. We offer the following possible amendments to the L-grammars of 3.3:

- $l \rightarrow$ labels are unique | labels are not unique
- $l \rightarrow$ labels range from 1 to k

The first, appended to the languages L_i and L_{in} will yield the L_c -languages L_{iuc} and L_{inuc} for $i = 1, 2, \dots, 6$. The second, appended to the languages L_i and L_{in} will yield the L_c -languages L_{ic} and L_{inc} for $i = 1, 2, \dots, 6$.

Finally, Σ must be augmented to test colors, as well as vertices and edges. We augment the original Σ grammars with the following:

- $l \rightarrow c(\text{vertex}) = c(\text{vertex}) \mid c(\text{vertex}) \neq c(\text{vertex}) \mid$
- $\rightarrow 1 \leq \text{color} \leq k \mid \text{color even} \mid \text{color odd}$
- $\text{color} \rightarrow \alpha \mid \beta \mid \gamma \mid \dots$

The expression $\neq$ will be interpreted semantically as "is different from" and the expression $=$ as "is identical to." By appending these forms to the languages Σ_1 through Σ_6 we produce the Σ_c -languages Σ_{1c} through Σ_{6c} , respectively. We now can formally define an R^c -property as the semantic interpretation of the triple

$\langle f, S, \sigma \rangle$, where f is a terminal P_c -expression, S is a terminal L_c -expression and σ is a terminal Σ_c -expression.

Each of the segments in the remainder of this section deals with a specific R^c -property. Either the property is for a labelled graph or its recursive formulation appears to require a labelled graph to be correct and complete.

4.4.1. K-Colored Graphs

Our first example of an R^c -property is k -colored. Several examples of 3-colored graphs appear in Figure 4-21.

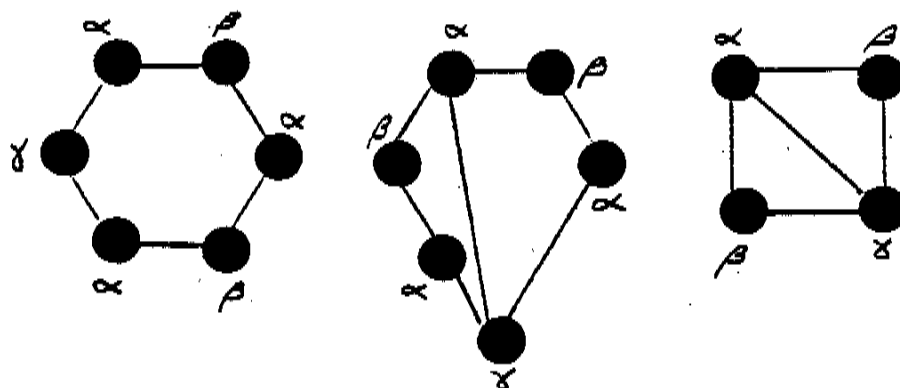


Figure 4-21: Some 3-Colored Graphs

The R -property K -COLORED is

$$(A_{xy} + \sum_{z \in V} A_z)(U_k) \text{ where } x, y \in V, c(x) \neq c(y) \\ z \in V, 1 \leq \alpha \leq k$$

The seed is U_k , the uniquely colored edgeless graph on k different-colored vertices:

$$U_k = Z_{v_1 1} Z_{v_2 2} \dots Z_{v_k k} E_k$$

Note that our "colors" are really integers between 1 and k , inclusive. U_k is in L_{1nuc} .

Figure 4-22 shows the iterative steps in a sample run of K -COLORED for $k = 4$.

The floor for K -COLORED is $\langle P_{1c} L_{1nuc} \Sigma_{1c} \rangle$.

Clearly K -COLORED is correct the only edges it adds are between

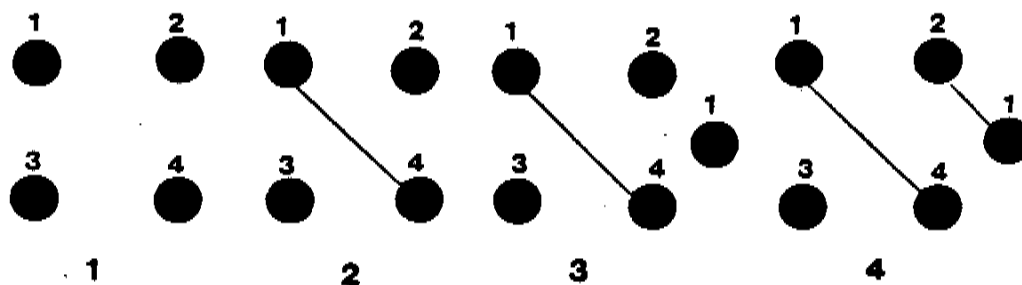


Figure 4-22: 4-COLORED in Operation

different-colored vertices, and each vertex is assigned a color when it is added to the graph.

The automatic inversion of an R^F -property raises an interesting question with respect to the operator $Z_{x\alpha}$. Other than keeping a list of all previous values for $c(x)$ (a computationally appalling prospect), we have no way of knowing what x 's label was prior to $Z_{x\alpha}$. Thus inversion will be severely limited unless we can assume that the label was λ , denoting irrelevant and/or unknown. We will therefore utilize

$$Z_{x\alpha}^{-1} = Z_{x\lambda}$$

with the understanding that some properties may not be automatically invertible.

The inverse $K\text{-COLORED}^{-1}$ is computed from:

$$\begin{aligned} f^{-1} &= (A_{xy} + Z_{z\alpha} A_z)^{-1} \\ &= A_{xy}^{-1} + (Z_{z\alpha} A_z)^{-1} \\ &= A_{xy}^{-1} + A_z^{-1} Z_{z\alpha}^{-1} \\ &= D_{xy} + D_z Z_{z\lambda} \\ \sigma_{pre} &= x, y \in V, xy \in E, c(x) \neq c(y) \\ &\quad z \in V, 1 \leq \alpha \leq k \\ \sigma^{-1} &= x, y \in V, xy \in E, c(x) \neq c(y) \\ &\quad \text{distinct } v, z \in V, d(z) = 0, c(v) = c(z) \end{aligned}$$

Note the post-profile statement that z 's color is not unique. The floor shifts to $\langle P_{2c}, L_{1nuc}, \Sigma_{1c} \rangle$.

Figure 4-23 shows $K\text{-COLORED}^{-1}$ operating on a graph $G \in G_p$ and a graph G

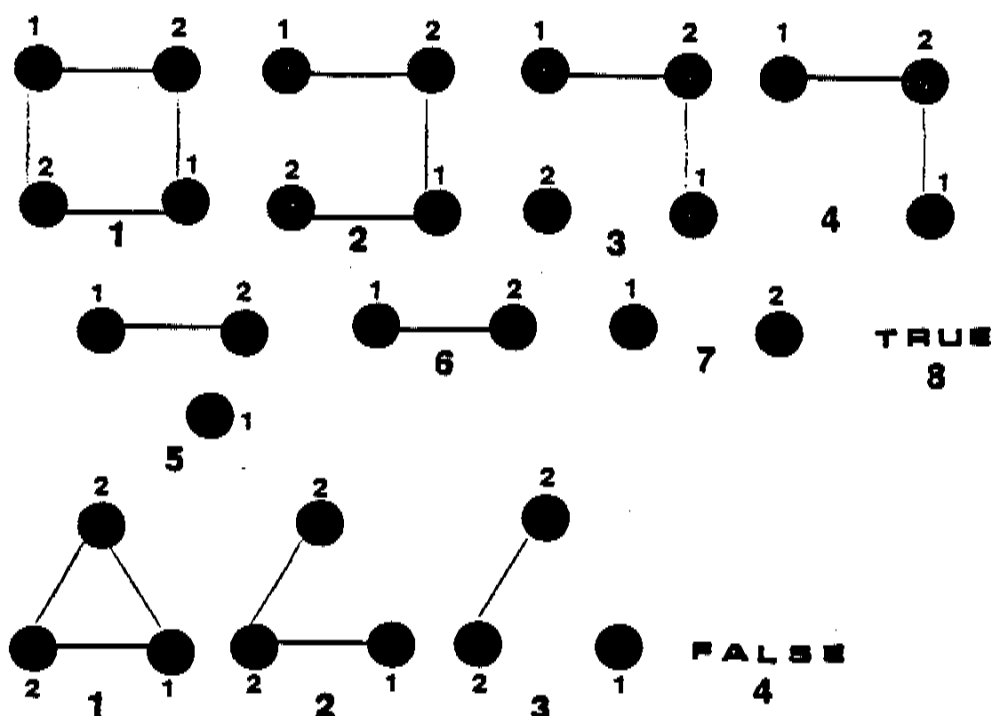


Figure 4-23: 2-COLORED⁻¹ in Operation

$\equiv G_p$ for $k = 2$. On a correctly-labelled k -colored graph, the edges will be removed one at a time and any degree zero vertex of non-unique color deleted until U_k is reached. If any vertex in G is improperly colored, some edge will not be removable. If $G \equiv G_p$ is colored with the wrong number of colors, there will be no isomorphism with U_k . Note that, as for loop labelling, a correct graph incorrectly labelled will fail. For example, a six-colored graph is also seven-colorable if $n \geq 7$, but if it is submitted to 7-COLORED⁻¹ in six colors it will fail. K -COLORED⁻¹ is correct and K -COLORED is complete.

4.4.2. K -Chromatic Graphs

A graph is said to be k -chromatic if it is k -colorable but not colorable in fewer than k colors. (This is equivalent to saying that it is k -colorable but not $k-1$ colorable.) If a graph is k -chromatic, k is the smallest number of colors with which it can be colored. Several examples of labelled k -chromatic graphs appear in Figure 4-24 for $k = 3$. For clarity of presentation we define two new composite

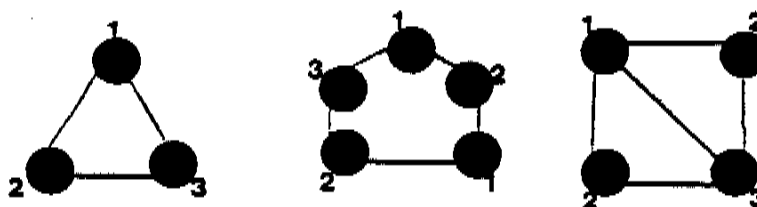


Figure 4-24: Some 3-Chromatic Graphs

operators, S_{xvwy} and $X_{v_1s_1 \dots v_rs_r}$. S_{xvwy} is a double-subdivide operator; it replaces the edge between x and y with a chain of length three, i.e.,

$$S_{xvwy} = D_{xy} A_{xv} A_{vw} A_{wy}$$

S_{xvwy} is distinguishable from the regular subdivide operator S_{xvy} by the arity of its subscript. $X_{v_1s_1 \dots v_rs_r}$ is an exchange operator which introduces similarly-labelled surrogate vertices $s_1, s_2, \dots, s_r$ for the vertices $v_1, v_2, \dots, v_r$, respectively. $X_{v_1s_1 \dots v_rs_r}$ replaces each edge between a v_i and a v_j with three edges, one between s_i and s_j , another between s_i and v_j , and a third between v_i and s_j . $X_{v_1s_1 \dots v_rs_r}$ also appends correctly-labelled vertices $s_1, s_2, \dots, s_r$ to G :

$$X_{v_1s_1 \dots v_rs_r} = D_{v_1v_2} A_{s_1v_2} A_{v_1s_2} A_{s_2v_3} \dots Z_{s_1c(v_1)} Z_{s_2c(v_2)} \dots Z_{s_rc(v_r)} A_{s_1} A_{s_2} \dots A_{s_r}$$

where $D_{v_1v_2} A_{s_1v_2} A_{v_1s_2} A_{s_2v_3}$ occurs for each $v_i, v_j \in E$.

The R^c -property K -CHROMATIC] is

$$(Z_{x\alpha} A_x + A_{yz} + A_{tw_1} A_{tw_2} \dots A_{tw_{k-3}} A_{uw_1} A_{uw_2} \dots A_{uw_{k-3}} Z_{pc(w)} Z_{qc(t)} S_{tpqu} \\ + X_{v_1s_1 \dots v_rs_r})^*(T_k)$$

where $x \in V$, $1 \leq \alpha \leq k$,

$$y, z \in V, c(y) \neq c(z)$$

$$\text{distinct } t, u, w_i \in V, p, q \in V, tu \in E, \text{ distinct } c(t), c(u), c(w_i)$$

$$\text{distinct } v_i \in V, \text{ distinct } s_i \in V, r \geq k-1$$

Figure 4-25 shows the iterative steps in a sample run of K -CHROMATIC for $k = 4$. The seed is T_k , the complete graph on k vertices with each vertex a different color. Clearly T_k is k -chromatic, for each pair of vertices is adjacent and must be a different color. The floor for K -CHROMATIC is $\langle P_{2c}, L_{1nuc}, \Sigma_{2c} \rangle$. There are four property-preserving choices for an iteration of K -CHROMATIC. The first two,

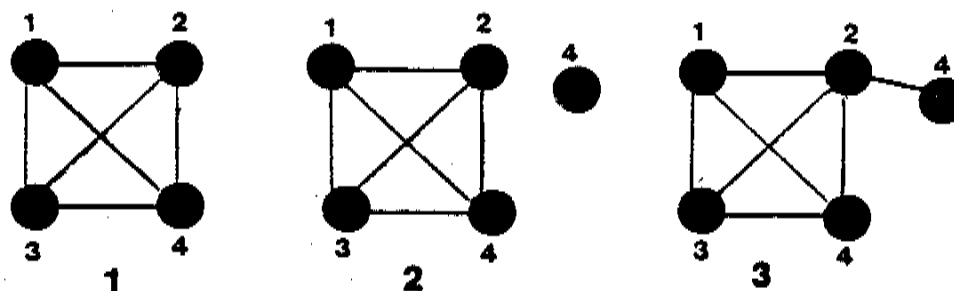
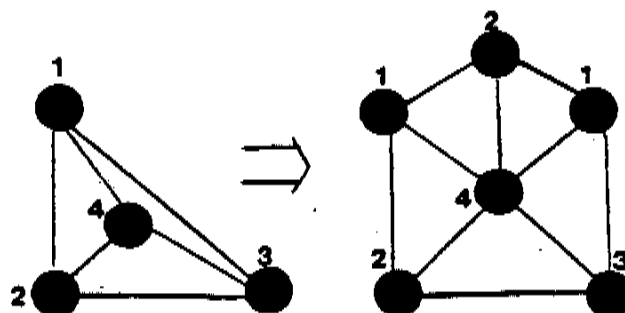


Figure 4-25: 4-CHROMATIC in Operation

$Z_{x\alpha}A_x$ and A_{yz} add a properly-labelled vertex and a legal edge, respectively. If we had stopped here, with seed T_k we would know that we had forced k colors. Consider, however, the wheel $W_{1,5}$. To color C_5 alone requires three colors. Since the hub is adjacent to every vertex on the rim, a fourth color is required, and thus $W_{1,5}$ has chromatic number four. How would we reach $W_{1,5}$ from T_4 ? The third choice, a double subdivision of any edge with appropriate linkage is the answer. This not-so-obvious construction is displayed in Figure 4-26.

Figure 4-26: The Generation of $W_{1,5}$

Unfortunately, this does not solve all our problems. Graph theorists have shown that for any $k > 0$, there exist k -chromatic graphs containing no triangle (cycle of length three). A famous example of such a graph for $k = 4$ is the Grotzsch graph, shown in Figure 4-27. Certainly T_k is filled with triangles and we must provide ways to obliterate them. The fourth choice is a surrogate procedure which enables us to construct triangle-free graphs. The exchange selects a set of vertices $\{v_i\}$ already in the graph and appends a set of similarly-labelled vertices $\{s_i\}$. Since $\{v_i\}$

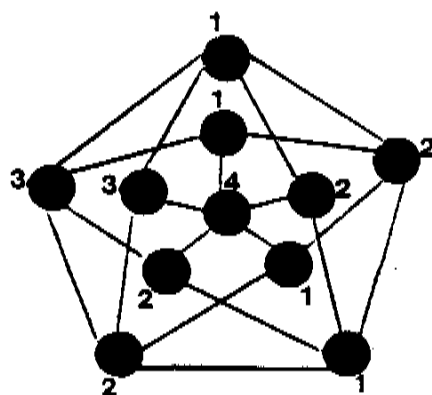


Figure 4-27: The Grotzsch Graph

is not necessarily all of V , it is possible to obliterate many, or even all, triangles. In particular, the Grotzsch graph is constructable from $W_{1,5}$ by this technique, taking the $\{v_i\}$ to be the rim vertices. (See Figure 4-28.)

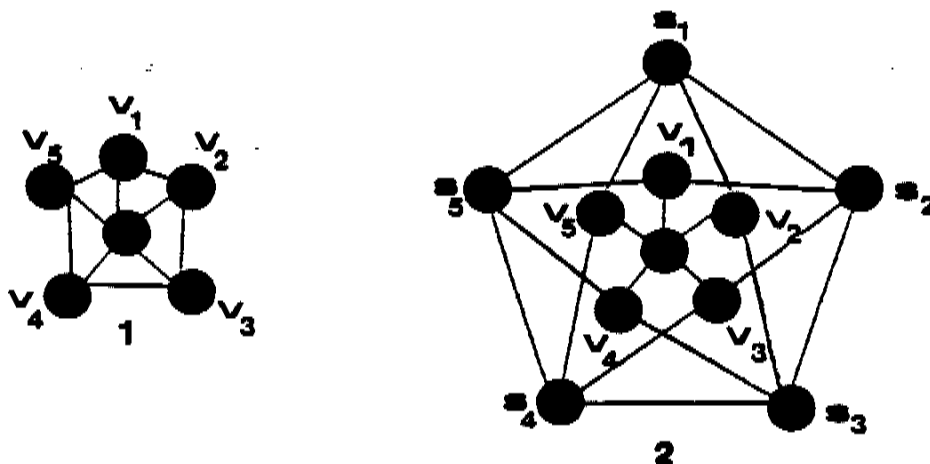


Figure 4-28: Generating the Grotzsch Graph

Having explained the motivation for K -CHROMATIC, we now demonstrate that it is correct. If any $k-1$ coloring were possible for $Z_{x\alpha}A_xG$ or $A_{yz}G$, it would have been possible for G and thus the first two choices are correct. The third choice creates a chain of length three which alternates colors $c(t)$ and $c(u)$ on its four vertices. Since $c(t)$ and $c(u)$ are distinct from $c(w_i)$, the new graph will still be k -colorable. If a $k-1$ coloring is possible after the third choice, it would have to color t and u the same since tu is the only edge removed from G by this option,

but then we would need two additional colors for p and q , and $k - 3$ more colors for the w 's, for a total of k colors. Therefore G will be k -chromatic after the third choice. Finally we examine the fourth choice. Since $c(s_1) \neq c(v_1)$ all the added edges are legal. If there were a $k-1$ coloring after the fourth choice, it would have to color some v_1 and v_2 the same, for a previously existing $v_1 v_2 \in E$. Note, however, that $c(s_1)$ and $c(s_2)$ must still be distinct because $s_1 s_2$ is now in E and no reduction in the number of colors needed is possible. Since there are $r \geq k-1$ distinct surrogate colors, this will not be possible. We have, at length, shown K -CHROMATIC to be correct.

The inverse K -CHROMATIC $^{-1}$ is computed from:

$$\begin{aligned}
 f^{-1} &= (Z_{x\alpha} A_x + A_{yz} + \\
 &\quad A_{tw_1} A_{tw_2} \dots A_{tw_{k-3}} A_{uw_1} A_{uw_2} \dots A_{uw_{k-3}} Z_{pc(w)} Z_{qc(u)} S_{tpqu} + \\
 &\quad X_{v_1 s_1 \dots v_r s_r})^{-1} \\
 &= (Z_{x\alpha} A_x)^{-1} + A_{yz}^{-1} + (A_{tw_1} A_{tw_2} \dots A_{tw_{k-3}} A_{uw_1} A_{uw_2} \dots A_{uw_{k-3}} \\
 &\quad Z_{pc(w)} Z_{qc(u)} S_{tpqu})^{-1} + X_{v_1 s_1 \dots v_r s_r}^{-1} \\
 &= D_x Z_{x\lambda} + D_{yz} + D_{qu} D_{pq} D_{tp} A_{tw} D_{uq} D_{p\lambda} Z_{q\lambda} Z_{p\lambda} \\
 &\quad D_{uw_{k-3}} D_{uw_{k-4}} \dots D_{uw_1} D_{tw_{k-3}} D_{tw_{k-4}} \dots D_{tw_1} + \\
 &\quad D_{s_r} D_{s_{r-1}} \dots D_{s_1} Z_{s_r \lambda} Z_{s_{r-1} \lambda} \dots Z_{s_1 \lambda} D_{s_1 s_2} D_{v_1 s_2} \dots D_{v_r s_r} A_{v_r v_1} \\
 &= x \in V, 1 \leq \alpha \leq k
 \end{aligned}$$

σ_{pre}

$$y, z \in V, yz \in E, c(y) \neq c(z)$$

$$\text{distinct } t, u, w_1 \in V, p, q \in V, tu \in E, \text{ distinct } c(t), c(u), c(w_1)$$

$$\text{distinct } v_1 \in V, \text{ distinct } s_1 \in V, r \geq k-1$$

σ^{-1}

$$= \text{distinct } x, x' \in V, d(x) = 0, 1 \leq \alpha \leq k, c(x') = c(x)$$

$$y, z, y', z' \in V, yz, y'z' \in E, c(y) \neq c(z), c(y') = c(y),$$

$$c(z') = c(z)$$

$$\text{distinct } p, q, t, u, w_1 \in V, \text{ distinct } c(t), c(u), c(w_1);$$

$$c(w) = c(p), c(q) = c(t), d(p) = k-1, d(q) = k-1;$$

$$tp, pq, qu \in E, tu \in E$$

$$\text{distinct } v_1, s_1 \in V, c(v_1) = c(s_1), s_1 s_2, s_1 v_1, v_1 s_2 \in E, v_1 v_2 \in E,$$

$$c(v_i) \neq c(v_j)$$

The floor shifts to $\langle P_{2c}, L_{1nuc}, \Sigma_{3c} \rangle$. Figure 4-29 shows $K\text{-CHROMATIC}^{-1}$ operating on a graph $G \in G_p$ and a graph $G \notin G_p$ for $k = 2$.

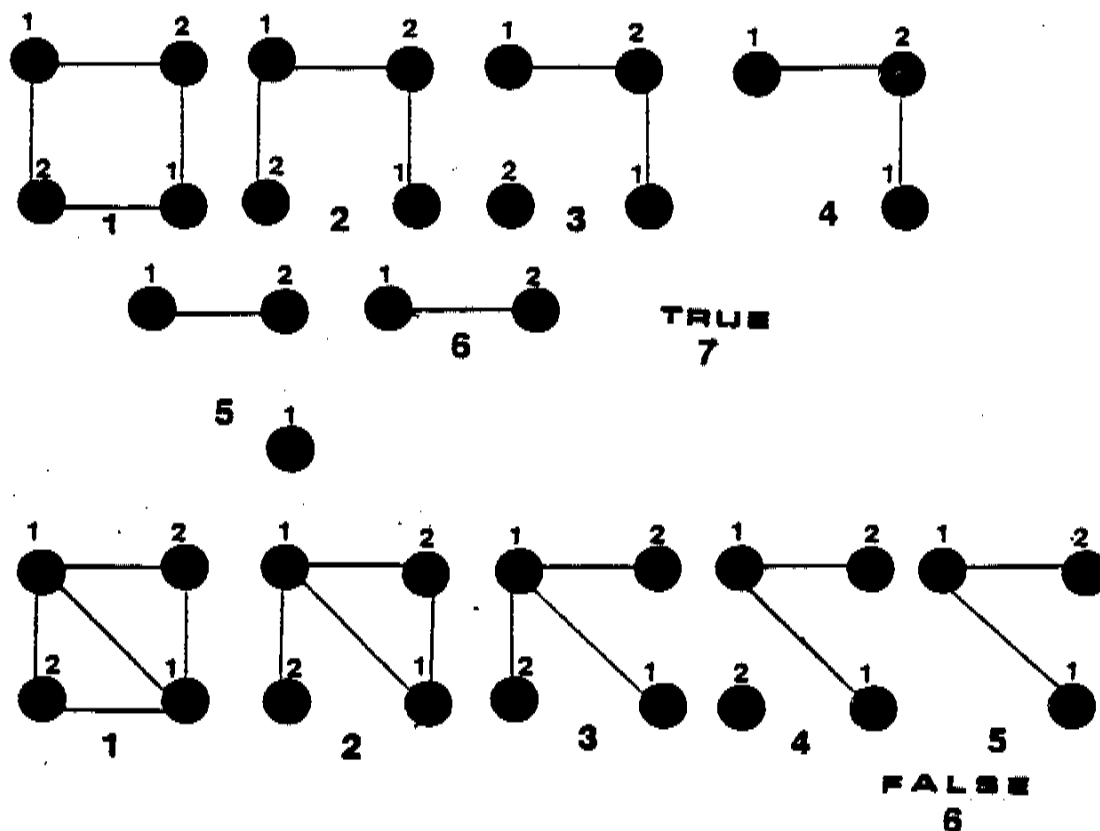


Figure 4-29: 2-CHROMATIC^{-1} in Operation

If $G \notin G_p$ because some edge has endpoints of the same color or because G has the wrong number of colors, $K\text{-CHROMATIC}^{-1}$ will not change those conditions and the graph will fail. If $G \notin G_p$ because a $k-1$ coloring is possible, G cannot reduce to T_k under $K\text{-CHROMATIC}^{-1}$ and G will fail. It remains only to show that $G \in G_p$ will reduce to T_k . Such a proof requires some background first.

An *elementary edge contraction* is defined to be $I_{xy}D_{xy}$ for $x, y \in V$, $xy \in E$. A graph G is *contractible* to a graph H if there exists a sequence of elementary

contractions transforming G into H . Hadwiger's Conjecture states that every connected k -chromatic graph is contractible to K_k . Hadwiger's Conjecture has been shown true for $n \leq 4$ and equivalent to the Four Color Theorem for $n = 5$, which this author accepts as proven. Since the inverse of a double subdivision may be seen as a sequence of two elementary edge contractions ($|_{tp} D_{tp}$ and $|_{uq} D_{uq}$) and since the inverse of the surrogate exchange process $X_{v_1 s_1 \dots v_r s_r}$ is a sequence of r elementary edge contractions ($|_{v_i s_i} D_{v_i s_i}$), we assert that the completeness of K -CHROMATIC is equivalent to the truth of Hadwiger's conjecture which, thus far, has held up since 1943. Thus K -CHROMATIC appears "reasonably complete," i.e., within our current knowledge of graph theory.

It is interesting to observe that an attempt to formulate k -CHROMATIC based only on Hadwiger's Conjecture is doomed to failure, i.e., not any sequence of elementary edge subdivisions (the opposite of contractions) will maintain the chromatic number. Notably, $S_{xvy} C_4 = C_5$, but C_4 has chromatic number two and C_5 has chromatic number three.

4.4.3. Graphs with Vertex Covering Number K

A graph $G = \langle V, E \rangle$ has *vertex covering number* k if there is a k -vertex cover for it and no vertex cover of smaller cardinality exists. (This is equivalent to saying that no vertex cover of cardinality $k-1$ exists for it.) A graph with vertex covering number k is k -vertex-coverable, but not necessarily vice versa. Several examples of graphs with vertex covering number five appear in Figure 4-30. For a graph $G = \langle V, E \rangle$, $G' = \langle V', E' \rangle$ is a *subgraph* of G if $V' \subseteq V$ and $E' \subseteq E$. A subgraph G' of a graph G is a *block* of G if every pair of edges in G' lies on a common cycle and there is no larger subgraph of G containing G' which is also a block. (In other words, G' is a maximal subgraph of G for which every pair of edges lie on a common cycle.) The blocks of G do not necessarily partition V ; two blocks share at most one vertex. The blocks of G do partition E , however, and the partition is finer than that imposed by connected components. Two blocks have at most one vertex in common; such a shared vertex is called a *cutpoint*. Cutpoints between

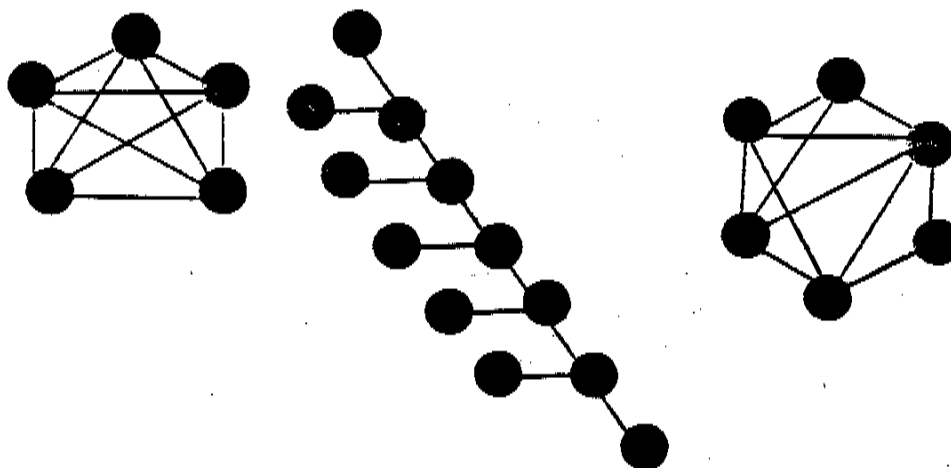


Figure 4-30: Some Graphs with Vertex Covering Number 5

blocks of more than one edge can always be covering vertices in a minimal vertex cover. Every block, except K_2 , is biconnected. The vertex covering number of a graph is the sum of the vertex covering numbers of its connected components. That is, if G has r components with vertex covering numbers $c_1, c_2, \dots, c_r$, the vertex covering number of G is $\sum_{i=1}^r c_i$.

Our approach will be to construct a graph each of whose connected components is a different color. Within a given component, every vertex will be the same color. Within each component is a skeleton subgraph, consisting of the largest cycle in each block. (If the block is a single edge, that edge lies in the skeleton.) The skeleton determines the size k of the minimal vertex cover. Thus there are two kinds of operations within the R^{C^+} -property we will describe: operations g which enlarge k by expanding the skeleton, and operations h which leave the value of k unchanged. Because each iteration of an R^+ -property is supposed to increment k by one, the R^{C^+} -property VERTEX-COVER is of the form $(h^*g)^*$:

$$\begin{aligned}
 & ([Z_{vc(x)} S_{xy} + Z_{zc(w)} B_{wz} + A_{pq}]^* [Z_{t_1 c(r_1)} Z_{s_1 c(r_1)} (A_{s_1 s_1} + A_{t_1 t_1}) B_{r_1 s_1} \\
 & + Z_{t_2 c(r_2)} Z_{s_2 c(r_2)} A_{s_2 s_2} B_{r_2 s_2} + Z_{bc(a)} Z_{dc(a)} A_{bb} S_{abde} \\
 & + Z_{gc(f)} Z_{hc(f)} A_{gg} A_{hf} B_{gh} B_{fg} + Z_{jc(i)} A_{jj} B_{ij} + A_{u_1 u_1} A_{u_1 u_3}])^* (T_1, 0)
 \end{aligned}$$

where $x, y \in V$, $v \in V$, $xy, xx, yy \in E$, $c(x) = c(y)$

$$w \in V, z \in V, ww \in E$$

$$p, q \in V, |\{pp, qq\} \cap E| \geq 1, c(p) = c(q)$$

$$r_1 \in V, \text{ distinct } s_1, t_1 \in V, r_1 r_1 \in E$$

$$r_2 \in V, \text{ distinct } s_2, t_2 \in V, r_2 r_2 \in E$$

$$\text{distinct } a, e \in V, \text{ distinct } b, d \in V, ae, ae, ee \in E$$

$$f \in V, \text{ distinct } g, h \in V, ff \in E$$

$$i \in V, j \in V, |V| = 1, |E| = 0$$

$$\text{distinct } u_1, u_2, u_3 \in V, u_1 u_2, u_2 u_2, u_2 u_3 \in E, |V| = 3, |E| = 2$$

The seed is the uniquely colored complete graph T_1 . The floor for VERTEX-COVER is $\langle P_{2c}, L_{1cu}, \Sigma_{5c} \rangle$. Figure 4-31 shows the iterative steps in a sample run of VERTEX-COVER. (All of the labels are identical and omitted.)

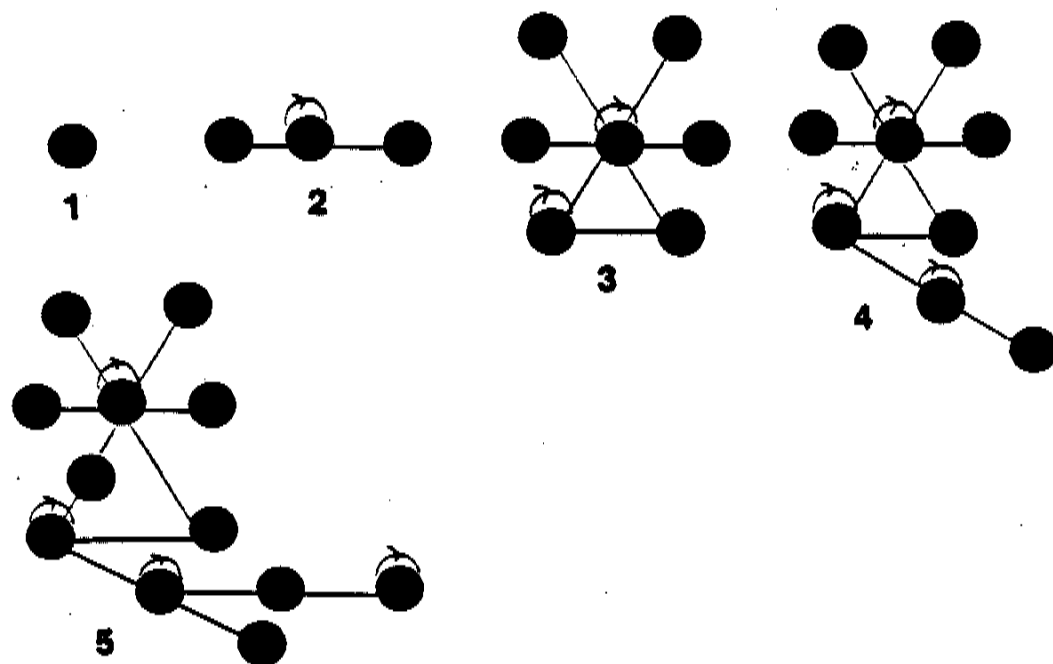


Figure 4-31: VERTEX-COVER in Operation

VERTEX-COVER is of the form $(h^*g)^*$, where h has three options and g has seven. The looped vertices are the covering vertices throughout the execution of the algorithm. The uniform coloring of each component is maintained. Each of the three h options for iteration safely expands the graph, adding a covered edge and not adding vertices which could permit a smaller cover by their participation. The

subdivision of an edge (xy) between two covering vertices will result in two new, covered edges (xv and vy), and a new vertex which, to reduce the covering number, would have to replace x and y, an impossibility. A branch from a covering vertex (w) results in a covered edge and a new vertex (z) which cannot possibly reduce the vertex covering number. An edge addition (A_{pq}) with at least one covering endpoint is covered and cannot change the vertex covering number. Those options do not increment k and any number of them may appear in a single iteration of VERTEX-COVER. These are followed by the seven options which will increment k. The first two options add chains of length two to a looped vertex (r_1). The first (s_1) or second (t_1) added vertex becomes a covering vertex. The third option adds a chain of length two to a non-looped vertex (r_2). The first added vertex (s_2) becomes a covering vertex. The fourth option double subdivides an edge between two looped vertices (a and a). The first added vertex (b) becomes a covering vertex. The next option adds a triangle with one new looped vertex (g), appending it to a looped vertex (f). The last two options are applicable only once. One of them produces an appropriately looped and labelled K_2 ; the other moves from a correctly looped and labelled chain on three vertices to a correctly looped and labelled cycle on three vertices. Every edge introduced by $(h^*g)^*$ is covered. No vertex introduced by h could make a vertex cover smaller by its inclusion, and the increment to k is carefully controlled by g. Thus VERTEX-COVER is correct.

The inverse VERTEX-COVER⁻¹ is computed by:

$$\begin{aligned}
 f^{-1} &= ([Z_{vc(x)} S_{xvy} + Z_{zc(w)} B_{wz} + A_{pq}]^* [Z_{t_1 c(r_1)} Z_{s_1 c(r_1)} (A_{s_1 s_1} + \\
 &\quad A_{t_1 t_1}) B_{s_1 t_1} B_{r_1 s_1} + Z_{t_2 c(r_2)} Z_{s_2 c(r_2)} (A_{s_2 s_2} B_{s_2 t_2} B_{r_2 s_2} + \\
 &\quad Z_{bc(a)} Z_{dc(a)} A_{bb} S_{abde} + A_{gc(f)} A_{hc(f)} A_{gg} A_{hf} B_{gh} B_{fg} + \\
 &\quad Z_{jc(i)} A_{jj} B_{ij} + A_{u_1 u_1} A_{u_1 u_3}])^{-1} \\
 &= [Z_{t_1 c(r_1)} Z_{s_1 c(r_1)} (A_{s_1 s_1} + A_{t_1 t_1}) B_{s_1 t_1} B_{r_1 s_1} + \\
 &\quad Z_{t_2 c(r_2)} Z_{s_2 c(r_2)} (A_{s_2 s_2} B_{s_2 t_2} B_{r_2 s_2} + \\
 &\quad Z_{bc(a)} Z_{dc(a)} A_{bb} S_{abde} + A_{gc(f)} A_{hc(f)} A_{gg} A_{hf} B_{gh} B_{fg} + \\
 &\quad Z_{jc(i)} A_{jj} B_{ij} + A_{u_1 u_1} A_{u_1 u_3})]^{-1} ([Z_{vc(x)} S_{xvy} + Z_{zc(w)} B_{wz} + \\
 &\quad A_{pq}]^*)^{-1}
 \end{aligned}$$

$$= [Z_{t_1 c(r_1)} Z_{s_1 c(r_1)} (A_{s_1 s_1} + A_{t_1 t_1}) B_{s_1 t_1} B_{r_1 s_1} + \\ Z_{t_2 c(r_2)} Z_{s_2 c(r_2)} A_{s_2 s_2} B_{s_2 t_2} B_{r_2 s_2} + \\ Z_{bc(a)} Z_{dc(a)} A_{bb} S_{abde} + A_{gc(f)} A_{hc(f)} A_{gg} A_{hf} B_{gh} B_{fg} + \\ Z_{jc(ii)} A_{jj} B_{ij} + A_{u_1 u_1} A_{u_1 u_3}]^{-1} ([Z_{vc(x)} S_{xvy} + Z_{zc(w)} B_{wz} + \\ A_{pq}]^{-1})^*$$

$$= [D_{s_1} D_{r_1 s_1} D_{t_1} D_{s_1 t_1} (D_{s_1 s_1} + D_{t_1 t_1}) Z_{s_1 \lambda} Z_{t_1 \lambda} + \\ D_{s_2} D_{r_2 s_2} D_{t_2} D_{s_2 t_2} D_{s_2 s_2} Z_{s_2 \lambda} Z_{t_2 \lambda} + \\ D_d D_b D_d D_{bd} D_{ab} A_{aa} D_{bb} D_{d\lambda} Z_{b\lambda} + \\ D_g D_{fg} D_h D_{gh} D_{hf} D_{gg} Z_{h\lambda} Z_{g\lambda} + D_j D_{ij} D_{jj} Z_{j\lambda} + \\ D_{u_1 u_3} D_{u_1 u_1}] [A_{xy} D_{vy} D_{xv} D_{xv\lambda} + D_z D_{wz} Z_{z\lambda} + D_{pq}]^*$$

σ_{pre}

$$= x, y \in V, v \in V, xy, xx, yy \in E, c(x) = c(y)$$

$$w \in V, z \in V, ww \in E$$

$$\text{distinct } p, q, r \in V, pq, rr \in E, |\{pp, qq\} \cap E| \geq 1,$$

$$c(p) = c(q), \text{ not } [pp, pr \in E, qq, qr \in E]$$

$$r_1 \in V, \text{ distinct } s_1, t_1 \in V, r_1 r_1 \in E$$

$$r_2 \in V, \text{ distinct } s_2, t_2 \in V, r_2 r_2 \in E$$

$$\text{distinct } a, e \in V, \text{ distinct } b, d \in V, aa, aa, ee \in E$$

$$f \in V, \text{ distinct } g, h \in V, ff \in E$$

$$i \in V, j \in V, |V| = 1, |E| = 0$$

$$\text{distinct } u_1, u_2, u_3 \in V, u_1 u_2, u_2 u_2, u_2 u_3 \in E, |V| = 3,$$

$$|E| = 2$$

σ^{-1}

$$= \text{distinct } t, u'x, v, y \in V, tx, u'y, xx, yy, xv, vy \in E,$$

$$vv, tt, u'u', xy \in E, d(v) = 2,$$

$$c(x) = c(y) = c(v) = c(t) = c(u')$$

$$\text{distinct } s', w, z \in V, s'w, ww, wz \in E, s's', zz \in E,$$

$$d(z) = 1, c(w) = c(z) = c(s')$$

$$\text{distinct } a', b', p, q \in V, a'p, b'q, pq \in E,$$

$$a'a', b'b' \in E, |\{pp, qq\} \cap E| \geq 1,$$

$$c(p) = c(q) = c(a') = c(b'),$$

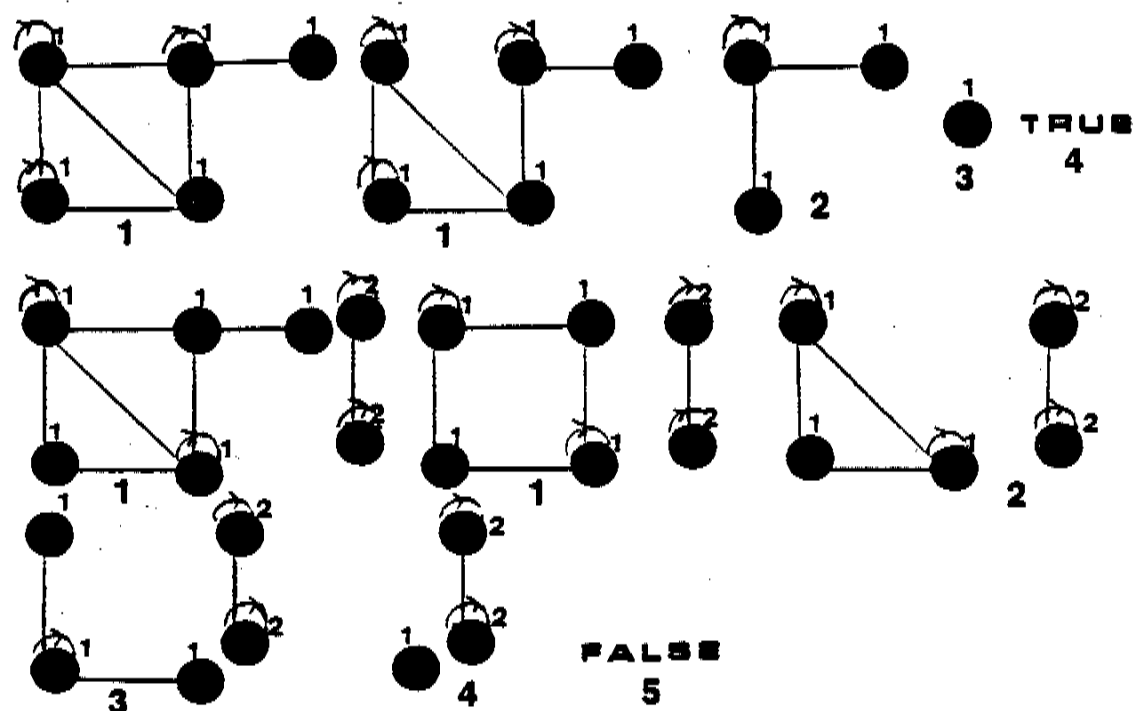
$$\text{there exists a cycle containing } pq$$

$$\text{distinct } r_1, s_1, t_1 \in V, r_1 r_1, r_1 s_1, s_1 s_1, s_1 t_1 \in E, t_1 t_1 \in E,$$

$d(s_1) = 2, d(t_1) = 1, c(r_1) = c(s_1) = c(t_1)$
 distinct $r_1, s_1, t_1 \in V, r_1 r_1, r_1 s_1, s_1 t_1, t_1 t_1 \in E, s_1 s_1 \in E$,
 $d(s_1) = 2, d(t_1) = 1, c(r_1) = c(s_1) = c(t_1)$
 distinct $r_2, s_2, t_2 \in V, t_2 t_2, r_2 r_2 \in E, s_2 s_2, r_2 s_2, s_2 t_2 \in E$,
 $d(s_2) = 2, d(t_2) = 1, c(r_2) = c(s_2) = c(t_2)$
 distinct $a, b, d, e \in V, aa, bb, ee, ab, bd, de \in E, dd, ae \in E$,
 $d(b) = 2, d(d) = 2, c(a) = c(b) = c(d) = c(e)$
 distinct $f, g, h \in V, ff, gg, fg, gh, fh \in E, hh \in E, d(g) = 2$,
 $d(h) = 2, c(f) = c(g) = c(h)$
 distinct $i, j \in V, ij, jj \in E, ii \in E, d(j) = 1, |V| = 2$,
 $|E| = 1, c(i) = c(j)$
 distinct $u_1, u_2, u_3 \in V, u_{11}, u_{22}, u_{12}, u_{13}, u_{23} \in E, u_{33} \in E$,
 $|V| = 3, |E| = 3, c(u_1) = c(u_2) = c(u_3)$

The floor shifts to $\langle P_{2c}, L_{1cu}, \Sigma_{8c} \rangle$. Figure 4-32 shows VERTEX-COVER⁻¹ operating on a graph $G \in G_p$ and a graph $G \in G_p$. VERTEX-COVER⁻¹ deletes as many edges as possible which do not destroy the underlying skeletal graph. It accomplishes this by testing for other "justifying" (primed) adjacencies which would argue for retaining the covering status of a vertex and the connectedness of a block. Then VERTEX-COVER⁻¹ will contract the underlying skeletal graph. On $G \in G_p$ VERTEX-COVER⁻¹ will return G to K_1 , decrementing k as it goes. If the cover to be tested is not of size k , k will not be zero on termination and G will fail. If the cover of the input graph is not minimal, some looped vertices will not be deleted and G will fail. If the cover of the input graph is incorrectly indicated, some edge will lie between two unlooped vertices and never be deleted, causing failure. Thus VERTEX-COVER⁻¹ is correct and VERTEX-COVER is complete.

This algorithm is an interesting construction. The skeleton could have served as a seed set for an R-property K-VERTEX-COVER instead. Such an elaborate seed set would require L_0 and has interesting connotations, to be discussed in 4.7.

Figure 4-32: VERTEX-COVER⁻¹ in Operation

4.4.4. Graphs with Independence Number K

The cardinality of the largest independent vertex set in a graph is its *independence number*. Several examples of graphs with independence number 3 appear in Figure 4-33.

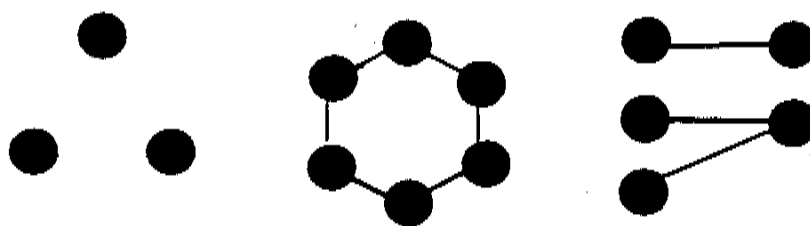


Figure 4-33: Some Graphs with Independence Number 3

The R^C-property INDEPENDENCE-K is:

$$(Z_{v_1 \alpha_1} \dots Z_{v_n \alpha_n} Z_{x \alpha} A_{xy_i}^* A_{xz_1} \dots A_{xz_p} A_x)^*(LE_k)$$

where $x \in V$, $y_i, z_i \in V$, z_i is in the i th independent set.

p = number of label-indicated sets, α, α_i are correctly constructed labels

Figure 4-34 shows the iterative steps in a sample run of INDEPENDENCE-K for $k = 3$.

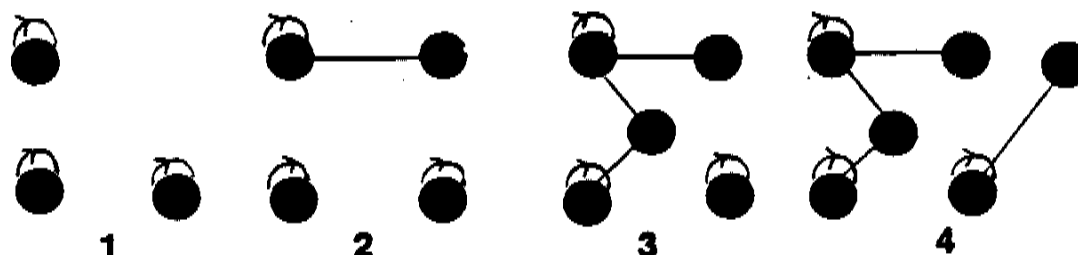


Figure 4-34: INDEPENDENCE-3 in Operation

The floor for graphs with independence number k is $\langle P_{1c}, L_{1nc}, \Sigma_{8c} \rangle$. The elaborate labels are the key to the success of this algorithm. Initially, we have a maximal independent looped set of k vertices. On the first iteration, we add a vertex x , creating the potential for k new sets of k independent vertices and one set of $k+1$ independent vertices. To prevent the formation of $k+1$ independent vertices we deliberately attach x to one of the looped vertices y . Now there are two independent sets of size k , $V - \{y\}$ and $V - \{x\}$. We label x and relabel each v_i in V to reflect the "names" of these two sets and the v_i 's membership/non-membership in each of these two sets. (Such a label is probably a numerical encoding and need not be elaborated upon here. We may "interpret" the loop numerically to make it consistent with the notation for subsequent iterations.) On any subsequent iteration the number and names of each extant set of k independent vertices may be deciphered from the label of any vertex. We carefully attach the new vertex x to one vertex (z_i) in every such set, and then to as many more vertices (y_i) as we choose. The loops refer to the original (but not necessarily the only) set of k -independent vertices. There may be up to $\binom{n}{k-1}$ new independent sets of k vertices formed by this iteration; x and all of V must be relabelled accordingly. INDEPENDENCE-K is correct.

The inverse INDEPENDENCE-K⁻¹ is computed by:

$$f^{-1} = (Z_{v_1 \alpha_1} \dots Z_{v_n \alpha_n} Z_{x \alpha} A_{xy_1}^* A_{xz_1} \dots A_{xz_p} A_x)^{-1}$$

$$\begin{aligned}
&= A_x^{-1} A_{xz_p}^{-1} \dots A_{xz_1}^{-1} (A_{xy_i} A_{xy_i}^*)^{-1} Z_{v\alpha}^{-1} Z_{v_n \alpha_n}^{-1} \dots Z_{v_1 \alpha_1}^{-1} \\
&= D_x D_{xz_p} \dots D_{xz_1} A_{xy_i}^* Z_{v_i \lambda} Z_{v_n \lambda} \dots Z_{v_1 \lambda} \\
\sigma_{pre} &= x \in V, y_i, z_i \in V, z_i \text{ is in the } i\text{th independent set,} \\
&\quad p = \text{number of label-indicated sets, } \alpha, \alpha_i \text{ are correctly} \\
&\quad \text{constructed labels} \\
\sigma^{-1} &= x, y_i, z_i \in V, xx \in E, z_i \text{ is in the } i\text{th independent set,} \\
&\quad p = \text{number of label-indicated sets, } \alpha, \alpha_i \text{ correctly} \\
&\quad \text{constructed labels, } c(x) \text{ indicates } y_i \text{ is not independent} \\
&\quad \text{of } x
\end{aligned}$$

The floor shifts to $\langle P_{2c}, L_{inc}, \Sigma_{6c} \rangle$. Here is an example of an automated inverse computation on which we must improve. In particular, the labels for V must be recalculated to reflect the remaining independent sets of size k , so that

$$f^{-1} = D_x D_{xz_p} \dots D_{xz_1} A_{xy_i}^* Z_{v_i \beta_i} \dots Z_{v_1 \beta_1}$$

This is permissible, we argue, because the next correct label is computable from the encoding. Figure 4-35 shows INDEPENDENCE- K^{-1} operating on a graph $G \in G_p$ and a graph $G \in G_p$ for $k = 4$. On $G \in G_p$, the label maintenance and the fact that xx may not be in E insures a return to the original seed. On $G \in G_p$, there must be k loops, or G will fail. Given $G \in G_p$ with k loops, either some set of vertices labelled independent is not (making some edge unremovable) or some set of more than k vertices is independent (making some vertex unremovable because the appropriate z_i 's cannot be found). In either case G will fail. Thus INDEPENDENCE- K^{-1} is correct and INDEPENDENCE- K is complete.

4.4.5. Graphs with Labelled Edges

In the development of R^c -properties, we specified that the labels or colors be applied to the vertices. If we apply labels to the edges instead (using the operator Z_{xyz} to color edge xy with α), we will call the properties R^e -properties. We immediately extend all the definitions and terminology of vertex labelling to edge labelling, producing the languages $P_{ie}, L_{ie}, L_{ine}, L_{iue}, L_{inue}$ and Σ_{ie} , corresponding to $P_{ic}, L_{ic}, L_{inc}, L_{iuc}, L_{inuc}$ and Σ_{ic} , respectively. The only difference between vertex

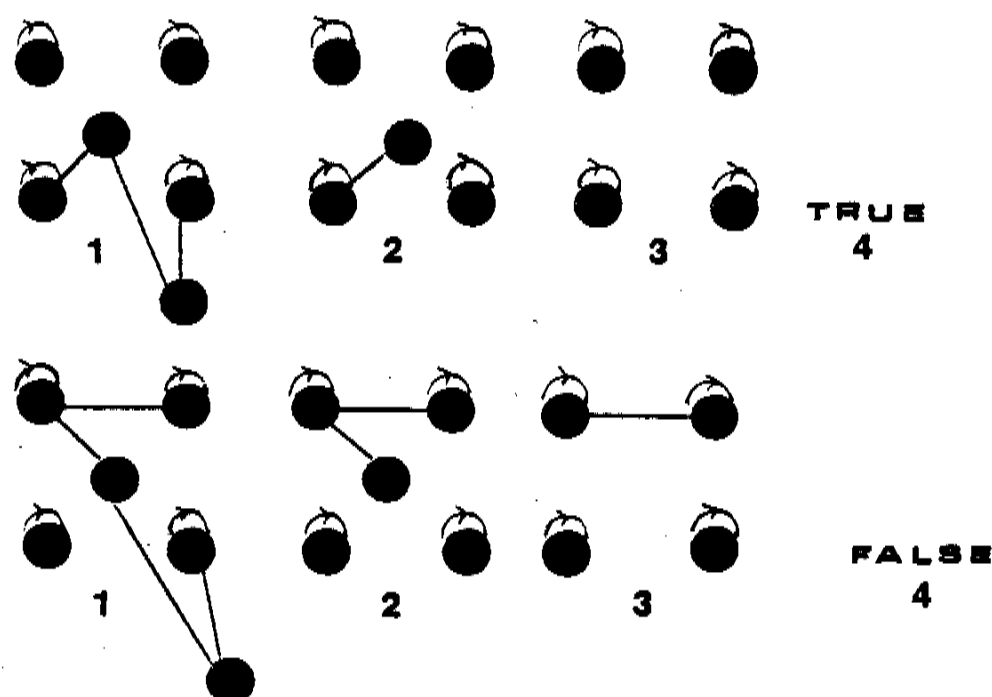


Figure 4-35: INDEPENDENCE-4⁻¹ in Operation

coloring and edge coloring is that no equivalent of loop labelling vertices is available for edges. Edge labelling facilitates the construction of graphs with properties thus far inaccessible.

4.4.6. Graphs with Circumference K

The *circumference* of a graph is the length of any longest cycle it contains. Several examples of graphs with circumference $k = 5$ appear in Figure 4-36. Although labels are not required for the definition of circumference, they do facilitate the construction of graphs with circumference k . We will denote the block partitioning of E by coloring the edges uniformly within a block.

Figure 4-37 shows a graph with four blocks. Any cycle in a graph is totally contained within a single block. We will construct graphs with circumference k using the R^* -property CIRCUMFERENCE- K :

$$(Z_{xy\beta}A_{xy} + A_z + Z_{pq\alpha}B_{pq} + Z_{v_1v_2\alpha}Z_{v_{r-1}v_r\alpha}Z_{v_rv_1\alpha}Y_{v_1-v_r})^*(C_k)$$

where $x, y, x', y' \in V$, $xx', yy' \in E$, $c(xx') = c(yy') = \beta$

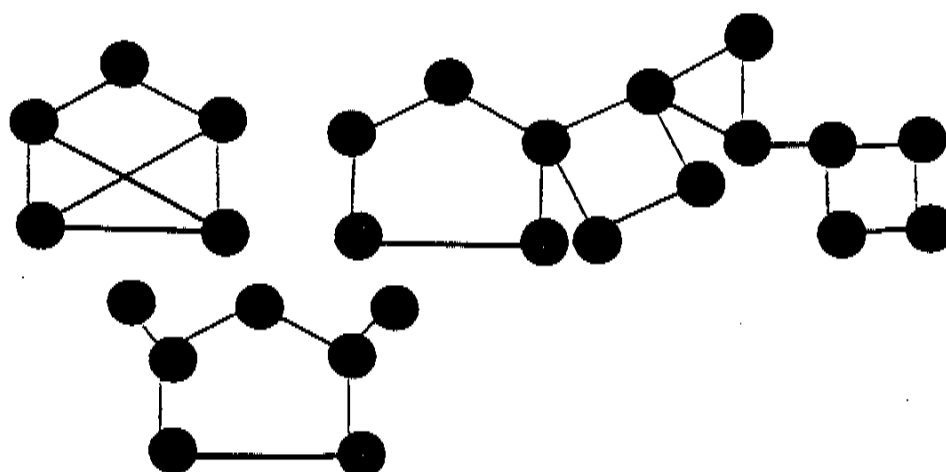


Figure 4-36: Some Graphs with Circumference 5

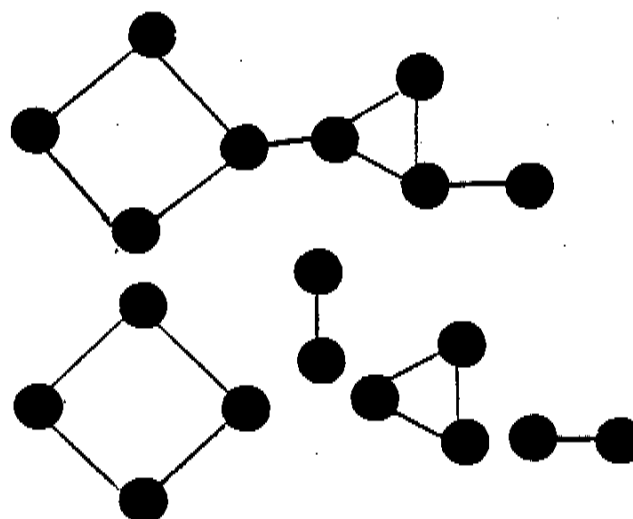


Figure 4-37: A Graph and its Blocks

$p \in V, q \in V, \alpha$ is a new color

$v_1 \in V, v_2, \dots, v_r \in V, \alpha$ is a new color, $r \leq k$

The expression " α is a new color" is an abbreviation for " $c(t_1) \neq \alpha, c(t_2) \neq \alpha, \dots, c(t_n) \neq \alpha, \{t_1, t_2, \dots, t_n\} = V$." The seed, C_k^1 , is the cycle C_k with all its edges colored one. Figure 4-38 shows the iterative steps in a sample run of CIRCUMFERENCE-K for $k = 6$. The floor for CIRCUMFERENCE-K is $\langle P_1, L_0, \Sigma_5 \rangle$.

We will prove the correctness of CIRCUMFERENCE-K as we explain its

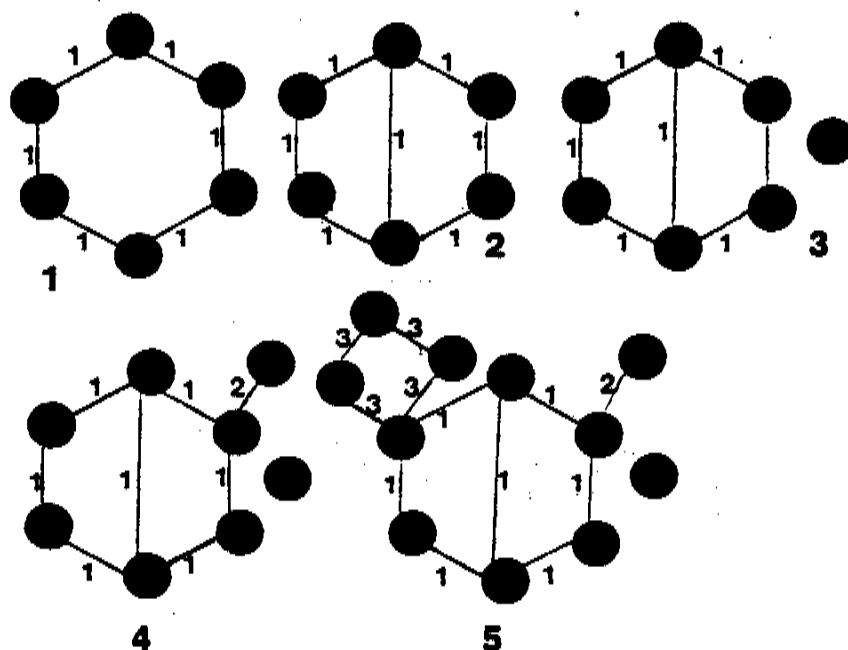


Figure 4-38: CIRCUMFERENCE-6 in Operation

workings. There are four options for growth. First, an edge (xy) may be drawn between any two vertices in the same block and colored the color of their common block. Such an edge can introduce smaller cycles than those already present in the block, but not larger ones. The second option is to begin a new connected component by the addition of an isolated vertex to the graph. (Note that no color is associated with such a vertex at this time.) The next option is to branch from any vertex (p) in the graph to a new vertex (q), beginning a new block and coloring the new edge unlike any currently in the graph. The final option is to add a small enough cycle ($r \leq k$) to the graph, attaching it at v_1 and coloring it unlike any other block currently in the graph. Since each of these options maintains the circumference at k , we have shown CIRCUMFERENCE- K to be correct.

The inverse CIRCUMFERENCE- K^{-1} is computed by:

$$\begin{aligned}
 f^{-1} &= (Z_{xy\beta} A_{xy} + A_z + Z_{pq\alpha} B_{pq} + \\
 &\quad Z_{v_1 v_2 \alpha} \dots Z_{v_{r-1} v_r \alpha} Z_{v_r v_1 \alpha} Y_{v_1 \dots v_r})^{-1} \\
 &= (Z_{xy\beta} A_{xy})^{-1} + A_z^{-1} + (Z_{pq\alpha} B_{pq})^{-1} +
 \end{aligned}$$

$$\begin{aligned}
& (Z_{v_1 v_2} \alpha \dots Z_{v_{r-1} v_r} \alpha Z_{v_r v_2} \alpha Y_{v_1 \dots v_r})^{-1} \\
& = D_{xy} Z_{xy} \lambda + D_z + D_q D_{pq} Z_{pq} \lambda + Y_{v_1 \dots v_r} Z_{v_r v_1} \lambda Z_{v_{r-1} v_r} \lambda \dots Z_{v_1 v_2} \lambda \\
& = x, y, x', y' \in V, xx', yy' \in E, xy \in E, c(xx') = c(yy') = \beta \\
& \quad z \in V \\
& \quad p \in V, q \in V, \alpha \text{ is a new color} \\
& \quad v_1 \in V, v_2, \dots, v_r \in V, \alpha \text{ is a new color, } r \leq k \\
& a^{-1} \\
& = x, y, x', y' \in V, xy, xx', yy' \in E, c(xx') = c(yy') = c(xy), x' \neq y, \\
& \quad y' \neq x, \text{ there exist two edge-disjoint paths from } x \text{ to } y \\
& \quad \text{without edge } xy, \\
& \quad z \in V, d(z) = 0 \\
& \quad p, q \in V, pq \in E, d(q) = 1, c(pq) = \alpha, \alpha \text{ is a unique color} \\
& \quad v_1 \in V, v_1 v_2, \dots, v_{r-1} v_r, v_r v_1 \in E, \\
& \quad |\{rs \mid r, s \in V, rs \in E, c(rs) = c(v_1 v_2)\}| = r, \\
& \quad \text{there exists another cycle of at least size } r
\end{aligned}$$

The floor shifts to $\langle P_{2s}, L_{Qs}, \Sigma_{6s} \rangle$. Figure 4-39 shows CIRCUMFERENCE- k^{-1} operating on a graph $G \in G_p$ and a graph $G \in G_p$ for $k = 3$.

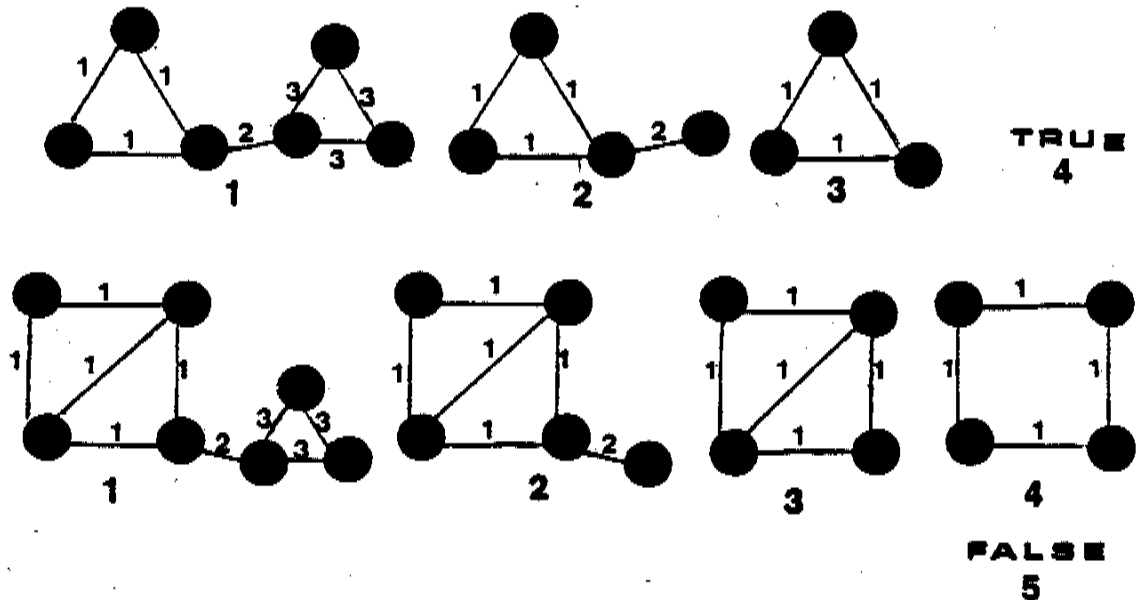


Figure 4-39: CIRCUMFERENCE- 3^{-1} in Operation

The input for the inverse may be edge colored according block in time $O(n)$. Then

CIRCUMFERENCE- K^{-1} deletes isolated vertices (z), small enough cycles attached only at one vertex (v_1), and blocks containing a single edge (pq). It removes edges (xy) internal to a cycle. Thus a graph from G_p will be returned to C_k^α where the color α is not relevant to the isomorphism testing. A graph $G \in G_p$ with overly-small cycles will be reduced to $\langle \phi, \phi \rangle$ and fail, while a graph with an overly-large cycle will retain it and fail. Thus CIRCUMFERENCE- K^{-1} is correct and CIRCUMFERENCE- K is complete.

4.4.7. Graphs with Edge Covering Number K

For a graph $G = \langle V, E \rangle$ an edge subset $E' \subseteq E$ is an *edge cover* if every vertex in V lies on at least one edge in E' . If E' is an edge cover for G , $|E'| = k$ and there is no edge cover for G of smaller cardinality, then G is said to have *edge covering number* k . Several examples of graphs with edge covering number $k = 4$ appear in Figure 4-40.

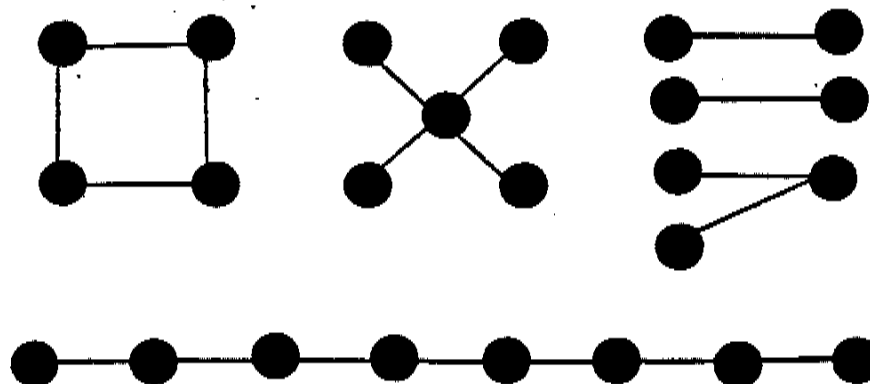


Figure 4-40: Some Graphs with Edge Covering Number 4

Interestingly, the edge covering number of G is bounded by the nature of the minimal spanning tree for G . It is this fact which motivates our approach. If G were connected, the maximum value of n for edge covering number k would be $n = 2k$, where G would be a chain on $2k$ vertices, and the minimum value would be $n = k+1$, as in the star $W_{1,k}$. Since the edge covering number of a graph is the sum of the edge covering numbers of its connected components, for fixed k the seed graphs must represent distinct, additive, non-zero sums of k . For example, if $k = 3$,

we can write $k = 3$, $k = 1 + 2$, or $k = 1 + 1 + 1$. The seeds will be based on the sum, substituting disjoint chains or stars for the integers. We color each seed edge to denote both its connected component and whether it is covering or non-covering. A color is restricted to a single component. Within the p th component two colors appear: even $(2p)$ for covering edges, and odd $(2p - 1)$ for non-covering edges. All edges in a star seed are labelled covering. Edges in a chain seed are alternately labelled covering and noncovering, beginning and ending with the covering color. The set of such appropriately colored seed graphs we will denote as S_k . Figure 4-41 shows the four labelled seed graphs in S_3 .

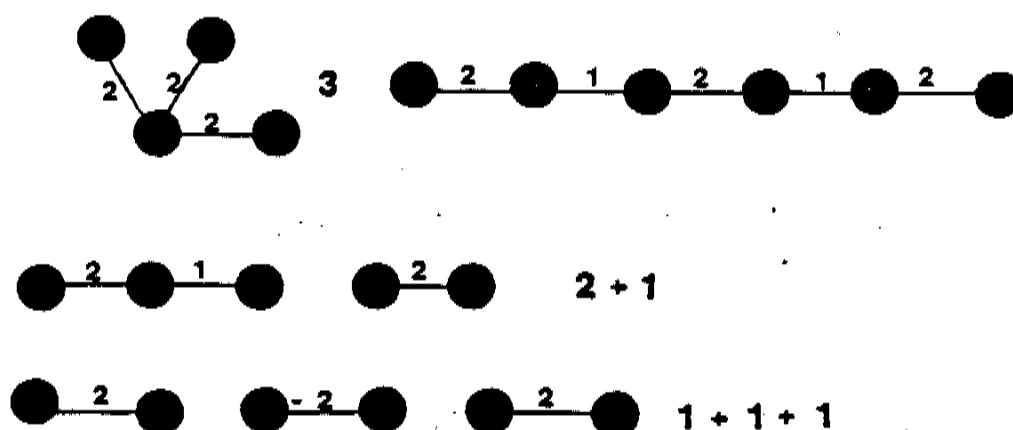


Figure 4-41: The Seed Graphs for Edge Cover 4

Now we can state the R^0 -property K-EDGE-COVER as:

$$(Z_{xyc(xt)-1} A_{xy} F_{xyu} + Z_{vwv(vt)-1} A_{vw})^*(S_k) \text{ where}$$

distinct $x, t, u \in V$, $y \in V$, $xt, xu \in E$, $c(xt) = c(xu)$ is even
 $v, w, t, u \in V$, $vt, wu \in E$, $c(vt) = c(wu)$ even,
 not [$r \in V$, $rv, rw \in E$, $c(rv) = c(rw)$ even]

Figure 4-42 shows the iterative steps in a sample run of K-EDGE-COVER for $k = 6$. The floor for K-EDGE-COVER is $\langle P_{4e}, L_{0e}, \Sigma_{5e} \rangle$. The first option for K-EDGE-COVER fragments a vertex x (on at least two covering edges xt and xu) into two adjacent vertices x and y . The new edge xy is not covering. This operation will be applicable k times to a star on $k+1$ vertices. The operation

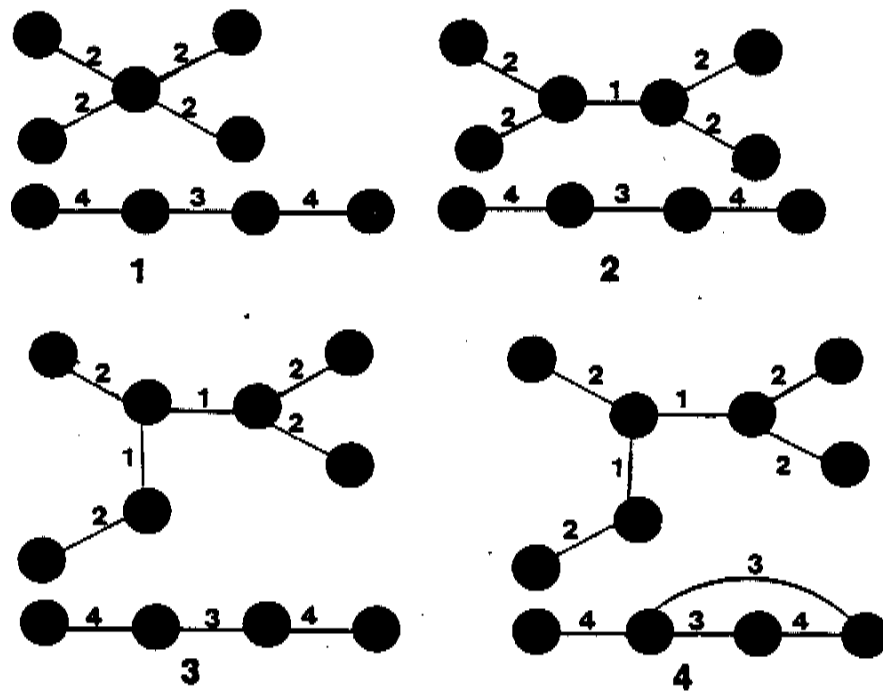


Figure 4-42: 6-EDGE-COVER in Operation

ultimately expands a star on $k+1$ vertices into a tree on $2k+1$ vertices, with alternatingly labelled edges containing at least one vertex of degree three. Such a tree will have covering number k . The second option adds an edge between any two vertices of a component which is as treelike as it can get, i.e., there are no more fragmentable vertices under the first option. No edge will be able to reduce the edge covering number at that point. Thus K-EDGE-COVER is correct.

The inverse K-EDGE-COVER⁻¹ is computed by:

$$\begin{aligned}
 f^{-1} &= (Z_{xyc(xt)-1} A_{xy} F_{xyu} + Z_{vw(vt)-1} A_{vw})^{-1} \\
 &= (Z_{xyc(xt)-1} A_{xy} F_{xyu})^{-1} + (Z_{vw(vt)-1} A_{vw})^{-1} \\
 &= I_{xy} D_{xy} Z_{xy\lambda} + D_{vw} Z_{vw\lambda} \\
 \sigma_{pre} &= \text{distinct } x, t, u \in V, y \in V, xt, xu \in E, c(xt) = c(xu) \text{ is even} \\
 &\quad v, w, t, u \in V, vt, wu \in E, vw \in E, c(vt) = c(wu) \text{ even,} \\
 &\quad \text{not } [r \in V, rv, rw \in E, c(rv) = c(rw) \text{ even}] \\
 \sigma^{-1} &= \text{distinct } x, y, t, u \in V, xy, xt, yu \in E, c(xy) \text{ odd,} \\
 &\quad c(xt) = c(yu) \text{ even, } c(xt) = c(xy) + 1, c(xt) = c(yu).
 \end{aligned}$$

$$|\{rs | rs \in E, c(sz) = c(xy)\}| < |\{rs | rs \in E,$$

$$c(rs) = c(xy) + 1\}|, |V| = |E| + 1$$

$$v, w \in V, vw \in E, c(vw) \text{ odd},$$

$$|\{rs | rs \in E, c(rs) = c(vw)\}| \geq |\{rs | rs \in E,$$

$$c(rs) = c(vw) + 1\}|,$$

$$\text{not}[r \in V, rv, rw \in E, c(rv) = c(rw) \text{ even}]$$

The floor remains constant. Figure 4-43 shows $K\text{-EDGE-COVER}^{-1}$ operating on a graph $G \in G_p$ and a graph $G \in G_p$ for $k = 3$.

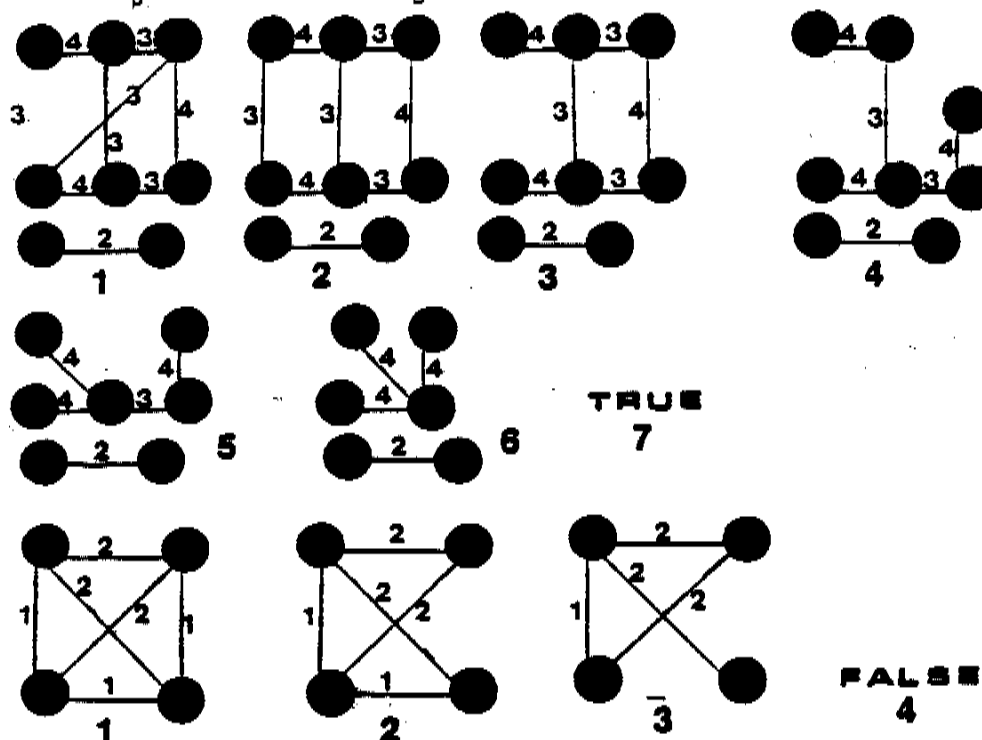


Figure 4-43: 3-EDGE-COVER^{-1} in Operation

The generation process for $K\text{-EDGE-COVER}$ really has two stages: the construction of a spanning tree and the addition of extraneous edges. During the construction of a spanning tree from a star, there will always be more even (covering) edges than odd edges. The inverse exploits this two stage process. For $G \in G_p$, any edge cover spans (touches all) the vertices of G . If the edge cover is connected, such a spanning tree will be contractible into the star or the chain on $\lceil n/2 \rceil$ vertices. If the edge cover is disconnected, it will be contractible into one of the k sum images. For $G \in G_p$, either the number of covering edges is incorrect (and the graph will

ultimately fail) or the indicated edges do not cover the graph. If there is a smaller covering, some uncovered edge will form a cycle and be unremovable. If the covering is inadequate, there will be some chain ending in an uncovered vertex which will not be removed. In either case the graph will fail. Thus $K\text{-EDGE-COVER}^{-1}$ is correct and $K\text{-EDGE-COVER}$ is complete.

4.4.8. Graphs with a k-Factor

A *k-factor* of a graph $G = \langle V, E \rangle$ is a regular subgraph of degree k (>0) which spans V and is not totally disconnected. Several examples of graphs and their 3-factors appear in Figure 4-44.

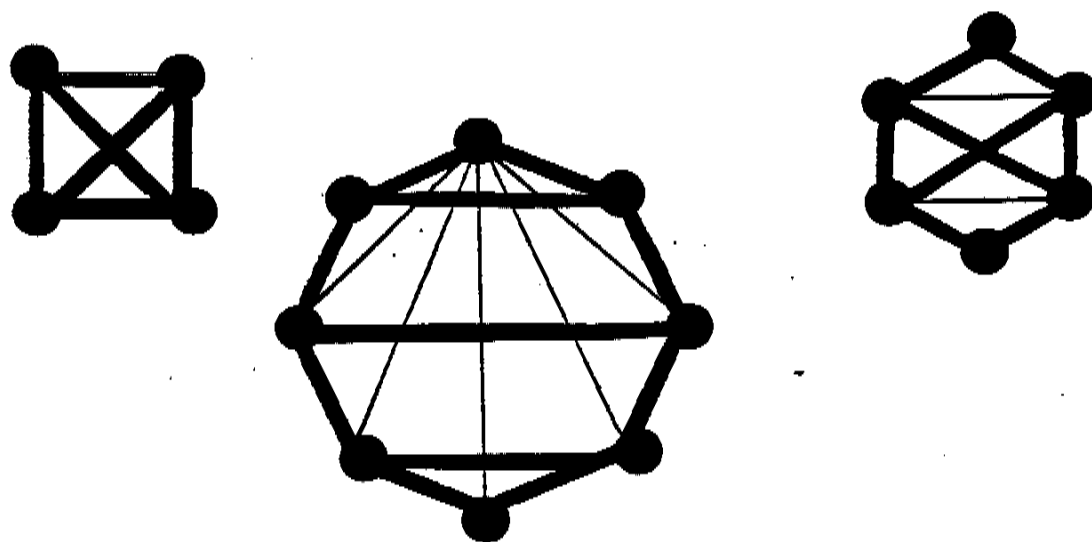


Figure 4-44: Some Graphs with 3-Factors

The factor edges appear darkened in the figure. The R^0 -property $K\text{-FACTOR}$ has separate options for k even and k odd, using the composite operators $EM_{xv_1 \dots v_k}$, $OM_{vw_1 \dots w_{k+1} w_1 \dots w_{k-1}}$, $CM_{v_1 \dots v_k}$ and $FR_{v_1 \dots v_k}$ defined in 3.7.20 for appending even and odd degree vertices without changing the degree of any previously-existing vertex. The R^0 -property $K\text{-FACTOR}$ is

$$\begin{aligned}
 & (Z_{xy\alpha} A_{xy} + EM_{ss_1 \dots s_k} + OM_{vw_1 \dots w_{k+1} w_1 \dots w_{k-1}} + CM_{u_1 \dots u_k} + \\
 & \quad A_{y_2 q} A_{y_1 p} D_{y_1 y_2} CM_{y_1 \dots y_{k+1}} D_{pq} + FR_{z_1 \dots z_k})^*(K_{k+1}) \\
 & \text{where } x, y \in V, xy \in E
 \end{aligned}$$

distinct $s_1, s_2, \dots, s_k \in V$, $s \in V$, $s_{2j-1}s_{2j} \in E$, $c(s_{2j-1}s_{2j}) \neq \alpha$,
 $j = 1, 2, \dots, k/2$; k even

distinct $v_1, v_2, \dots, v_{k+1} \in V$, distinct $w_1, w_2, \dots, w_{k-1} \in V$, distinct $v, w \in V$,
 $v_{2i-1}v_{2i}, w_{2j-1}w_{2j} \in E$, $c(v_{2i-1}v_{2i}) \neq \alpha$, $c(w_{2j-1}w_{2j}) \neq \alpha$,
 $i = 1, 2, \dots, (k+1)/2$; $j = 1, 2, \dots, (k-1)/2$; k odd

distinct $u_1, u_2, \dots, u_k \in V$

distinct $p, q \in V$, distinct $y_1, y_2, \dots, y_{k+1} \in V$, $pq \in E$, $c(pq) \neq \alpha$

distinct $x_1, x_2, \dots, x_k, z_1 \in V$, distinct $z_2, z_3, \dots, z_k \in V$, $z_1x_i \in E$, $i = 1, 2, \dots, k$;
 $c(z_1x_i) \neq \alpha$

Figure 4-45 shows the iterative steps in a sample run of K-FACTOR for $k = 4$.

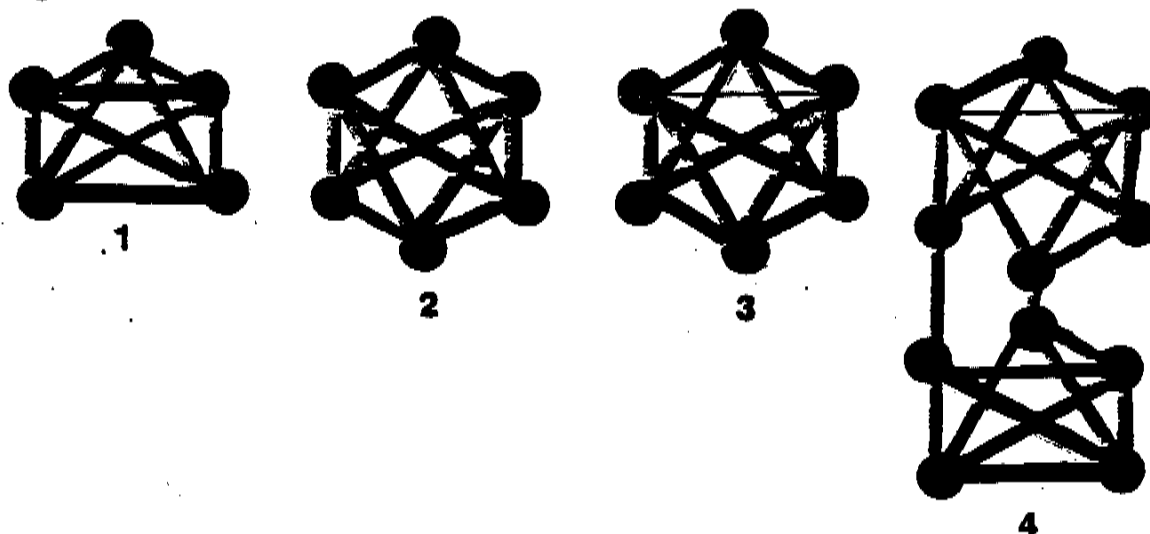


Figure 4-45: 4-FACTOR in Operation

Throughout its execution, K-FACTOR distinguishes the edges in the factor (unlabelled) from the edges not in the factor (labelled α). The floor for graphs with a k -factor is $\langle P_{2\alpha}, L_{1\alpha}, \Sigma_{2\alpha} \rangle$.

K-FACTOR has six options. The first adds a non-factor edge (xy). The second, applicable only for even k , alters the k -factor correctly, appending a single new vertex (s) without changing the previously-existing degrees of any of the vertices. All the new edges are unlabelled and appear in the factor. The third, applicable only for odd k , alters the k -factor correctly by appending two new vertices (w and v) without changing the previously-existing degrees of any of the

vertices. Again, all the new edges appear in the factor. The fourth option adds a set of $k+1$ vertices $(u_1, \dots, u_k)$ simultaneously to the k -factor, with a complete subgraph on them. The fifth option adds $k+1$ vertices to the k -factor, replacing a previously-existing edge with $1 + (k+1)k/2$ edges. The sixth option adds $k-1$ vertices to the k -factor, replacing a previously-existing vertex with a copy of K_k , each of whose vertices maintains one of the old vertex's previous adjacencies. Clearly K -FACTOR is correct.

The inverse K -FACTOR $^{-1}$ is computed by:

$$\begin{aligned}
 f^{-1} &= (Z_{xy\alpha} A_{xy} + EM_{ss_1 \dots s_k} + OM_{vw_1 \dots v_{k+1} w_1 \dots w_{k-1}} + CM_{u_1 \dots u_k} \\
 &\quad + A_{y_2 q} A_{y_1 p} D_{y_1 y_2} CM_{y_1 \dots y_{k+1}} D_{pq} + FR_{z_1 \dots z_k})^{-1} \\
 &= (Z_{xy\alpha} A_{xy})^{-1} + EM_{ss_1 \dots s_k}^{-1} + OM_{vw_1 \dots v_{k+1} w_1 \dots w_{k-1}}^{-1} + \\
 &\quad CM_{u_1 \dots u_k}^{-1} + A_{y_2 q} A_{y_1 p} D_{y_1 y_2} CM_{y_1 \dots y_{k+1}} D_{pq} + FR_{z_1 \dots z_k}^{-1} \\
 &= D_{xy} Z_{xy\alpha} + EM_{ss_1 \dots s_k}^{-1} + OM_{vw_1 \dots v_{k+1} w_1 \dots w_{k-1}}^{-1} + CM_{u_1 \dots u_k}^{-1} \\
 &\quad + A_{pq} CM_{y_1 \dots y_{k+1}}^{-1} A_{y_1 y_2} D_{y_1 p} D_{y_1 q} + FR_{z_1 \dots z_k}^{-1} \\
 \sigma_{pre} &= x, y \in V, xy \in E \\
 &\quad \text{distinct } s_1, s_2, \dots, s_k \in V, s \in V, s_{2j-1} s_{2j} \in E, \\
 &\quad c(s_{2j-1} s_{2j}) \neq \alpha, j = 1, 2, \dots, (k+1)/2; j = 1, 2, \dots, (k-1)/2; \\
 &\quad k \text{ odd, } d(s_i) = k, i = 1, 2, \dots, k \\
 &\quad \text{distinct } v_1, v_2, \dots, v_{k+1} \in V, \text{ distinct } w_1, w_2, \dots, w_{k-1} \in V, \\
 &\quad \text{distinct } v, w \in V, v_{2i-1} v_{2i} w_{2j-1} w_{2j} \in E, \\
 &\quad c(v_{2i-1} v_{2i}) \neq \alpha, c(w_{2j-1} w_{2j}) \neq \alpha, \\
 &\quad i = 1, 2, \dots, (k+1)/2; j = 1, 2, \dots, (k-1)/2; k \text{ odd: } d(v_i) = k, \\
 &\quad i = 1, 2, \dots, k+1; d(w_i) = k, i = 1, 2, \dots, k-1 \\
 &\quad \text{distinct } u_1, u_2, \dots, u_k \in V \\
 &\quad \text{distinct } p, q \in V, \text{ distinct } y_1, y_2, \dots, y_{k+1} \in V, pq \in E \\
 &\quad d(p) = k, d(q) = k \\
 &\quad \text{distinct } x_1, x_2, \dots, x_k, z_1 \in V, \text{ distinct } z_2, z_3, \dots, z_k \in V, \\
 &\quad z_i x_i \in E, i = 1, 2, \dots, k, d(z_i) \neq k \\
 \sigma^{-1} &= x, y \in V, xy \in E, c(xy) = \alpha
 \end{aligned}$$

distinct $s, s_1, s_2, \dots, s_k \in V$, $s_{2j-1}s_{2j} \in E$, $c(s_{2j-1}s_{2j}) \neq \alpha$,
 $j = 1, 2, \dots, k/2$; k even, $d(s) = k$
 distinct $v, v_1, v_2, \dots, v_{k+1} \in V$, distinct $w, w_1, w_2, \dots, w_{k-1} \in V$,
 $vw, vv_1, ww_1 \in E$, $v_{2i-1}v_{2i}, w_{2j-1}w_{2j} \in E$,
 $c(v_{2i-1}v_{2i}) \neq \alpha$, $c(w_{2j-1}w_{2j}) \neq \alpha$, k odd, $d(v) = k$,
 $d(w) = k$, $d(v_i) = k$, $d(w_j) = k$, $i = 1, 2, \dots, (k+1)/2$;
 $j = 1, 2, \dots, (k-1)/2$;
 distinct $u_1, u_2, \dots, u_k \in V$,
 $|\{pu_i \mid p \in V\}| = |\{u_i u_j \mid u_i u_j \in E\}| = k(k-1)/2$
 distinct $p, q, y_1, y_2, \dots, y_{k+1} \in V$, $pq \in E$, $y_i y_j \in E$, $d(y_i) = k$,
 $i = 1, 2, \dots, k+1$, $j = 1, 2, \dots, k+1$ except $y_1 y_2$;
 $y_1 p, y_2 q \in E$
 distinct $x_i, z_i \in V$, $x_i z_i \in E$, $d(x_i) = k$, $d(z_i) = k$, $i = 1, 2, \dots, k$;
 $x_i z_j \in E$, $j = 2, 3, \dots, k$; $z_{rs} \in E$, $r, s = 1, 2, \dots, k$

The floor shifts to $\langle P_{2\alpha}, L_{1\alpha}, \Sigma_{3\alpha} \rangle$. Figure 4-46 shows $K\text{-FACTOR}^{-1}$ operating on a graph $G \in G_p$ and a graph $G \in G_p$ for $k = 2$. In the figure the unlabelled factor edges are darkened. On a graph with a correctly (un)labelled k -factor, $K\text{-FACTOR}^{-1}$ will remove the irrelevant edges, contract connected components to K_{k+1} (assuming the completeness of EVEN-REGULAR and ODD-REGULAR), and remove all but one of the K_{k+1} 's, until G succeeds. On a graph $G \in G_p$, $K\text{-FACTOR}^{-1}$ will remove the irrelevant edges and then discover that some vertex of degree not equal to k is irremovable, warranting failure. Thus $K\text{-FACTOR}^{-1}$ is correct and $K\text{-FACTOR}$ is complete.

4.4.9. K -Factorable Graphs

Let $G_1 = \langle V_1, E_1 \rangle$ and $G_2 = \langle V_2, E_2 \rangle$ be two graphs. The *union* of G_1 and G_2 is a new graph $G = \langle V, E \rangle$ where $V = V_1 \cup V_2$ and $E = E_1 \cup E_2$. If a graph G is the union of a finite set of k -factors, we say that G is *k -factorable*. Several examples of 2-factorable graphs appear in Figure 4-47. Note that the degree of every vertex in a k -factorable graph is a multiple of k .

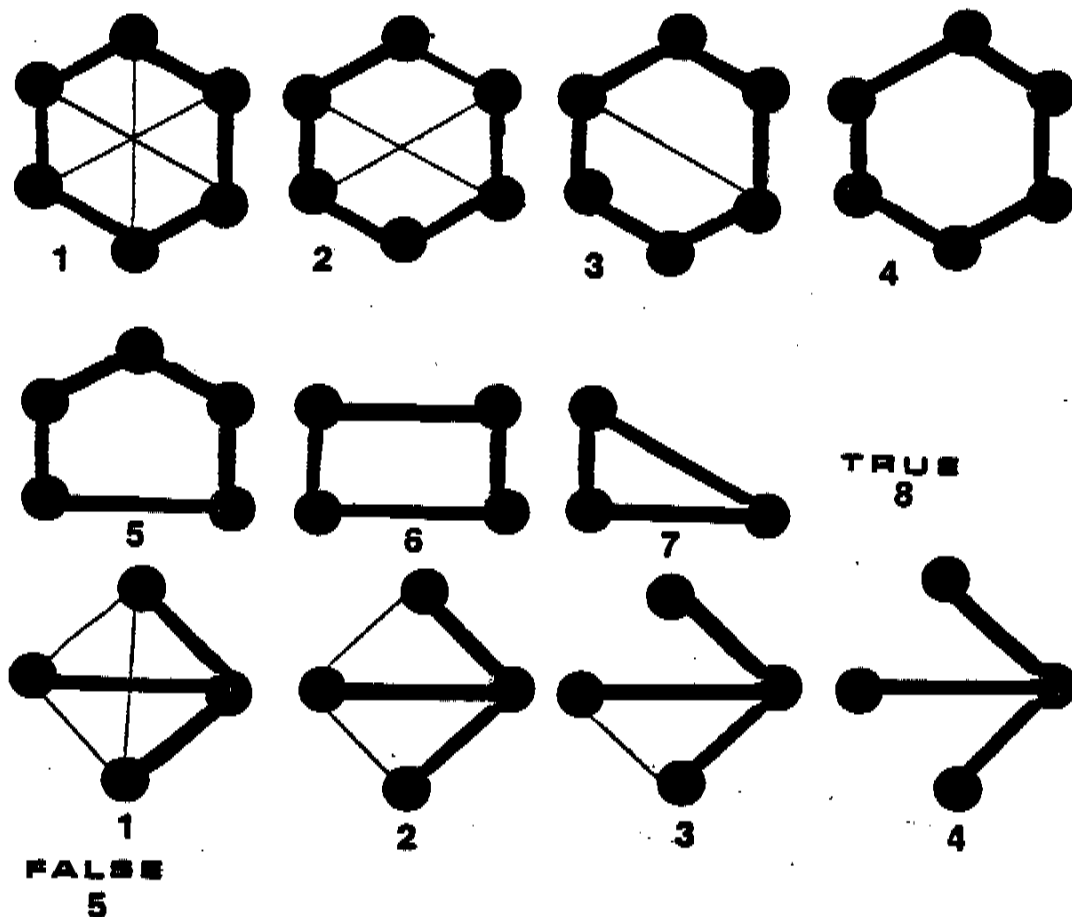
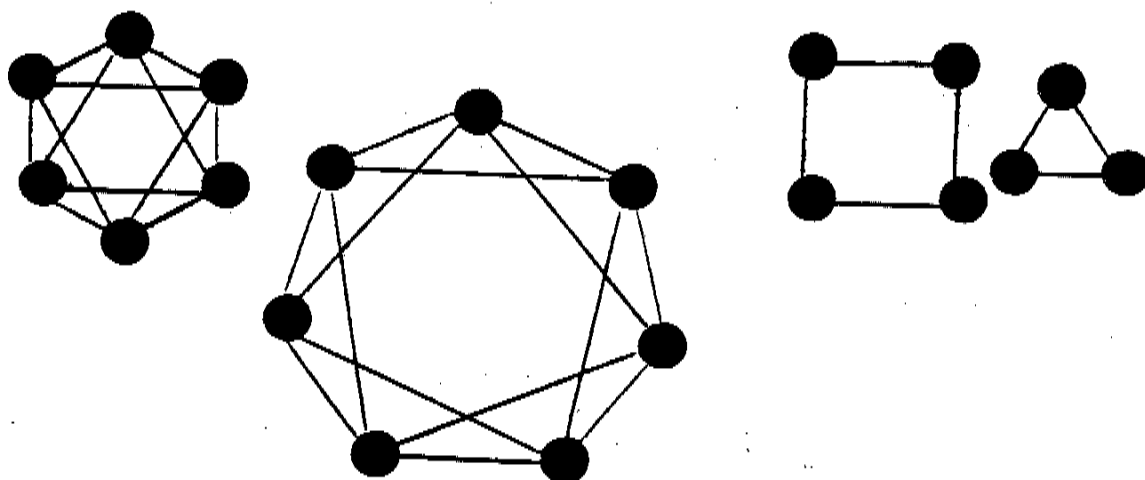
Figure 4-46: 2-FACTOR⁻¹ in Operation

Figure 4-47: Some 2-Factorable Graphs

The R^{*+} -property K-FACTORABLE is:

$$(Z_{x_1 x_2^{p+1}} \dots Z_{x_{nk/2-1} x_{nk/2}^{p+1}} A_{x_1 x_2} \dots A_{x_{nk/2-1} x_{nk/2}} (\prod_{i=1}^p f^i) (\prod_{i=1}^p g^i) (Q_{k+1}, 1))$$

where $x_i \in V$, $|\{x_i\}| = n$, every $x \in V$ appears exactly k times in $\{x_i\}$.

$$x_{2j-1} x_{2j} \in E, j = 1, 2, \dots, nk/4$$

f is applied according to EVEN-REGULAR, k even, same vertices restriction

g is applied according to ODD-REGULAR, k odd, same vertices restriction

Throughout its execution, K-FACTORABLE distinguishes each factor by a unique edge label. Figure 4-48 shows the iterative steps in a sample run of K-FACTORABLE for $k = 3$. In the figure the edges of one factor appear darkened. The floor for k -factorable graphs is $\langle P_{2e}, L_{1ne}, \Sigma_{2e} \rangle$. In an attempt to make this algorithm readable, we have abbreviated it somewhat. The first operator adds an entire new k -factor to the graph and appears exactly once on each iteration. Recall that p is the register value representing the number of k -factors composing G . The second operator, f^i , denotes an application of EVEN-REGULAR in which the i th factor is expanded to cover (1 or $k+1$ or $k-1$) more vertices and the new edges are appropriately labelled for their factor. The selector is intended to indicate that the same vertices must be added to each factor under $\prod_{i=1}^p f^i$ by each application of f . The third operator, g^i , denotes an application of ODD-REGULAR in which the i th factor is expanded to cover (2 or $k+1$ or $k-1$) more vertices and the new edges are appropriately labelled for their factor. The selector is intended to indicate that the same vertices must be added to each factor under $\prod_{i=1}^p g^i$ by each application of g . Within any iteration each of these last two operators may be applied any number of times without changing the number (p) of k factors. K-FACTOR is correct.

The inverse K-FACTORABLE⁻¹ is computed by:

$$\begin{aligned} f^{-1} &= (Z_{x_1 x_2^{p+1}} \dots Z_{x_{nk/2-1} x_{nk/2}^{p+1}} A_{x_1 x_2} \dots A_{x_{nk/2-1} x_{nk/2}} (\prod_{i=1}^p f^i) (\prod_{i=1}^p g^i))^{-1} \\ &= (\prod_{i=p}^1 (g^i)^{-1}) (\prod_{i=p}^1 (f^i)^{-1}) (Z_{x_1 x_2^{p+1}} \dots Z_{x_{nk/2-1} x_{nk/2}^{p+1}} A_{x_1 x_2} \dots A_{x_{nk/2-1} x_{nk/2}})^{-1} \end{aligned}$$

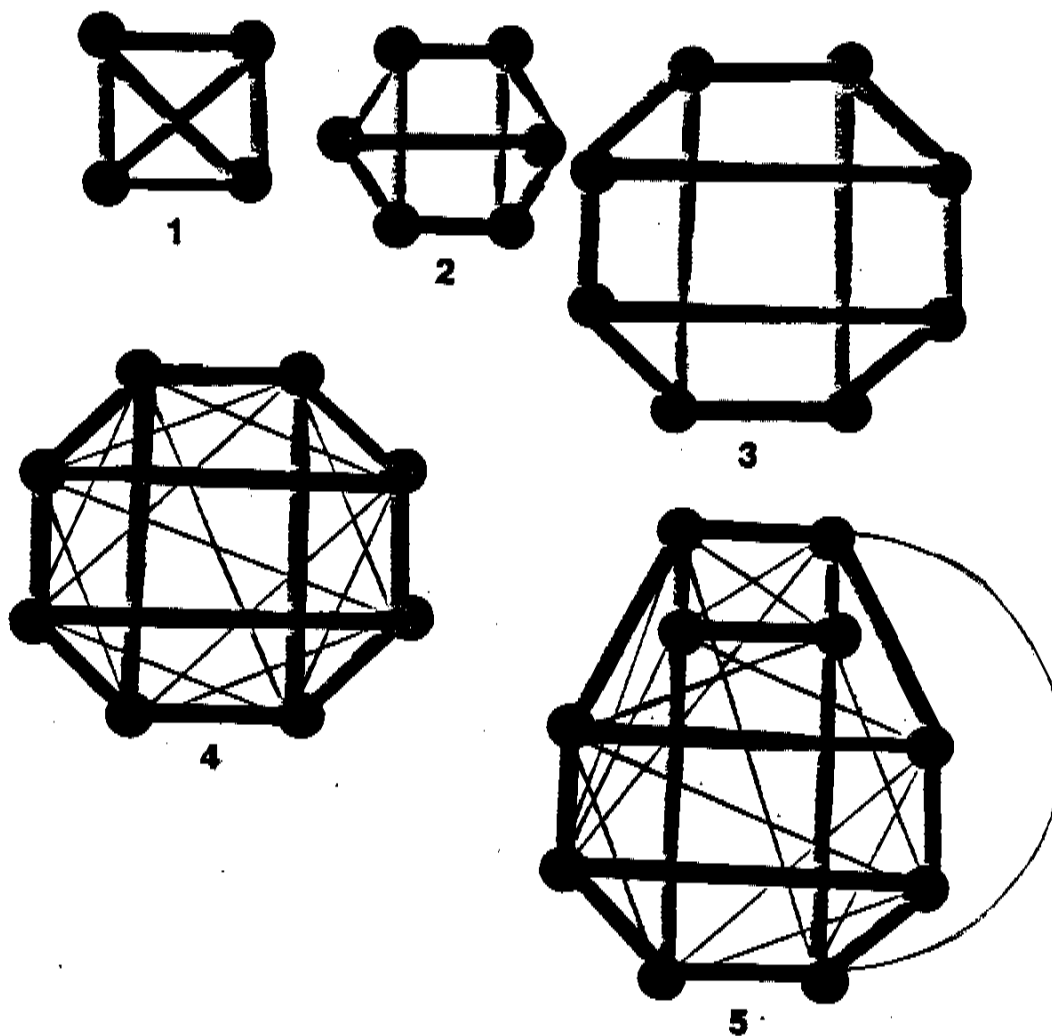


Figure 4-48: 3-FACTORABLE in Operation

$$\begin{aligned}
 &= (\prod_{i=p}^1 (g^{-1})^i) (\prod_{i=p}^1 (f^{-1})^i) D_{x_{nk/2-1} x_{nk/2}} \dots D_{x_1 x_2} \\
 &\quad Z_{x_{nk/2-1} x_{nk/2}} \lambda^{-1} Z_{x_1 x_2} \lambda \\
 \sigma_{\text{pre}} = \sigma &= x_i \in V, |\{x_i\}| = n, \text{ every } x \in V \text{ appears exactly } k \\
 &\quad \text{times in } \{x_i\}, x_{2j-1} x_{2j} \in E, j = 1, 2, \dots, nk/4 \\
 &\quad f \text{ is applied according to EVEN-REGULAR, } k \text{ even,} \\
 &\quad \text{same vertices restriction} \\
 &\quad g \text{ is applied according to ODD-REGULAR, } k \text{ odd,} \\
 &\quad \text{same vertices restriction} \\
 \sigma^{-1} &= x_i \in V, |\{x_i\}| = n, \text{ every } x \in V
 \end{aligned}$$

appears exactly k times in $\{x_i\}$, $x_{2i-1}x_{2i} \in E$.

$c(x_{2i-1}x_{2i}) = p$, $i = 1, 2, \dots, nk/4$

f^{-1} is applied according to EVEN-REGULAR $^{-1}$, k even,

same vertices restriction

g^{-1} is applied according to ODD-REGULAR $^{-1}$, k odd,

same vertices restriction

We are presuming that f^{-1} and g^{-1} are relabelling the restored edges correctly. The floor shifts to $\langle P_{2e}, L_{1ne}, \Sigma_{5e} \rangle$. Figure 4-49 shows K-FACTORABLE $^{-1}$ operating on a graph $G \in G_p$ and a graph $G \notin G_p$ for $k = 2$.

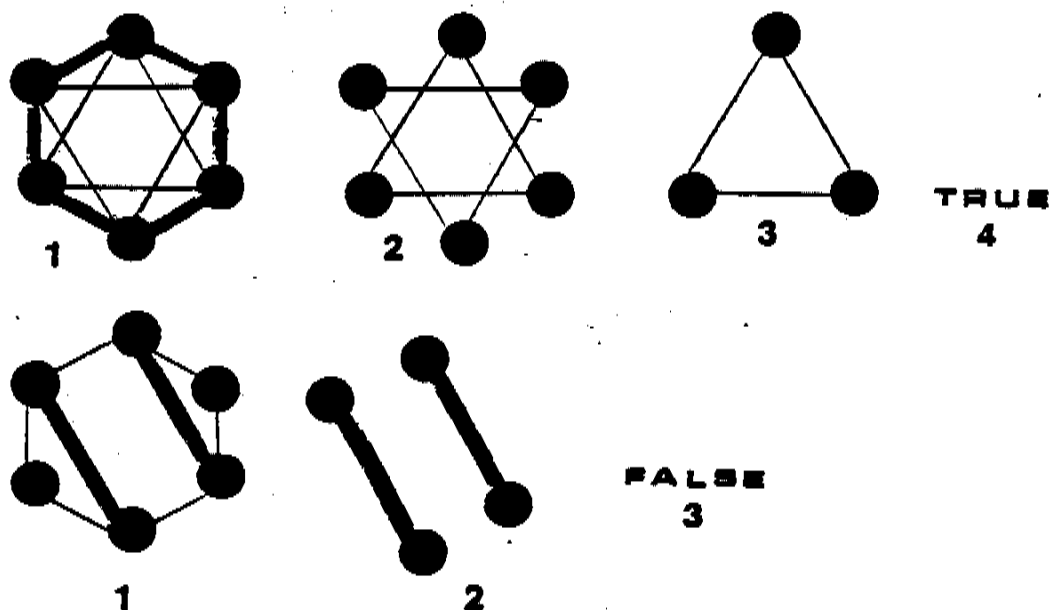


Figure 4-49: 2-FACTORABLE $^{-1}$ in Operation

On each iteration K-FACTORABLE $^{-1}$ removes an entire, correctly labelled k -factor and deletes as many correctly attached vertices as possible from the graph (assuming the completeness of EVEN-REGULAR and ODD-REGULAR). If no k -factor can be found, $G \notin G_p$ and will fail. K-FACTORABLE $^{-1}$ is correct and K-FACTORABLE is complete.

4.5. Subsumption

Having demonstrated our ability to describe most graph properties in some extension of our original recursive format, we discuss in this section one merit of such a representation: subsumption.

Given two recursively-formulated properties (in the same R-language, R^+ -language, R^c -language or R^* -language) $p_1 = \langle f_1, S_1, \sigma_1 \rangle$ and $p_2 = \langle f_2, S_2, \sigma_2 \rangle$, we say that property p_1 *subsumes* property p_2 if every graph with property p_2 also has property p_1 . If p_1 subsumes p_2 , p_2 is a special case of p_1 . In addition, if property p_1 subsumes property p_2 and property p_2 subsumes property p_1 , then p_1 and p_2 are *equivalent properties*. The recursive formulation makes a test for subsumption quite simple. p_1 subsumes p_2 if and only if:

- f_2 is subsumed by f_1
- $p_1^{-1}(S)$ is TRUE for every $S \in S_2$
- σ_2 is subsumed by σ_1

Thus we need only specify how operators and selectors subsume each other. We first define selector subsumption. Let σ_1 and σ_2 be selectors which select vertex sets V_1 and V_2 with respect to a graph G . We say that selector σ_1 subsumes selector σ_2 if and only if there exists a mapping $\phi: V_2 \rightarrow V_1$ such that

- ϕ is a function (i.e., $\phi(v)$ is unique for each $v \in V_2$)
- ϕ is one-to-one (i.e., $\phi(v) = \phi(v')$ if and only if $v = v'$)
- ϕ preserves the following relationships:
 - $u \in V$
 - $v \in V$
 - vertex color
 - edge color
- a vertex v selected by σ_2 will always be accepted as $\phi(v)$ by σ_1

We can be certain that σ_1 will accept v as $\phi(v)$ if and only if the description of $\phi(v)$ in σ_1 is consistent with and no more restrictive than the description of v in σ_2 . The following is a list of such relationships, where "expr" denotes the degree

of a vertex or the cardinality of a set

- "description 1 on v " is less restrictive than "description 1 and description 2 on v ." For example, " $v \in V$ " is less restrictive than " $v \in V, vw \in E$." The first permits v to be isolated, the second does not.
- " $\text{expr} < k_1$ " is less restrictive than " $\text{expr} < k_2$ " if $k_1 > k_2$.
- " $\text{expr} \leq k_1$ " is less restrictive than " $\text{expr} \leq k_2$ " if $k_1 > k_2$.
- " $\text{expr} > k_1$ " is less restrictive than " $\text{expr} > k_2$ " if $k_1 < k_2$.
- " $\text{expr} \geq k_1$ " is less restrictive than " $\text{expr} \geq k_2$ " if $k_1 < k_2$.
- " $\text{expr} \leq k$ " is less restrictive than " $\text{expr} < k$ ".
- " $\text{expr} \leq k$ " is less restrictive than " $\text{expr} = k$ ".
- " $\text{expr} \geq k$ " is less restrictive than " $\text{expr} > k$ ".
- " $\text{expr} \geq k$ " is less restrictive than " $\text{expr} = k$ ".
- " $d(v) = \text{max}$ " is consistent with " $d(v) \text{ rel } k$ " only if rel is $=$ and k is max or rel is $\geq, >, \leq, <$ when $k \leq \text{max}, k < \text{max}, k \geq \text{max}, k > \text{max}$, respectively
- " $d(v) < \text{max}$ " is consistent with " $d(v) = k$ " only if $k < \text{max}$.
- " $d(v) < \text{max}$ " is consistent with " $d(v) > k$ " only if there exists an integer i such that $k < i < \text{max}$.
- " $d(v) < \text{max}$ " is consistent with " $d(v) \geq k$ " only if $k \leq \text{max}$.

We offer the following example of selector subsumption:

σ_1 : $x, y \in V, xy \in E, d(x) \geq 2$

σ_2 : $x, y \in V, xy \in E, d(y) > 2, d(x) = 1$

σ_1 subsumes σ_2 under the mapping $\phi(x) = y$ and $\phi(y) = x$

We postulate the following conditions for operator subsumption:

Condition 1

f subsumes f .

Condition 2

$f + g$ subsumes f . Clearly f is a special case of " f or g ."

Condition 3

f^* subsumes f^k . We may choose to iterate k times, and f^k is a special case of f^* .

Condition 4

f^* subsumes N . We may also choose to iterate f no times, and the null primitive N is a special case of f^* .

These operator subsumption conditions may be combined in fairly lengthy reasoning procedures. For example, if $f_1 = (f + g^*)(f^2 + g)$ we can rewrite f_1 as

$$f_1 = f^3 + fg + g^*f^2 + g^*g$$

Then we can show that f_1 subsumes, among others, each of the following:

$$f^3$$

$$g^5f^2$$

$$(f + g^*)f^2$$

$$g$$

$$fg + g$$

We are now ready to demonstrate the hierarchical concepts inherent in our representation of graph theory. We offer a simple example. "Every chain is a tree." We "prove" this by examining the R-properties p_1 (TREE):

$$B_{xy}^*(K_1) \text{ where } x \in V, y \in V$$

and p_2 (CHAIN):

$$B_{xy}^*(K_2) \text{ where } x \in V, y \in V, d(x) = 1$$

B_{xy} subsumes itself. $p^{-1}(K_2)$ returns TRUE. Define $\phi(x) = x$, $\phi(y) = y$. Thus our R-language "knows" the relationship between trees and chains. The R-language representations are inherently capable of reasoning out hierarchical relationships.

4.6. Merger

Subsumption is one merit of our recursive formulation. This section describes another, a rigorous way to combine graph properties. Several sample mergers are offered.

Given a graph property p_1 and a graph property p_2 , we define their *merger* to be a graph property $p = p_1 \wedge p_2$ (read " p_1 and p_2 ") which is the set of all graphs with both properties, i.e., $G_{p_1 \wedge p_2} = G_{p_1} \cap G_{p_2}$. In the context of our recursive representation, $p_1 = \langle f_1, S_1, \sigma_1 \rangle$, $p_2 = \langle f_2, S_2, \sigma_2 \rangle$, and the merger is a new algorithm $p = \langle f, S, \sigma \rangle$ generating exactly the set

$$\{G \mid p_1^{-1}(G) = \text{TRUE}, p_2^{-1}(G) = \text{TRUE}\}$$

One of the strengths of our recursive representation is that merger appears to be reasonably amenable to automatic computation. Given p_1 and p_2 , we will develop a series of principles for constructing p . We do not claim that every merger can be computed from these principles. We do claim that any merger constructed from these principles is correct. The principles are assembled gradually, each one motivated by an example.

Let p_1 be TREE and p_2 be CHAIN. Their merger p is clearly CHAIN, since p_1 subsumes p_2 .

PRINCIPLE 1

If p_1 subsumes p_2 , the merger of p_1 and p_2 is simply p_2 .

Many attempts at merger, upon examination, become simple cases of subsumption. Examples of this include:

- ACYCLIC and STAR = STAR
- CYCLE and EULERIAN = CYCLE
- WHEEL and PINWHEEL = WHEEL
- BICONNECTED and CONNECTED = BICONNECTED
- K-COLORABLE and K-CHROMATIC = K-CHROMATIC

We recall that $p_1 = \langle f_1, S_1, \sigma_1 \rangle$ subsumes $p_2 = \langle f_2, S_2, \sigma_2 \rangle$ only if f_1 subsumes f_2 , σ_1 subsumes σ_2 , and $p_1^{-1}(S) = \text{TRUE}$ for every $S \in S_2$. We will now examine variants where subsumption is not possible because one of these conditions fail. Whenever possible, we will state p_1 and p_2 with variable names which suggest the direction the merger should pursue. An automated version would, of course, need to search for such pairings. Consider the following example:

$$p_1: (B_{xy} + A_{wz})^*(K_3, K_4, K_5) \\ \text{where } x \in V, y \in V, d(x) > 2 \\ w, z \in V$$

$$p_2: B_{xy}^*(K_3) \text{ where } x \in V, y \in V, d(x) \text{ even}$$

Property p_1 generates K_3 and graphs with a "center" subgraph of K_4 or K_5 . Property p_2 generates only four graphs, K_3 with a branch possible on any vertex. Although f_1 subsumes f_2 and $p_1^{-1}(K_3)$ is TRUE, neither σ subsumes the other and it is their combination we desire, i.e.,

$$p: B_{xy}^*(K_3) \text{ where } x \in V, y \in V, d(x) > 2, d(x) \text{ even}$$

This is particularly interesting because p cannot iterate; the merger consists only of K_3 . We have arrived at

PRINCIPLE 2

If f_1 subsumes f_2 and $p_1^{-1}(S) = \text{TRUE}$ for every $S \in S_2$, then the merger p is $\langle f_2, S_2, \sigma \rangle$. The variables are mapped so as to demonstrate the subsumption of f_2 by f_1 and so that σ eliminates any references to variables not in σ_2 . If σ_2 subsumes σ_1 , σ will be simply σ_1 .

Consider next the example:

$$p_1: (B_{xy} + A_{wz})^*(K_3, K_4) \\ \text{where } x \in V, y \in V \\ w, z \in V$$

$$p_2: B_{xy}^*(K_3, K_5) \text{ where } x \in V, y \in V, d(x) \text{ even}$$

The p_1 graphs have "center" subgraphs of K_3 or K_4 ; p_2 graphs are tree-like and have "center" subgraphs of K_3 or K_5 . Their merger demands a common seed:

$$p: B_{xy}^*(K_3) \text{ where } x \in V, y \in V, d(x) \text{ even}$$

We can now postulate:

PRINCIPLE 3

If f_1 subsumes f_2 , σ_1 subsumes σ_2 and $S_1 \cap S_2 \neq \emptyset$, then the merger p is $\langle f_2, S_1 \cap S_2, \sigma_2 \rangle$. Again we assume a proper mapping of the variables.

The most difficult variant is when f_1 does not subsume f_2 . Consider next the example:

$$\begin{aligned} p_1: & (A_x A_y A_z + A_{pq})^*(K_1) \\ & \text{where distinct } x, y, z \in V \\ & \text{distinct } p, q \in V \\ p_2: & (A_x A_y + A_{pq})^*(K_1) \\ & \text{where distinct } x, y \in V \\ & \text{distinct } p, q \in V \end{aligned}$$

Property p_1 adds vertices three at a time, p_2 two at a time, to K_1 . The merger must deal with the fact that p_1 graphs have $n \equiv 1 \pmod{3}$ and p_2 graphs have $n \equiv 1 \pmod{2}$. The most complete solution is $n \equiv 1 \pmod{6}$, where

$$\begin{aligned} p: & (A_{x_1} A_{x_2} \dots A_{x_6} + A_{pq})^*(K_1) \text{ where} \\ & \text{distinct } x_i \in V, i = 1, 2, \dots, 6 \\ & \text{distinct } p, q \in V \end{aligned}$$

We observe at this time that incremental graph algorithms "grow" graphs in iterative steps. We denote the change in n after a single iteration of p_i as Δn_i , and the change in m as Δm_i . We define Δn and Δm correspondingly for property p .

PRINCIPLE 4

If p is the merger of p_1 and p_2 , Δn is the least common multiple of Δn_1 and Δn_2 , and Δm of Δm_1 and Δm_2 .

Clearly principle 4 is only guidance for dealing with uncooperative f 's. Thus far, most of our examples have been on "toy" graph properties, that is, ones artificially constructed to make a point. When we attempt to apply these principles to "real" properties, our experience suggests some techniques for f construction.

First, composite operators may obscure the nature of f_1 and f_2 ; rewrite them in terms of the primitive operators. Second, look for possible subsumption relationships. Third, attempt to create a hybrid f which is a specialization of both f_1 and f_2 . This f is formed by specializing f_1 and f_2 until they are equivalent, or one subsumes the other. This series of transforms is guided by the Δ_1 's and Δ_m 's. We offer here a limited list of such specializations. The reader may feel free to augment it.

PRINCIPLE 5

Each of the following is a valid specialization:

$$\bullet f_{ab,\dots}^* \Rightarrow (f_{a_1,b_1,\dots} f_{a_2,b_2,\dots} \dots f_{a_k,b_k,\dots})^*$$

This restricts the number of times f is applied within an iteration to some multiple of k . Subscripts are presumed distinct.

$$\bullet f_{ab,\dots}^* \Rightarrow f_{a_1,b_1,\dots} f_{a_2,b_2,\dots} \dots f_{a_k,b_k,\dots}$$

This fixes the number of times f is applied within an iteration to exactly k .

$$\bullet f_{ab,\dots}^* \Rightarrow f_{ab,\dots}^+$$

This denotes "at least one iteration is required."

$$\bullet f^* \Rightarrow N$$

This means f is not iterated at all.

$$\bullet (f + g)^* \Rightarrow f^* g^*$$

$$(f + g)^* \Rightarrow g^* f^*$$

These require that the applications of f and g appear in a specific order.

$$\bullet (f + g)^* \Rightarrow (f + fg)^*$$

$$(f + g)^* \Rightarrow (f + gf)^*$$

$$(f + g)^* \Rightarrow (g + fg)^*$$

$$(f + g)^* \Rightarrow (g + gf)^*$$

These insist that some alternatives may not occur alone.

$$\bullet (f + g)^* \Rightarrow f^*$$

$$(f + g)^* \Rightarrow g^*$$

These eliminate an option.

$$\bullet f_{\alpha, \beta, \dots} \Rightarrow f_{a, b, \dots}$$

This represents a consistent substitution of variable a for α , b for β ,...

within the constraints of σ . For example, if σ does not say $x \neq y$,

then A_{xy} may be specialized to A_{xx} or A_{yy} .

The astute reader should have noticed that these "specializations" are merely subsumption tests applied in reverse, i.e., if f_1 subsumes f_2 , then f_2 is a specialization of f_1 . We recognize that as we transform f_1 and f_2 , σ_1 and σ_2 must be modified accordingly to keep track of the restrictions on newly-introduced variables.

Now we try an interesting "real" merger, to create trees (p_1) with an odd number of vertices (p_2):

$$p_1: B_{xy}^*(K_1) \text{ where } x \in V, y \in V$$

$$p_2: (A_{xw} + A_y A_z)^*(K_1)$$

$$\text{where } x, w \in V$$

$$\text{distinct } y, z \in V$$

In keeping with the techniques discussed above, we first rewrite f_1 :

$$f_1: (A_{xy} A_y)$$

The seeds are identical, but no other subsumption relationships are visible. We calculate $\Delta n_1 = 1$, $\Delta m_1 = 1$. For p_2 , however, there are choices: either $\Delta n_2 = 0$ and $\Delta m_2 = 1$, or $\Delta n_2 = 2$ and $\Delta m_2 = 0$. We must specialize both f_1 and f_2 so that merger is possible. The motivation for the particular specializations given is an attempt to match Δn_1 with Δn_2 , and Δm_1 with Δm_2 . First we push f_1 toward $\Delta n = 2$:

$$f_1^* = (A_{xy} A_y)^* \Rightarrow (f_1^2)^* = (A_{xy} A_y A_{wx} A_z)^*$$

$$\sigma_1 = x \in V, y \in V \Rightarrow x, w \in V, \text{ distinct } y, z \in V$$

Note that it is quite legitimate for x and w to be the same, but y and z must be distinct because y is added *after* z and is not in V at the time. Now we push f_2 toward $\Delta n = 2$ and $\Delta m = 2$:

$$f_2^* = (g + hh)^* = (A_{xw} + A_y A_z)^* \Rightarrow (gghh)^* = (A_{xw} A_{pq} A_y A_z)^*$$

$$\sigma_1 = x, w \in V, \text{ distinct } y, z \in V \Rightarrow x, w, p, q \in V, \text{ distinct } y, z \in V$$

We will continue our example in a moment. After specialization we will frequently need to verify that f_1 and f_2 are equivalent or one subsumes the other. Thus we offer some verification rules in:

PRINCIPLE 6

Each of the following pairs of expressions may be verified equivalent

- $f_{a,b,\dots} f_{a,b,\dots} = f_{a,b,\dots}$

Note that the subscripts are identical, hence the lack of impact on the graph.

- $fN = f$

$$Nf = f$$

The null operator may be ignored.

- $f_{a,b,\dots} f_{a,b,\dots}^{-1} = N$

$$f_{a,b,\dots}^{-1} f_{a,b,\dots} = N$$

An operator and its inverse cancel each other out, as long as they are applied to the same vertices/edges.

- $f^* f^* = f^*$

This is a notational equivalent.

- $(f + g)^* = f^*(f + g)^*$

$$(f + g)^* = (f + g)^* g^*$$

These are simplifications.

- $(f + g) = (g + f)$

This is the inherent commutativity in the iteration choice.

- $f = g$ where g is the defined primary equivalent of the composite f , a verification we assumed informally above.

- $A_x f_{a,b,\dots} = f_{a,b,\dots} A_x$ if σ prevents x from being $a, b, \dots$

This is a very limited form of commutativity.

- $f_{\alpha,\beta,\dots}^* f_{a,b,\dots}^* = f_{\alpha,\beta,\dots}^*$ if σ permits $\alpha = a, \beta = b, \dots$

$$f_{a,b,\dots}^* f_{\alpha,\beta,\dots}^* = f_{\alpha,\beta,\dots}^*$$
 if σ permits $\alpha = a, \beta = b, \dots$

These are principles of absorption.

- $f_{a_1,b_1,\dots} f_{a_2,b_2,\dots} = f_{a,b,\dots}$ if σ can be changed to select variables

appropriately.

Now continuing with our example, we can rewrite f_2 (and σ_2) in an attempt to match f_1 . In f_2 we uniformly replace w with y , p with w , and q with z to get

$$(A_{xy} A_{wz} A_y A_z)^* \text{ where } x, y, w, z \in V, \text{ distinct } y, z \in V$$

Because y and z are added during the iteration, $y, z \in V$ is irrelevant and we now have a specialization of f_2 and σ_2 that is

$$(A_{xy} A_{wz} A_y A_z)^* \text{ where } x, w \in V, \text{ distinct } y, z \in V$$

When we contrast this with the specialization of f_1 and σ_1 :

$$(A_{xy} A_y A_{wx} A_z)^* \text{ where } x, w \in V, \text{ distinct } y, z \in V$$

we see that applying the limited commutativity rule to permute A_y and A_{wx} will demonstrate the equivalence of these two algorithms. With their common seed, then, we create the merger, an algorithm (TREE-AND-ODD-N) which generates all trees with an odd number of vertices:

$$p: (A_{xy} A_y A_{wx} A_z)^*(K_1) \text{ where } x, w \in V, \text{ distinct } y, z \in V$$

As our next real example, we offer the merger for complete (p_1) Eulerian (p_2) graphs:

$$p_1: F_x^*(K_1) \text{ where } x \in V, \text{ distinct } v_i \in V, |\{v_i\}| = n$$

$$p_2: (S_{wvz} + Y_{v_1-v_k})^*(K_3)$$

$$\text{where } w, z \in V, wz \in E,$$

$$|\{v_i\} \cap V| \geq 1, \text{ distinct } v_i \in V, v_i v_{i+1}, v_k v_1 \in E, i = 1, 2, \dots, k$$

We rewrite f_1 and f_2 as:

$$f_1: A_{xv_1} \dots A_{xv_n} A_x$$

$$f_2: D_{wz} A_{wv} A_{vz} A_z + A_{v_1 v_2} \dots A_{v_{k-1} v_k} A_{v_k v_1} A_{v_1} \dots A_{v_k}$$

We observe that $\Delta n_1 = 1$, $\Delta m_1 = n$ and either $\Delta n_2 = 1$, $\Delta m_2 = 1$ or $\Delta n_2 < k$, $\Delta m_2 = k$. Using f_2 's second option on K_3 it will not be possible to iterate and restrict Δn_2 to 1, since all the cycle edges must be new to the graph. Thus we specialize f_1 (and σ_1) to f_1^2 :

$$f_1: F_x F_y \text{ distinct } x, y \in V, \text{ distinct } v_i \in V, |\{v_i\}| = n$$

Now $\Delta n_1 = 2$, $\Delta m_1 = 2n + 1$. A specialization of f_2 is

$$Y_{v_1 x y} Y_{v_2 x v_3 y} Y_{v_4 x v_5 y} \dots Y_{v_{n-1} x v_n y}$$

This set of cycle additions is equivalent to the specialized f_1 , and therefore subsumed by it. We need a seed, however. $p_2^{-1}(K_1)$ is FALSE. In this particular case, we select the "first" graph G generated by p_1 for which $p_2^{-1}(G)$ is TRUE. K_2 fails, but K_3 is acceptable. Thus the merger, the algorithm COMPLETE-EULERIAN for complete eulerian graphs, is

$$p: (F_x F_y)^*(K_3) \text{ where distinct } x, y \in V, \text{ distinct } v_i \in V, |\{v_i\}| = n$$

This "discovery" of the seed in this example is more good fortune than technique. An extended discussion of the appropriate seed for a merger appears in Chapter 5.

Our next example is the merger for connected (p_1) bipartite (p_2) graphs:

$$p_1: B_{xy}^*(K_1) \text{ where } x \in V$$

$$p_2: (A_x + A_{xx}A_x + A_{yz})^*(<\{1,2\},\{11\}>)$$

$$\text{where } x \in V$$

$$y, z \in V, |\{yy, zz\} \cap E| = 1$$

f_1 may be rewritten as $A_{xy}A_x$. Noting that $\Delta n_1 = 0$ or 1 (depending on whether y is or is not already in V), $\Delta m_1 = 1$, we have the following alternatives from f_2 :

$$\Delta n_2 = 1 \quad \Delta m_2 = 0$$

$$\Delta n_2 = 1 \quad \Delta m_2 = 0$$

$$\Delta n_2 = 0 \quad \Delta m_2 = 1$$

Note that we choose *not* to count loops in any Δm . The first two alternatives require specialization to match f_1 , so we specialize f_2 (and σ_2) to:

$$f_2: A_{xv}A_x + A_{xw}A_{xx}A_x + A_{yz}$$

$$\text{where } x \in V, v \in V, vw \in E$$

$$x \in V, w \in V, ww \in E$$

$$y, z \in V, |\{yy, zz\} \cap E| = 1$$

Examining f_1 , we see that B_{xy} is equivalent to the first alternative (for $v \in V$), $B_w B_{xv}$ to the second (for $v \in V$), and B_{yz} (for $y, z \in V$) to the third. Thus f_1 is equivalent to the specialized f_2 . The seed for the merger is the minimal bipartite connected graph $<\{1,2\},\{11,12\}>$. (Again, we refer the reader to Chapter 5 for a discussion of seed choice.) The final merger, to generate connected, bipartite graphs

is

$$p: (A_{xv}A_x + A_{xw}A_{xx}A_x + A_{yz})^*(\{1,2\},\{11,12\})^*$$

$$\text{where } x \in V, v \in V, vv \in E$$

$$x \in V, w \in V, ww \in E$$

$$y,z \in V, |\{yy,zz\} \cap E| = 1$$

This example demonstrates that *selective iteration* (such as $A_{xv}A_x$), where the variables are more restricted, can be the key to the creation of f . It also indicates that loop labels may participate in a merger for a single property. If both properties utilized loops, the meaning of the label would likely be obscured and coloring might be more appropriate.

Another application of merger is to test for the existence of a graph with certain characteristics. For example, do there exist odd regular graphs on an odd number of vertices? We consider the merger of odd-regular (p_2) graphs with an odd number of vertices (p_1):

$$p_1: (A_{xy} + A_wA_z)^*(K_1) \text{ where } x,y \in V, \text{ distinct } w,z \in V$$

$$p_2: (OM_{vw_1 \dots v_{k+1} w_1 \dots w_{k-1}} + A_{y_2 q} A_{y_1 p} D_{y_1 y_2} CM_{y_1 \dots y_{k+1}} D_{pq} + FR_{z_1 \dots z_k})^* \\ (Q_{k+1}) \text{ where distinct } v,w \in V, \text{ distinct } v_1, v_2, \dots, v_k \in V, \\ \text{distinct } w_1, w_2, \dots, w_{k-1} \in V, v_{2i-1} v_{2i}, w_{2j-1} w_{2j} \in E, \\ i=1,2,\dots,(k+1)/2; j=1,2,\dots,(k-1)/2$$

$$\text{distinct } p,q \in V, \text{ distinct } y_1, y_2, \dots, y_{k+1} \in V, pq \in E$$

$$\text{distinct } x_1, x_2, \dots, x_k, z_1 \in V, \text{ distinct } z_2, z_3, \dots, z_k \in V, z_i x_i \in E, i=1,2,\dots,k$$

Either $\Delta n_1 = 0$, $\Delta m_1 = 1$ or $\Delta n_1 = 2$, $\Delta m_1 = 0$, and either $\Delta n_2 = 2$, $\Delta m_2 = k$, or $\Delta n_2 = k+1$, $\Delta m_2 = (k+1)k/2$, or $\Delta n_2 = k-1$, $\Delta m_2 = k(k-1)/2$. The seeds indicate that n_1 will always be 1,3,5,... and n_2 will always be $k+1, k+3, \dots$, where k is odd. Clearly no merger is possible since no common seed will ever be found. This fact is well-known in graph theory. Characterizations of n and m may be based on the generating algorithms, producing not hypotheses, but proved theorems about the nature of graphs with multiple properties.

4.7. NP-Completeness and R-Properties

Another, unanticipated strength of our representation is the peculiar formulation NP-complete problems seem to assume. (We adopt the definitions of [Garey 79] and use it as a source for our examples in this section.) A seed set is *simple* if it is finite or definable in an edge-set language other than L_G , otherwise it is *complex*. Most of the R-properties presented thus far have had simple seed sets. When one attempts to write an R-property, and cannot find a formulation with a simple seed, this is not a proof that such a formulation does not exist. It is interesting, however, that those properties which we have not been able to formulate with a simple seed are also known to be those for which a testing algorithm is NP-complete. In this section we discuss some properties with complex seed sets. By an NP-complete property, we mean a property whose testing algorithm is NP-complete.

A cycle which visits every vertex of a graph is called a *Hamiltonian cycle*. A graph with a Hamiltonian cycle is a *Hamiltonian graph*. Several examples of Hamiltonian graphs appear in Figure 4-50, with one Hamiltonian cycle appearing as darkened edges.

The R-property HAMILTONIAN is:

$$A_{xy}^*(C) \text{ where } x, y \in V$$

Figure 4-51 shows the iterative steps in a sample run of HAMILTONIAN using C_5 as a seed. The seed set C is the set of all cycles, i.e.,

$$C = \{C_k \mid k = 1, 2, \dots\}.$$

The floors for Hamiltonian graphs are $\langle P_1, \langle P_2, L_2, \Sigma_1 \rangle, \Sigma_1 \rangle$ and $\langle P_1, \langle P_2, L_{1n}, \Sigma_1 \rangle, \Sigma_1 \rangle$. Note that the language in which the seed is described is itself an R-language.

HAMILTONIAN begins with a cycle (the Hamiltonian cycle) and adds only edges, thereby insuring that the graph remains Hamiltonian. Clearly HAMILTONIAN is correct. The inverse HAMILTONIAN⁻¹ is computed by:

$$f^{-1} = A_{xy}^{-1}$$

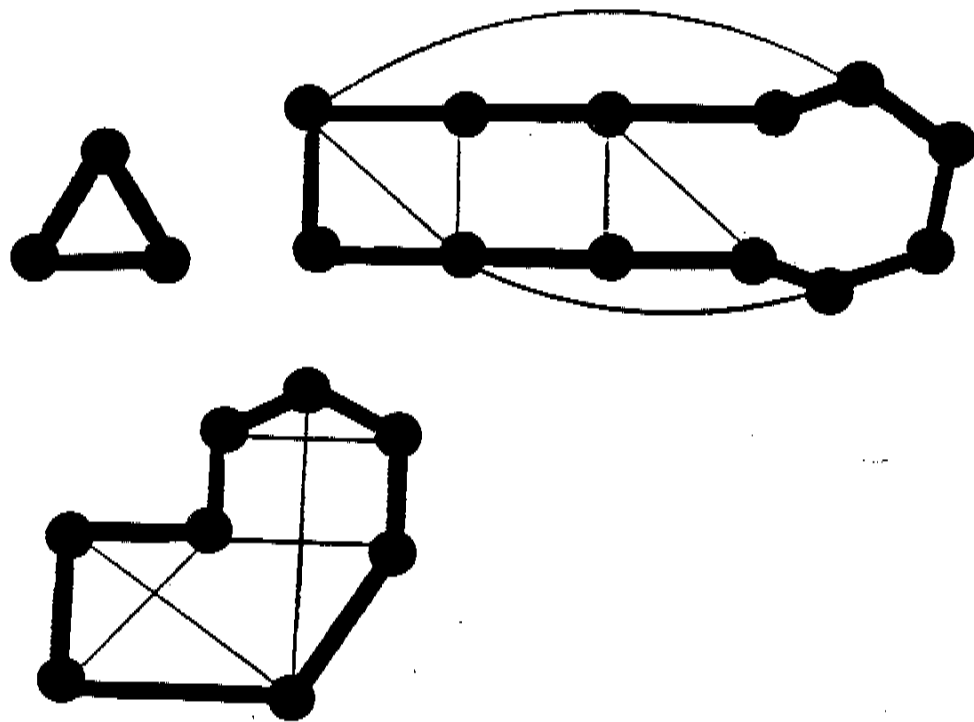


Figure 4-50: Some Hamiltonian Graphs

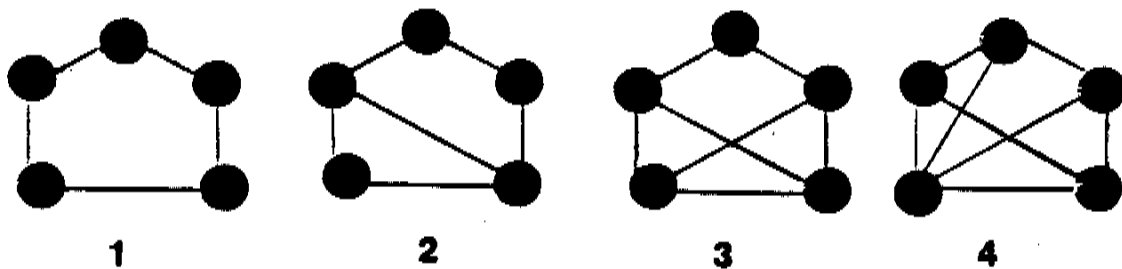
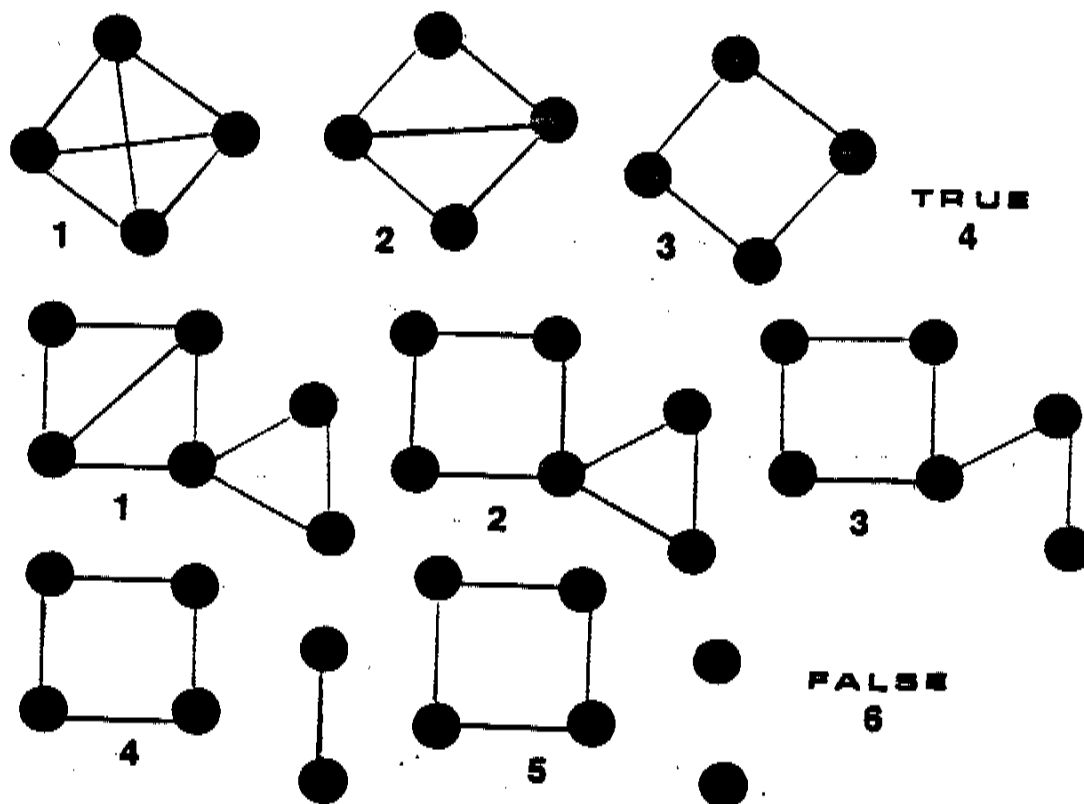


Figure 4-51: HAMILTONIAN in Operation

$$\begin{aligned}
 &= D_{xy} \\
 \sigma_{pre} &= x, y \in V, xy \in E, d(x) \geq 2, d(y) \geq 2 \\
 \sigma^{-1} &= x, y \in V, xy \in E.
 \end{aligned}$$

there exists some largest cycle not including xy

The floors shift to $\langle P_2, \langle P_2, L_2, \Sigma_1 \rangle, \Sigma_6 \rangle$ and $\langle P_2, \langle P_2, L_{1n}, \Sigma_1 \rangle, \Sigma_6 \rangle$. Figure 4-52 shows $HAMILTONIAN^{-1}$ operating on a graph $G \in G_p$ and a graph $G \in G_p$. If xy does not

Figure 4-52: HAMILTONIAN^{-1} in Operation

destroy some largest cycle in G , HAMILTONIAN^{-1} will preserve a Hamiltonian cycle and HAMILTONIAN^{-1} will return TRUE for $G \in G_p$. On $G \notin G_p$, G will reduce to its largest cycle and some set of isolated vertices, and ultimately fail. HAMILTONIAN^{-1} cannot create any new edges, so HAMILTONIAN^{-1} is correct and HAMILTONIAN is complete.

4.7.1. Subgraph Properties and Two-Stage Algorithms

One interesting way to see HAMILTONIAN is as a two-stage algorithm of the form $f^*(g^*(S))$ subject to σ , i.e.

$$A_{xy}^*(S_{wvz}^*(K_3))$$

where $x, y \in V$

$w, z \in V, v \in V, wv, vz \in E$

There are many graph problems which are known to be NP-complete [Garey 79] and can be formulated as "test to see if G has an induced subgraph with property

p." If $G = \langle V, E \rangle$ is a graph and $A \subseteq V$, the graph $G_A = \langle A, \{xy \mid x, y \in A, xy \in E\} \rangle$ is the subgraph of G induced by the vertex set A , and G_A is an *induced subgraph* of G . Our recursive formulation readily produces all such graphs, for if $p = \langle f, S, \sigma \rangle$ then all graph with a subgraph in G_p are generated by:

$$(A_x + A_{yz})^*(f(S))$$

where f is subject to σ

$$y, z \in V$$

The p properties which have been shown to make such a formulation NP-complete include bipartite, acyclic and 3-regular [Garey 79]. Our representation seems to model such NP-completeness by the use of a complex seed set. Once again, our inability to model a property in another way is not a proof, merely a suggestion of some underlying pattern.

Of course, not all properties of the form "G contains an induced subgraph with property p " require such a two-stage formulation, and those which do not will not be NP-complete. For example, if p were edgelessness, every graph $G = \langle V, E \rangle$ has an edgeless subgraph $G = \langle V, \emptyset \rangle$. In addition, the nature of G may be so restricted by the problem formulation that the problem becomes linear. For example, "G contains an independent set of at least k vertices" is a restricted form of edgelessness and is NP-complete, but if G is also bipartite, a formulation with a simple seed set is possible.

Another problem, shown NP-complete, is whether or not a graph $G = \langle V, E \rangle$ has a degree-constrained ($d(x) \leq k$ for all $x \in V$) spanning tree. This too may be viewed as a two-stage algorithm:

$$(A_x + A_{yz})^*(B_{vw}^*(K_1))$$

where $y, z \in V$

$$v \in V, w \in V, d(v) < k$$

Essentially, "having an induced subgraph with property p " has a two-stage formulation $f^*(g^*(S))$ because the kind of operations permitted in g^* may no longer

be permissible after one or more iterations of f . There is the danger of a loss of information. Once f begins, the Σ -language for g becomes inadequate. The only known prevention, within our formulation, is to construct a two-stage procedure.

4.7.2. Graph Properties with Elaborate Seed Sets

Whether or not a graph has a k -vertex cover is an NP-complete problem. The reader may recall that the one-stage algorithm VERTEX-COVER required an underlying skeletal graph, almost as though there were an elaborate seed set being built upon. Looking back, we find such an awkward construction noteworthy, because it is associated with an NP-complete problem.

Our immediate impulse now is to leaf back through Chapters 3 and 4, looking for properties whose L-language is L_Q . In some instances, although the L-language is L_Q the seed set consists of one or two graphs and we are confident from results in graph theory that such a property can be tested in linear time (STAR, K-EDGES, MAX-K, PINWHEEL). Only EVEN-REGULAR and ODD-REGULAR have seed sets in L_Q which are infinite and can be tested for in linear time. Upon reflection we see that, rather than beginning with Q_{k+1} , a set of finitely many disjoint copies of K_{k+1} , we might have written the regular formulations utilizing $CM_{v_1 \dots v_{k+1}}$ to permit the addition of a complete graph on $k+1$ vertices at any time. These "improved" algorithms, and the ease with which they are developed suggest that "linear" properties have simple seed sets and that NP-complete properties do not.

The only exceptions to this neat little package are the labelling properties, those which implicitly or explicitly use labels. Such NP-complete properties (e.g., "has independence number k " or "is k -colorable") have simple seed sets. Thus in the transition to R^c - or R^e -languages we seem to lose the language's ability to predict NP-completeness. This may well be due to the fact that labelling will distinguish among previously-isomorphic graphs.

4.7.3. NP-Completeness and the Recursive Formulation

The cleverness of the automated inversion technique was the pre-profile construction σ_{pre} and the preservation after f of the information σ_{pre} contained. If we cannot construct σ_{pre} adequately or if f destroys that information, then search is required. For example, in HAMILTONIAN, if we begin with a cycle and add some edges to construct G , which edges can we delete in our search to return to the seed? Only those which would not have been in the seed to begin with, i.e., those which some largest cycle does not contain. The embedding languages (both L and Σ) state what data is explicitly represented and representable. If, for example, a graph were characterized in L by describing its cycles and Σ referred to its cycles, HAMILTONIAN would have an implementation which was not NP-complete. This suggests that if one knows the properties of interest in a set of graphs, a language L could be designed to characterize graphs based on only those properties as L -characteristics, speeding the implementation of properties previously regarded as NP-complete.

CHAPTER 5

CONCLUSIONS

*A mathematician, like a painter or a poet, is a maker of patterns.
If his patterns are more permanent than theirs, it is because they are
made with ideas.*

--Hardy

The purpose of this chapter is to draw together the various themes in this work. Rather than a synopsis, this chapter is an evaluation, a critique and a plan for future work. Our work has not consisted of theorems, or even conjectures. We postulated a representational framework and then explored its adequacy from an experimental sample. It is therefore appropriate that this chapter consists of observations, comments and intuitions. We evaluate first the formal language framework, and then the two families of languages. We detail a hypothetical implementation, and conclude with some open questions and implications of this work.

5.1. Languages for Graph Properties

This section evaluates the formal language framework for knowledge representation in graph theory. It provides an overview of the more detailed material in the subsequent section.

This work chose two complementary approaches to the problem of representation in graph theory. The first approach (in Chapter 2) tried to describe an edge set's behavior under simple manipulations on a fixed number of vertices. We explored the edge-set languages L_1 , L_{1n} , L_2 , L_{2n} , L_3 , L_{3n} and L_1^* . The properties available turned out to be

- finite
- hierarchical
- far fewer than the theoretical upper bound
- perfectly capable of inversion
- perfectly capable of merger
- rarely mentioned in graph theory texts

Experimental results suggest that such edge-set languages may provide an adequate hashing technique for graphs up to a certain size. Edge-set languages offer valuable classification schemes for similarities and differences within sets of graphs. Further details appear in 5.2.

The second approach (in Chapters 3 and 4) uses the edge-set languages to represent a given graph property in a recursive formulation. The graph property is an algorithm, which incrementally constructs precisely the set of all graphs which have the property. The R-languages were shown to have substantial expressive and procedural power. The strengths of this representation are its

- clarity and conciseness
- ability to express a wide range of "common" graph properties
- hierarchical transparency (See subsumption in 4.5.) The representation provides efficient testing of such hierarchical statements as "every tree is acyclic" or "every biconnected graph is connected." These are trivially deducible from the R-language representations for those properties.
- amenability to inversion (See 3.5.) An algorithm in this representation can be manipulated to construct a new algorithm which tests an arbitrary graph for a property defined by a graph generator without reference to any other graphs.
- amenability to merger (See 4.6.) The representation can usually be used to construct a new algorithm which computes from two algorithms the set of graphs with both properties.

Further details appear in 5.3.

How do these representations compare with others for mathematics? Lenat's AM could only construct examples and make conjectures based on its observations. It had no facility for proof. We also believe that the poverty of and restrictions on its concept representation (a frame language) substantially hampered its ability to hypothesize. An R-language representation, if automated, could provide the facility to *hypothesize and prove* theorems from their representational structure, i.e., perform mathematical research in graph theory. It would be able to *observe* theorems, i.e., postulate statements which are suggested by the structure of the representation and immediately test their validity. Recall our observations on the format of graph theory theorems in 1.3. We observe that a theorem of the form "if a graph has property p and property q then it has property r" is a statement first of merger and then of subsumption. A theorem of the form "a graph has property p if and only if it has property q" is merely a double subsumption (equivalence) test. A theorem of the form "it is not possible for a graph to have both property p and property q" is a report of merger failure. An implementation which searched out and attempted merger and subsumption relationships would be performing the conjecture and proof research behaviors of a mathematician.

Mathematicians perform other tasks as well. They organize knowledge, as Michener has suggested, and are able to detect significance and relations among concepts. Her frame representation evolves into a set of vague but rigid hierarchical structures, requiring value judgements (is a result basic? key? culminating?) to pigeonhole the knowledge. Our languages have systematized her spaces to achieve procedural power. In return we have had to sacrifice notions of cognitive power and interestingness (such as "key results"). We could also generate arbitrary terminal strings in an R language, merely by following its grammatical property rules. The semantic interpretations of such random strings would be graph properties. Whether or not such properties would be mathematically interesting is open to question. An AM-type guidance system for property development would be necessary.

5.2. Edge-Set Language Results

This section summarizes the results of empirical exploration on the DEC-20 using the edge-set languages L_1 , L_{1n} , L_2 , L_{2n} , L_3 and L_{3n} .

The major result in this area is that the languages L_1 , L_2 and L_3 are, in fact, finite, i.e., that each grammar, whose set of terminal strings is infinite, has only a finite number of interpretations for those strings. The theoretically-calculated number of these interpretations and the number empirically observed under machine computation for the stated n values is summarized in Table 5-1 for L_{1n} , L_{2n} and L_{3n} , both directed and undirected cases.

Undirected Language	Properties	Characterizations
L_{1n} , $n = 1, 2, \dots$	4	12
L_{2n} , $n \leq 25$	27	106
L_{3n} , $n \leq 25$	229	259
Directed Language	Properties	Characterizations
L_{1n} , $n = 1, 2, \dots$	6	24
L_{2n} , $n \leq 25$	202	4849
L_{3n} , $n \leq 13$	2567	>20,000

Table 5-1: Edge-Set Language Properties

Listings of the programs used to achieve these results appear in Appendices II through V.

The edge-set languages have substantial procedural power. They make merger and subsumption, as well as generation and testing, virtually trivial. In addition they have an interesting potential for the kind of graphs which arise [Roberts 76] in many application areas: an ability to find similarities and differences among a set of graphs from their edge-set language characterizations. The languages' ability to categorize graphs into exactly one of finitely many possible classes (for fixed or variable n) suggests that their graph signatures have significant potential as a hashing function.

The operations defined on the edge sets, however, were deliberately limited to control the expressive hierarchy. These limitations also severely restrict the expressive power of the edge-set languages. Even L_3 can be reduced to describing an ordering of the cardinalities of the partitioning sets in a Venn diagram. Graph properties commonly appearing in graph theory texts (with the exception of something like edgelessness or loopfree) are generally not available in the edge-set languages.

5.3. R-Language Results

Having evaluated the edge-set languages, we turn in this section to the following facets of our R-language representations:

- expressive capability
- the $\langle P, L, \Sigma \rangle$ formulation (See 3.3.)
- floors (See 3.4.)
- inversion (See 3.5.)
- subsumption (See 4.5.)
- merger (See 4.6.)
- complexity
- redundancy

5.3.1. Expressive Power

We have no certain way to determine whether or not a given property is within the expressive range of a given R-language. One writes an R-property, as cleverly as possible, and then determines its floor. How do we judge whether the R-language representation as a whole is valid/adequate for all of graph theory? Our work has explored this question empirically. We originally began with $\langle P_1, L_1, \Sigma_1 \rangle$ and several respected texts on graph theory. From the indices of the books we selected many properties. The early choices (in Chapter 3) were simple properties and met with immediate success. The later, more complex choices (in Chapter 4) suggested natural extensions (a register, labels) to R-languages, but were realizable within the basic $\langle P, L, \Sigma \rangle$ formulation. The properties discussed in this document

represent a broad selection from contemporary graph theory.

It would be remarkable to report that all the experimental results (pick a property, express it in an R-language, show correctness and completeness) were positive. (See 3.2.) We did have a limited number of failures, instances where either

- we could not find any $\langle f, S, \sigma \rangle$ description for a property or where
- we could find an $\langle f, S, \sigma \rangle$ description whose correctness was apparent but we could not prove completeness

We suspect that the properties in the first category are merely awaiting a new extension to R-languages, just the way k -factorability needed edge labels. The only property we can cite in the first category is having diameter k . (The *diameter* of a graph is the maximal length of the shortest path between any pair of its vertices.) This property may require edge labels of an elaborate nature. As for the second category, the ingenuity brought to bear in constructing an R-language representation frequently reflects knowledge of theorems in graph theory about equivalent definitions or characterizations. We are hampered both by our own modest knowledge of graph theory and the current development of the subject, particularly with respect to complexity. We are also now aware of the two-stage formulation which NP-complete problems seem to require. (See 4.7.) We attribute our inability to prove completeness to these two factors for the following properties: self-complementary, uniquely k -colorable, k -edge-colorable. Table 5-2 summarizes the 43 properties correctly and completely expressed in this document. Many others, for example "line graph," with well-known characterizations are clearly expressible as well.

Graph theory, however, is not only properties but also relations among them. R-languages have impressive procedural power. We recall our examples of mathematical research behavior at the end of 1.6.3. A system using an R-language for representation will certainly be able to generate examples of any property known to it. As long as the inverse of a property is computable, the system will also be able to test objects for the property. What about proving theorems?

graph	connected graph
edgeless graph	biconnected graph
acyclic graph	k-connected graph
tree	graph on counted vertices
loopfree graph	graph with counted edges
chain	graph with calculated maximum degree
cycle	bipartite graph
star	complete bipartite graph
wheel	k-vertex-covered graph
complete graph	k-independent graph
graph on even number of vertices	k-colored graph
graph on odd number of vertices	k-chromatic graph
graph with even number of edges	graph with vertex covering number k
graph with odd number of edges	graph with circumference k
Eulerian graph	graph with edge covering number k
graph with n vertices	graph with a k-factor
graph with m edges	k-factorable graph
graph of minimum degree k	graph with independence number k
graph of maximum degree k	Hamiltonian graph
pinwheel	planar graph
graph with k components	non-planar graph
even-regular graph	odd-regular graph

Table 5-2: Graph Properties Studied under Recursive Generation

Looking back at 1.3 we recognize that relations among properties are usually verifiable with an R-language representation, and thus most theorems are provable. In particular:

- "If a graph has property p and property q, then it has property r" can be proved by demonstrating that the merger of p and q is subsumed by r.
- "A graph has property p if and only if it has property q" can be proved by demonstrating that p subsumes q and q subsumes p.
- "It is not possible for a graph to have both property p and property

q can be proved by attempting a merger on p and q and demonstrating that the merger is impossible. Inconsistent n and m values are one such proof, and there may be others.

More generally, an R-language representation offers the material for many types of classical mathematical conjectures. The concept of subsumption reflects perfectly the inclusion of one property by another. The merger technique enables us to consider graphs with any finite number of properties. Property equivalence is an expression of alternative characterization. Thus the R-language formulation appears to express not only graph theory properties but also the relations among them. We consider R-languages a potentially powerful representation for all of graph theory. (A detailed treatment of this potential appears in 5.4.)

5.3.2. The $\langle P, L, \Sigma \rangle$ Formulation

In effect, we developed a hierarchy of R-languages. Each language is based on a triple $\langle P, L, \Sigma \rangle$, and the hierarchy for R-languages stands upon the hierarchies for P-languages, L-languages, and Σ -languages diagrammed in Figure 3-6. Thus the R-language $\langle P_1, L_1, \Sigma_2 \rangle$ is less complex than $\langle P_2, L_3, \Sigma_3 \rangle$, but not comparable with $\langle P_1, L_2, \Sigma_1 \rangle$. The P-languages, although limited, appear adequate to provide the expressive power of the benchmark texts. The L-languages also appear adequate, although we would have preferred more edge-set languages and less need for L_Q . This reliance on L_Q may be an intrinsic limitation of the edge-set languages as we define them. The Σ -languages are adequate; although Σ_8 is merely a catchall ("everything you always wanted in an inverse but were afraid to ask for.")

5.3.3. Floors

The floor of a graph property is useful in categorizing the difficulty involved in the calculation of a property. Figure 5-1 summarizes these results for R-languages and R^+ -languages.

We observe that if p_1 subsumes p_2 , the floor for p_1 may be more complex than the floor for p_2 (GENERATE/EDGELESS), less complex (TREE/CHAIN) or the

$\langle P_4, L_0, \Sigma_5 \rangle$ K-EDGES	$\langle P_4, L_2, \Sigma_5 \rangle$ BICONNECTED	$\langle P_4, L_{1n}, \Sigma_5 \rangle$ BICONNECTED K-CONNECTED	$\langle P_2, L_{1n}, \Sigma_5 \rangle$ K-INDEPENDENT EULERIAN
$\langle P_2, L_0, \Sigma_5 \rangle$ PINWHEEL	$\langle P_2, L_0, \Sigma_2 \rangle$ EVEN-REGULAR ODD-REGULAR	$\langle P_4, L_{1n}, \Sigma_3 \rangle$ K-COMPONENTS CONNECTED	$\langle P_2, L_2, \Sigma_5 \rangle$ EULERIAN
$\langle P_2, L_{1n}, \Sigma_4 \rangle$ WHEEL	$\langle P_2, L_2, \Sigma_2 \rangle$ CYCLE	$\langle P_2, L_{1n}, \Sigma_2 \rangle$ CYCLE	$\langle P_3, L_{1n}, \Sigma_2 \rangle$ MAX
$\langle P_2, L_{1n}, \Sigma_1 \rangle$ CHAIN	$\langle P_2, L_3, \Sigma_1 \rangle$ CHAIN	$\langle P_1, L_2, \Sigma_5 \rangle$ BIPARTITE COMPLETE- BIPARTITE	$\langle P_1, L_{1n}, \Sigma_5 \rangle$ BIPARTITE COMPLETE- BIPARTITE K-VERTEX-COVERED
$\langle P_1, L_0, \Sigma_4 \rangle$ STAR	$\langle P_1, L_0, \Sigma_3 \rangle$ MAX-K	$\langle P_1, L_{1n}, \Sigma_3 \rangle$ MIN-K CHAIN ₂	$\langle P_1, L_1, \Sigma_5 \rangle$ COMPLETE
$\langle P_1, L_3, \Sigma_3 \rangle$ ODD-M CHAIN ₂	$\langle P_1, L_3, \Sigma_2 \rangle$ EVEN-N ODD-M	$\langle P_1, L_{1n}, \Sigma_2 \rangle$ EVEN-N ODD-N ODD-M	$\langle P_1, L_1, \Sigma_2 \rangle$ ODD-N LOOPFREE EVEN-M DEGREE

Figure 5-1: Graph Properties with Edge-Set L-Language Grouped by Floors

$\langle P_1, L_{1n}, \Sigma_1 \rangle$	$\langle P_1, L_1, \Sigma_1 \rangle$	$\langle P_1, L_1, \emptyset \rangle$
K-VERTICES	GENERATE	EDGELESS
	ACYCLIC	
	TREE	
	VERTICES	
	EDGES	

Figure 5-1: Graph Properties with Edge-Set L-Language
Grouped by Floors, continued

same (ACYCLIC/TREE). In general, the only way we can distinguish usefully among properties with an L-language is to categorize them as "requiring an edge-set language" or "requiring an R-language". A hierarchy of those R-properties and R^+ -properties using edge-set L-languages appears in Figure 5-2. This hierarchy is based only on P-languages and Σ -languages. The figure does not split properties between classifications (as Figure 5-1 did) and shifts the floor to include the inverse as well. It also makes explicit some of the following points:

- No property required P_5 for generation.
- Σ_5 might be replaced with two Σ -languages to improve the differentiation between Σ_4 and Σ_5 .
- The L-languages describe the minimal case(s) of the property but do little to clarify the hierarchy.

Floor shifting (see 3.6) occurs when the generation language is inadequate for the statement of the inverse. By definition of p^{-1} (in 3.5), the L-language cannot change from p to p^{-1} . If p utilizes P_1 , the P-language must change to P_2 for p^{-1} , since the inverses of the P_1 primitives are in P_2 and not in P_1 . Indeed, every P_1 -based property has a P_2 -based inverse. None of the other P-languages have this problem. The virtue of separating P_1 from P_2 lies in the ability to distinguish purely incremental procedures from those which may decrease the size of the graph. In our opinion this merits the separation and we are willing to have automatic shifts from P_1 -based properties to P_2 -based inverses.

A change in Σ from p to p^{-1} is somewhat more difficult to deal with. With

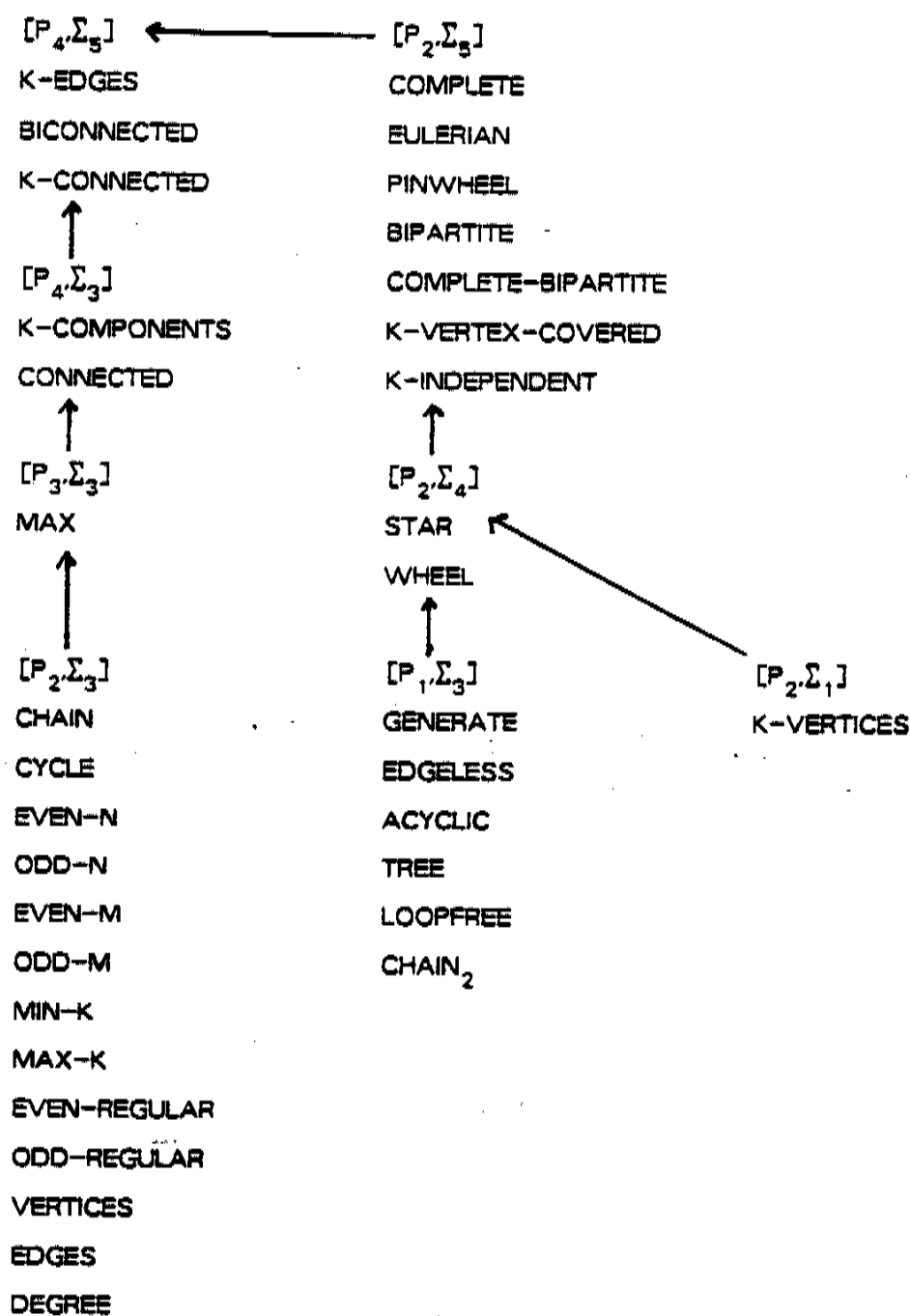


Figure 5-2: Graph Properties with Edge-Set L-Language Ranked by P-Language and Σ -Language

the exception of K-VERTICES (which never deletes a vertex) every inverse requires a Σ -language of at least Σ_3 . With the exception of EULERIAN, no inverse has a simpler Σ -language than its generator. With the exception of CIRCUMFERENCE-K, a Σ -language no more complex than Σ_3 for the generator is adequate for the inverse.

As we have mentioned before, there may be many adequate (correct and complete) formulations for a given property. The fact that, from our work, a property *appears* to have a particular floor is not a proof that no simpler R-language would suffice. For example, the formulation for connectedness which was originally mentioned in 3.7.21, has floor $\langle P_1, L_1, \Sigma_1 \rangle$, whereas we used a formulation with floor $\langle P_4, L_1, \Sigma_5 \rangle$. The first formulation, although correct, has an inverse which must reside in $\langle P_2, L_1, \Sigma_6 \rangle$ and selects edges "which will not disconnect the graph," at best an awkward construction. In much the same fashion, we suspect that CIRCUMFERENCE-K has a "better" formulation.

5.3.4. Inversion, Subsumption and Merger

The automated inversion technique is remarkably successful for generators with Σ -languages no more complex than Σ_4 . An R-language using Σ_5 could probably be automated by skillful programming. The few properties with inverses in Σ_5 , however, are simply not amenable to automation as we have conceived it and should be reformulated if at all possible. We have found that inversion even works correctly on the output of a merger, although there is really no need to calculate an inverse there.¹

Subsumption is an important relation in graph theory. The clarity of its definition for R-languages and the comparative ease with which it may be tested contribute substantially to the strength of our representation.

¹The theory of computability offers support for our inability to make certain absolute statements, such as "an R-property is always invertible if...." We allude to these similarities in footnotes in this chapter, and expect to pursue them at a later time. For inversion, we recall that a set is recursive if and only if both it and its complement are recursively enumerable. Thus the ability to generate a property as a set does not guarantee the ability to test for that property on a given input graph.

The merger technique is surprisingly adequate. We believe that the ability to assert the impossibility of merger (as in ODD-N and ODD-REGULAR) is as important as the ability to generate a merger, because it demonstrates a relation between the unmerged properties. There are probably more merger principles awaiting discovery. The most interesting open question is, "given a merged f and a merged σ , how do we find a common seed set when $S_1 \cap S_2 = \emptyset$?" The cases we tried were "lucky" in that the new seed set quickly appeared within a few iterations, but we have no guarantee that this will always occur. We suspect this to be quite a difficult problem.

5.3.5. Complexity and Redundancy

There has been very little consideration of the complexity of the algorithms which are semantic interpretations of the R-properties. We did note that the complexity of any algorithm is determined both by its internal representation and the matching requirements made by its selector. Thus generation under Σ_1 , Σ_2 , Σ_3 or Σ_4 can certainly be achieved in linear time with properly constructed (not necessarily linear) storage. Because Σ_5 and Σ_6 encompass a much broader range of choices, no such guarantee can be provided for them, and the complexity of algorithms based on them is an open question. The testing algorithms use, in the worst case, storage of $O(n)$ vertices and $O(n^2)$ edges, making the selectors dependent on the size of the input graph. Again, cleverness in storage organization should be able to overcome this for Σ_1 , Σ_2 , Σ_3 and Σ_4 , but probably not for many instances of Σ_5 and Σ_6 .

Redundancy is an interesting issue. The algorithms are non-deterministic; their selectors read "choose any...." Such selection could be randomized. A tester would always return the same output, a generator might not. This non-determinism would not affect the results of a testing algorithm, although its efficiency will be dependent, for certain properties, upon the value of the output and the efficacy of its choices. For example, testing completeness requires deleting one vertex of degree $n-1$ on each of $n-1$ iterations. On a complete graph, selection should be in constant time and TRUE arrived at after $O(n^2)$ edge deletions. On a graph which

would be complete but for a single edge, selection will be in constant time and FALSE arrived at after $O(n^2)$ edge deletions. On a graph without any vertex of degree $n-1$, however, FALSE will be arrived at in $O(n)$ time. Thus incomplete graphs may be faster to test. As another example, consider the graph in Figure 5-3.

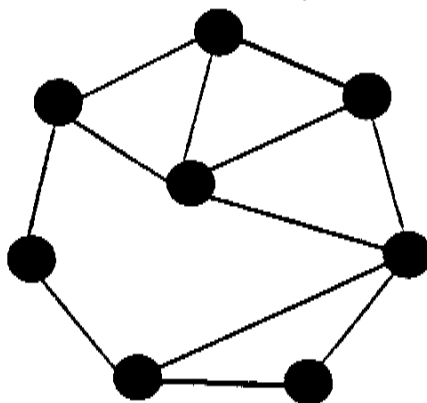


Figure 5-3: A Graph with Variable Testing Time

If we test that graph to see if it is Eulerian, the speed with which we arrive at a result depends upon the cycles we choose to delete. Deleting the largest cycle first will require only two iterations; the smaller cycles can cause greater delay.

When we generate a set of graphs with a specific property, even if we force distinct selections from one execution to the next we do not guarantee distinct (non-isomorphic) graphs². For example, we could generate the same tree on n vertices in many different sequences, growing the tree out from its center, and yet the output would be indistinguishable. Irredundant programs have been developed for, among others, the enumeration of all graphs on n vertices, all trees on n vertices and all spanning trees of a graph. This redundancy would be a problem if generation were our only objective. Fortunately, generation is merely our description of a set of graphs, and we have no intention of executing the same

²This ambiguity is due both to the ambiguity of the formal language and to the range of bindings permitted for the variables during execution of the algorithm.

algorithm repeatedly for distinct results. The same redundancy that may well produce isomorphic graphs also appears related to correct behavior on inversion, a worthwhile tradeoff.

5.3.6. Boolean Properties

Some properties, as we noted in Chapter 1, are boolean. Cyclic, connected and 3-chromatic are all examples of boolean properties. If we have an algorithmic formulation of property p , how will the algorithm for property not- p relate to it? Although the relaxation of a selector condition a would *permit* a graph with the opposite boolean value to appear in the generated set, it will certainly not *guarantee* that precisely the complement of the first graph set will be generated. For example, although

$$B_{xy}^*(K_1) \text{ where } x \in V, y \in V$$

generates all trees, the expression

$$B_{xy}^*(K_1) \text{ where } x \in V$$

does not generate all non-trees, merely all connected graphs (with the possibility of some loops). Let us consider this a bit more.

The reader with some knowledge of graph theory will have noticed an important gap in the properties of Chapters 3 and 4; there is no mention of planarity. A graph is *planar* if it can be drawn on the plane so that no two edges intersect. Several examples of planar and non-planar graphs appear in Figure 5-4.

Kuratowski's theorem provides what appears to be the ideal R-language characterization for planarity: a graph is planar if and only if it has no subgraph homeomorphic to K_5 or $K_{3,3}$. (A graph is homeomorphic to K_5 or $K_{3,3}$ if it can be obtained from one of them by a series of edge subdivisions of the form S_{xvy} .) Figure 5-5 shows the derivation of a non-planar graph from K_5 . Every graph in the figure is, by Kuratowski's theorem, non-planar.

It should, therefore, be quite simple to describe non-planarity in an R-language.

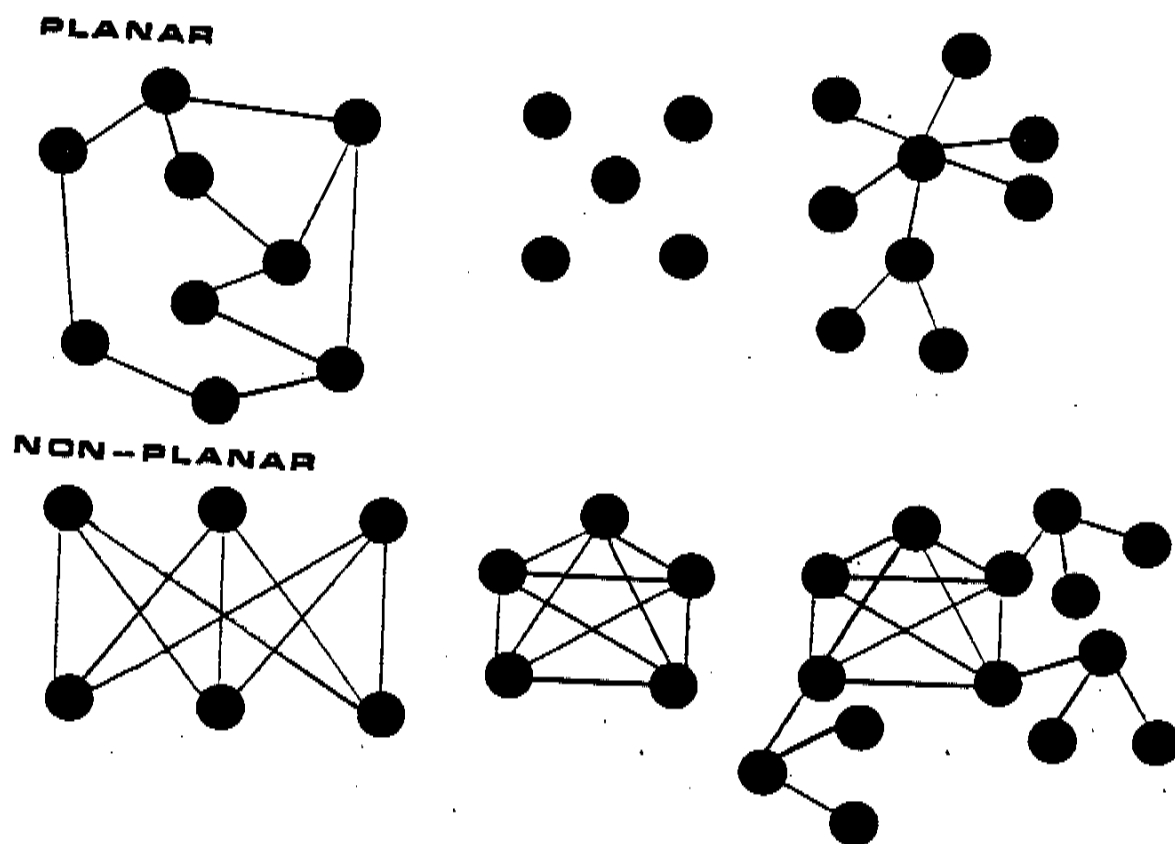
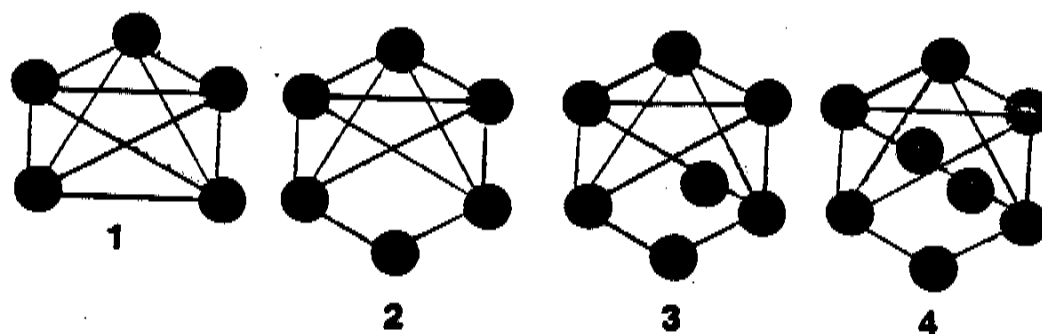


Figure 5-4: Some Graphs and Their Planarity

Figure 5-5: The Construction of a Homeomorph to K_5

The algorithm NON-PLANAR is

$$(A_x + A_{yz} + S_{pq})^*(K_5, K_{3,3})$$

where $y, z \in V$

$p, q \in V, v \in V, pq \in E$

Figure 5-6 shows the iterative steps in a sample run of NON-PLANAR. The floor for non-planar graphs is $\langle P_2, L_0, \Sigma_1 \rangle$.

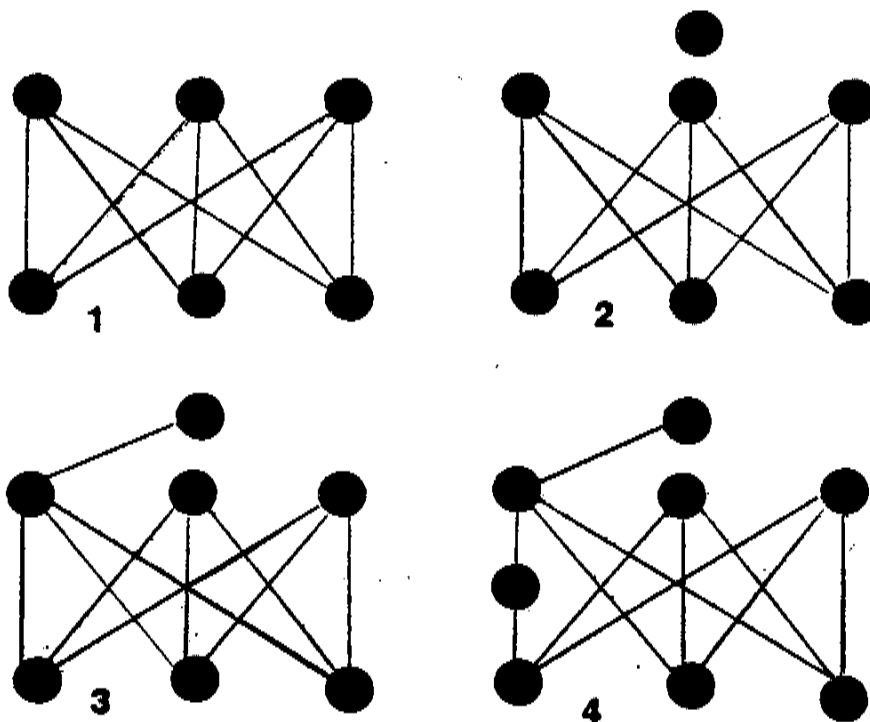


Figure 5-6: A Sample Run of NON-PLANAR

Note that we permit subdivisions to occur interspersed with vertex/edge additions. Although Kuratowski's theorem is suggestive of a two-stage algorithm (first build the homeomorph, then embed it as a subgraph), every subdivision of an edge not in the homeomorph can also be achieved by the addition, in sequence, of a vertex and two edges. Thus the one-stage algorithm is, by Kuratowski's theorem, both correct and complete. Unfortunately, the "automatically" computed inverse is an extremely unpleasant Σ_6 -based formulation:

$$\begin{aligned}
 f^{-1} &= (D_x + D_{yz} + D_v D_{vq} D_{pv} A_{pq}) \\
 \sigma^{-1} &= x \in V, d(x) = 0 \\
 &\quad y, z \in V, yz \in E, yz \text{ is not in every subgraph of } G \\
 &\quad \text{homeomorphic to } K_5 \text{ or } K_{3,3} \\
 &\quad p, v, q \in V, pv, vq \in E, pq \in E, d(v) = 2
 \end{aligned}$$

Of course, Kuratowski spares us any need for NON-PLANAR^{-1} in a completeness proof, but the awkwardness remains. Quite a different alternative is suggested by Tarjan's algorithm for planarity testing. Essentially Tarjan showed that every planar graph could be embedded on the plane with respect to a central chain. A representation which embodies this notion generates all planar graphs via PLANAR. The formulation requires extensive details on Tarjan's algorithm, beyond the scope of this work. Essentially PLANAR constructs the graph from a central (labelled) chain. (If $c(xy) = 1$ the edge is on the chain, else $c(xy) = 0$.) Every vertex has three labels associated with it, which may be concatenated into a single label and deciphered as necessary. The labels indicate upper and lower boundary pointers of the arc on which the vertex lies, and whether the vertex lies to the left of, to the right of, or on the central chain. Thus the R-language requires both edge labels and vertex labels. The generator begins with a chain on two vertices and can extend the chain, with appropriate labelling, at any time. In addition it provides for the construction of arcs and tree-like structures on either side of the chain, properly embedded and labelled. Figure 5-7 shows the iterative steps in a sample run of PLANAR.

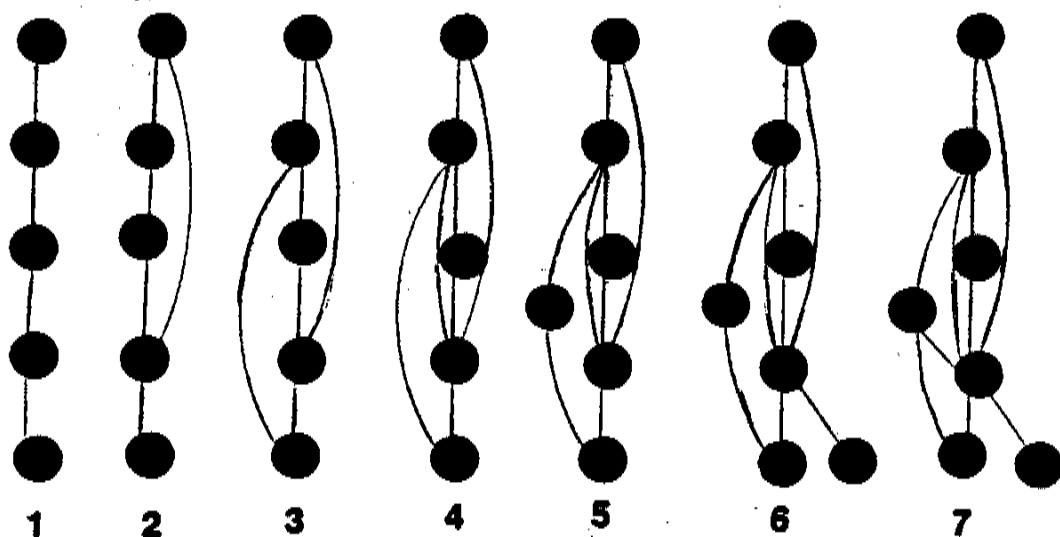


Figure 5-7: A Sample Run of PLANAR

The notation for the algorithm is not given here. Suffice it to say that Tarjan has provided theory to prove such an algorithm is both correct and complete. The

general format is a one-stage algorithm based on A_{xy} 's, S_{xvy} 's and B_{xy} 's, with a tester dependent, as usual, on correct labelling.

We would have liked there to be a clearer relationship between PLANAR and NON-PLANAR. This is not the only instance of this difficulty. The reader may confirm that the following algorithm CYCLIC with floors $\langle P_2, L_1, \Sigma_1 \rangle$ and $\langle P_2, L_2, \Sigma_1 \rangle$ is complete and correct

$$(A_x + A_{yz} + S_{pvq})^*(K_3)$$

where $y, z \in V$

$$p, q \in V, v \in V, pq \in E$$

It too bears a disappointingly unclear relationship to its opposite, the algorithm ACYCLIC with floor $\langle P_1, L_1, \Sigma_1 \rangle$ of 3.7.1:

$$B_{xy}^*(\langle V, \emptyset \rangle) \text{ where } x \in V, y \in E$$

We would, in retrospect, have preferred a more transparent relationship between such pairs of algorithms. In the edge-set languages, the opposite of a boolean property was always equally expressible. In the R-languages, with their procedural orientation, the value of a boolean property may have a different floor dependent on the boolean value, not to mention a different testing efficiency.³ The fact that the merger of two algorithms is impossible does not mean that they are opposite values of the same property. (Witness ODD-REGULAR and ODD-N)

³In the theory of computability, there are properties (i.e., subsets of the integers) which can be generated by a Turing machine, but whose complements cannot. (Such sets are called recursively enumerable non-recursive sets.) In the theory of NP-completeness, membership of a problem in NP does not imply membership of its complement in NP. (Problems with complements in NP are classified as co-NP and the relationship between NP and co-NP is unknown.) The parallels suggest that a theory for boolean properties in R-language contains difficult questions.

5.4. Applications

This section hypothesizes an implementation of our results to show their significance to artificial intelligence. Lenat's AM is used as a framework.

AM [Lenat 76] is intended to model scientific theory formation. AM is a program which makes mathematical discoveries. AM begins only with a hierarchy of 115 set theory concepts and a collection of 242 heuristics.

An AM concept is either an object (e.g., set, list) or an activity (e.g., set-union, first-element). Each concept is represented as a frame, a list of slots. A slot is a (name, value) pair. For concept C and input X, slot names include generalization (i.e., names of concepts more general than C), definitions (ways to test if X is a C), examples (sample X's satisfying C's definition), and worth (a point value assigned to C). The names and number of slots for all concepts are predetermined, uniformly fixed, and limited to a maximum of 25.

An AM heuristic is a rule in the form "if P then Q." P is the list of conditions the heuristic must satisfy to be applicable. Q is the list of actions which will occur if the heuristic is "fired." Heuristics focus AM's attention; they are predetermined and not subject to examination.

The only goal of AM is to fill in slots. AM is intended to perform mathematical research, i.e., increase its knowledge (as represented by its concepts) by acquiring new information and storing it appropriately. "Filling in the slots" is therefore an appropriate, if admittedly limited, translation of "research."

The control structure for the program is a list of tasks, called an agenda. Each task has a priority rating assigned to it. When AM is ready to perform a task, it selects the one with the highest priority and allots it machine resources (time and space) based upon its priority rating. A task ends either with success or by exhausting its resources. Algorithmically AM reads:

- i. Select the top task T on the agenda.
- ii. Assign resources $r(T)$.
- iii. While within $r(T)$, execute T .
- iv. Update the agenda.
- v. Go to i.

Tasks can only:

- add a new task to the agenda
- define a new concept
- add an entry to some slot in some concept

After an hour of CPU time, and without any initial notion of proof, formal reasoning, numbers or arithmetic, AM includes among its discoveries prime numbers and the fundamental theorem of arithmetic (unique factorization of an integer into primes). AM's failures are as interesting as its successes. It never "notices" negative numbers, closure or trichotomy, nor does it ever find any interesting properties of exponentiation. Lenat held AM's heuristics accountable for these lapses. Although the heuristics were initially effective, they lose power as the domain of exploration moves from set theory to number theory. (Lenat is currently working on EURISKO [Lenat 82], an extension to AM. EURISKO attempts to improve AM's research prowess by evolving new heuristics.)

We believe that our work in knowledge representation can make substantial contributions to AM-like activity. We postulate an automated Graph Theorist (GT) as an extension of AM and EURISKO. GT's domain is, of course, graph theory. GT, like AM, is capable of multiple definitions of a property (concept). A GT property definition is a generator in a recursive language, labelled according to its floor. Every definition has a corresponding testing algorithm, also labelled by floor. Examples in GT are readily constructed by executing the definition. (In AM, example generation is much more difficult and less general. Examples are generated by one task and tested by another. AM definitions are what we have described as testing algorithms.) Extremal examples in GT (of great importance to AM heuristics) are elements of the seed set and therefore readily disclosed. Non-examples in GT must

be constructed via generate-and-test, i.e., by running GENERATE and selecting those which fail the property test. Thus a major activity, example construction, is guaranteed correct and complete in GT, although not in AM.

Why is example generation so significant? AM "discovers" primarily by "randomly look[ing] at empirical data for regularities." [Lenat 82] (In a sample run, at least 198 of the 256 tasks were example construction. [Lenat 76]) In set theory, and also on a limited test in plane geometry [Lenat 79], this is a reasonably effective technique, without the breadth GT's generators can offer. With GT's representation, the data available for synthesis improve in quality and accessibility.

This leads us to the nature of a "discovery." An AM conjecture is a slot entry, a relationship observed among examples of concepts. With GT's "better" examples, its discoveries will be correct more often. GT also has an alternative set of heuristics for conjecturing. In addition to examining *examples* for similarities or differences, it can examine *definitions* as well. Because the GT definition is a correct and complete representation of a property in a uniform, highly-structured format, incorrect conjectures are less likely. Even better, many conjectures will be immediately provable using the subsumption techniques outlined in 4.5. Thus GT offers a more fertile representation for conjecture than AM, and a proof facility which AM lacks completely. In GT, a proved statement (theorem) increases the worth of its associated components. Thus temporarily fertile research areas are highlighted with greater efficiency.

AM creates generalizations and specializations of concepts by syntactic tinkering in the LISP concept definitions. GT can use subsumption and merger, thereby preserving the properties of its schema which support completeness and correctness. The GT schema (i.e., the $p = \langle f, S, \sigma \rangle$ formulation) is admittedly more restrictive than a LISP expression. Our empirical observation, however, has indicated that it has substantial expressive power and is more conducive to reasoning (by person or machine) than the typical λ -expression.

We come, finally, to the crucial issue of representation once again. Lenat acknowledges [Lenat 76, Lenat 82] that a representational shift is a powerful heuristic. AM's idea of a representational shift is to create a new concept, thereby enlarging its vocabulary. In reality AM has a single representational language, LISP, within which it discovers concepts. GT, however, permits, even encourages, multiple representation. Each representation is a language, as detailed in this dissertation. The following behaviors are accessible to GT:

- A GT concept can have different definitions in different languages. (Consider, for example, the three definitions of connectedness appearing on pages 129, 138 and 138.)
- GT can be programmed with heuristics appropriate to a specific representation.
- GT can find a common language for a set of concepts, using the partial order of the language hierarchy.
- GT can estimate task difficulty and allocate resources based on the complexity of its chosen representation. Because much of the computational effort will be on matching to bind the selector variables, this estimation should be fairly accurate.
- When a task fails in a given representation, GT can consider shifting to a more complex (and possibly slower) language. GT can be programmed to work in the simplest language possible.
- GT can explore the heuristic "if two properties have the same floor, they may be related."
- Best of all, as the domain of exploration changes we can guide GT to select and focus upon the most productive representations. Thus, if GT is studying cyclic properties, it may select a σ -language which accelerates its algorithms.

As demonstrated through GT, our recursive representational techniques are powerful tools.

5.5. Open Questions

The strengths of the representations have been discussed above. In this section we raise some questions for future work.

- The edge-set languages have been shown both to benefit and to suffer from the severely limited restrictions on their edge set operations. What other operations might "gently" expand their expressive ability, particularly toward properties commonly appearing in graph theory texts?
- Computer exploration of the graph equivalence classes for the edge-set languages is limited by machine space and time. Are there more efficient theoretical approaches which can bound these numbers?
- How might we extend edge-set languages to include graphs with labels? with weights? with minimality and maximality properties?
- The R-languages might be capable of non-redundant generation. What controls would we have to impose and what would they cost us?
- How should edge weights be implemented, either in an edge-set language or an R-language? Do they differ from edge labels in a meaningful way?
- Are R-languages capable of enumeration problems, e.g., finding all the spanning trees of a given graph, or all the distinct k -factors?
- Can R-languages be extended to deal with properties involving minimal/maximal conditions, e.g., the travelling salesman problem or the Chinese postman problem?
- Is there any theoretical proof that no one-stage algorithm based on Σ_1 , Σ_2 , Σ_3 or Σ_4 is NP-complete? Can any other relationships between NP-complete problems and R-properties be derived?
- What insights can the theory of computation give us into the properties of R-languages?

5.6. Implications of This Work

We will continue this artificial intelligence experiment in knowledge representation, and we hope that others will be interested in our approach. A major goal is to extend this kind of structure and organization of graph theory presented here to other areas of mathematical knowledge, such as number theory. In the meantime, we believe that the work already accomplished has implications in many areas.

R-languages are a way to categorize the simplicity or complexity of a graph property. They make explicit (or readily discoverable) many relationships implicit in the vast body of work mathematicians have already produced. An implemented version could provide graph theory with (in order of anticipated difficulty):

- generation of arbitrarily many, arbitrarily large graphs with specified properties, for use in algorithm testing
- theoretical exploration of the equivalence of two characterizations
- explication of implicit hierarchical structure
- suggestions for new, interesting graph properties

In the artificial intelligence community, knowledge representation has been characterized as an ill-defined problem. Consequently, work in knowledge representation has usually concentrated on small, well-defined, but toy, domains. Mathematics as a whole is a very large, well-defined domain. We chose an entire area of mathematics for our work in knowledge representation. Our results suggest that others in artificial intelligence might consider mathematics as a domain. Mathematical theory offers both the certainty and precision of measurement (notably lacking in most real domains) and the challenge of complex relations (notably lacking in most toy domains). In addition, we have suggested here an approach to modelling which combines the factual with the procedural. Our approach in the edge-set languages is important, we believe, because it is a model of controlled exploration with absolute certainty of the resultant impressive procedural power and modest expressive power. Our work in the R-languages is significant, we believe, because

it uses related algorithms to describe properties as well as procedures, resulting in an impressive multifunctional representation.

Finally, the work described here should be significant in several other areas of computer science. The transparency of the relationships among properties (algorithms/procedures/programs) is due to the structure we have imposed upon them. The ease with which certain inversions occur may suggest new approaches in code generation. Perhaps such techniques are more generally applicable in automatic programming. The ability to discern hierarchies may be relevant in data base work. The ability to hypothesize and prove graph theory theorems is certainly relevant to automated deduction. Last but not least, a machine which is told the definition of a property, and can then apply it (by subsumption, by merger, by inversion) must surely be said to learn, to understand, and, perhaps, to think.

APPENDIX A

KEY TO NOTATION

Symbol	Interpretation	Page
A_x	Add vertex x to the graph	66
A_{xy}	Add edge xy to the graph	66
B_{xy}	Branch from vertex x to vertex y	68
C_k	Cycle on k vertices	82
D_x	Delete vertex x from the graph	66
D_{xy}	Delete edge xy from the graph	66
E	Edge set of the graph	12
E_k	Empty graph on k vertices	62
F_x	Fully connect vertex x to the graph	68
$F_{xv_1v_2\cdots v_k}$	Fragment vertex x into vertices x and y	66
$FR_{v_1\cdots v_k}$	Fracture vertex v_1 into K_k maintaining previous adjacencies	131
G	Graph	12
$I_{xv_1\cdots v_k}$	Identify vertices $x, v_1, \dots, v_k$	66
K_k	Complete graph on k vertices	101
L	Loop on all vertices	66
$\bar{L}$	Unloop on all vertices	66
L_i	Language i for graph properties	14
m	Number of edges in the graph	12
M_k	Matching graph	115
n	Number of vertices in the graph	12
N	Null operator	66
O	Order complexity	69
P	Primitive language	64
Q_k	Quantity of disjoint complete graphs	132
S_{xy}	Subdivide edge xy by vertex v	68
S_{xvwy}	Subdivide edge xy by vertices v and w	171
T	Replacement system for testing equivalence of L -expressions	14
T_k	Complete graph on k different-colored vertices	171
U	Set of all finite graphs closed under isomorphism	13
U_k	Edgeless graph on k different-colored vertices	168
V	Vertex set of the graph	12

$W_{h,r}$	Pinwheel on h hubs and r rims	123
$X_{v_1 s_1 \dots v_r s_r}$	Surrogate operator	171
$Y_{u_1 \dots u_k}$	Add cycle $u_1 u_2 \dots u_k u_1$ to the graph	68
$\bar{Y}_{u_1 \dots u_k}$	Delete cycle $u_1 u_2 \dots u_k u_1$ from the graph	68
$Z_{x\alpha}$	Label vertex x with α	167
$Z_{xy\alpha}$	Label edge xy with α	184
σ	Selector, element of Σ	64
Σ	Selection language	64

APPENDIX B

INVESTIGATION OF THE LANGUAGE L_2 FOR UNDIRECTED GRAPHS

B.1. The Program L2

The following is a listing of the program L2.

```

C      PROGRAM NAME:  L2
C      AUTHOR:  SUSAN EPSTEIN
C      THIS PROGRAM CALCULATES SIGNATURES FOR LANGUAGE L2 FOR
C      UNDIRECTED GRAPHS OF UP TO N = 25 VERTICES

C      GRAPHS ARE DESCRIBED AS CASES.  THE CASE PARAMETERS ARE
C      A = THE NUMBER OF EDGES NOT IN THE GRAPH
C      C = THE NUMBER OF EDGES IN THE GRAPH
C      D = THE NUMBER OF LOOPS IN THE GRAPH
C      F = THE NUMBER OF LOOPS NOT IN THE GRAPH
C      I = THE NUMBER OF VERTICES IN THE GRAPH

C      THE SIGNATURE FOR EACH GRAPH IS CALCULATED AS THE VECTOR
C      S AND THEN PACKED, TO SAVE SPACE, INTO THE VECTOR FAKE.
C      A LIST OF ALL PREVIOUSLY ENCOUNTERED SIGNATURES IS STORED
C      IN THE VECTOR G.
C      THE NUMBER OF SIGNATURES AT ANY TIME IS CT.
C      MAT CONTAINS DATA ABOUT THE SIGNATURES.
C      THE MOST RECENT VALUE OF I AT WHICH A NEW SIGNATURE IS
C      FOUND IS LAST.
C      THISC TALLIES WHICH CASES OCCUR FOR FIXED I GREATER THAN 0.
C      ISUM AND IMAX PRESERVE THE TOTAL NUMBER OF CASES AND MOST
C      FREQUENTLY OCCURING CASE.
C      INTEGER N,A,C,D,F,I, LAST
C      INTEGER S (27) ,FAKE,G (107) ,CT,MAT (107) ,THISC (107)
C      INTEGER ISUM (25) ,IMAX (25)
C      DUMMY VARIABLES
C      INTEGER T,K,F2,F3,HUND, IDUM, IDUM2,FF,ZERO

DATA N,A,C,D,F,I,FAKE,CT, LAST/9*0/
DATA HUND,ZERO/100,0/
DATA (S (J) ,J=1,27) /27*0/
DATA (G (J) ,J=1,107) /107*0/
DATA (MAT (J) ,J=1,107) /107*0/

```

```

DATA (THISC(J),J=1,107)/107*0/
DATA (ISUM(J),J=1,25)/25*0/
DATA (IMAX(J),J=1,25)/25*0/
N=25

C      INITIALIZE CASE FOR GRAPH ON NO VERTICES
      CT = 1
      FAKE=0
      DO 1 K=1,27
        FAKE=FAKE*2+1
1      CONTINUE
      G(1)=FAKE

C      GENERATE HEADER AND FIRST DATA LINE
      TYPE 2
2      FORMAT(' VERTICES  CASES      CLASSES    LARGEST    DENSITY')
      TYPE 450,ZERO,CT,CT,CT,HUND

C      MAJOR LOOP ON I = # VERTICES
      DO 500 I=1,N
      DO 5 K = 1,107
        THISC(K) = 0
5      CONTINUE
      T=I*(I-1)/2

C      LOOPS ON A AND D TO CREATE CASES
      DO 400 A=0,T
      DO 300 D=0,I

C      VALUES FOR C AND F CALCULATED FROM A,D AND I
        C=T-A
        F=I-D

C      ZERO OUT SIGNATURE
      DO 10 K=1,27
        S(K)=0
10     CONTINUE

C      SIGNATURE CALCULATION
        IF (A+C.EQ.D+F) S(1)=1
        IF (A.EQ.0) S(2)=1
        IF (C.EQ.0) S(3)=1
        IF (D.EQ.0) S(4)=1
        IF (F.EQ.0) S(5)=1
        IF (A.EQ.C) S(6)=1
        IF (A.EQ.D) S(7)=1
        IF (A.EQ.F) S(8)=1
        IF (C.EQ.D) S(9)=1
        IF (C.EQ.F) S(10)=1
        IF (D.EQ.F) S(11)=1
        IF (A+D.EQ.F) S(12)=1
        IF (A+F.EQ.C) S(13)=1
        IF (A+F.EQ.D) S(14)=1
        IF (A+D.EQ.C) S(15)=1
        IF (C+D.EQ.A) S(16)=1
        IF (C+D.EQ.F) S(17)=1

```

```

      IF (C+F.EQ.A) S(18)=1
      IF (C+F.EQ.D) S(19)=1
      IF (D+F.EQ.A) S(20)=1
      IF (D+F.EQ.C) S(21)=1
      IF (A+C.EQ.F) S(22)=1
      IF (A+D.EQ.C+F) S(23)=1
      IF (A+F.EQ.C+D) S(24)=1
      IF (A.EQ.C+D+F) S(25)=1
      IF (C.EQ.A+D+F) S(26)=1
      IF (A+C.EQ.D) S(27)=1

C      PACKING SIGNATURE S INTO FAKE TO SAVE SPACE, 1 GROUP OF 30
      FAKE=0
      DO 44 L1=1,27
      FAKE=FAKE*2+S(L1)
44      CONTINUE

C      TEST FOR SIGNATURE ALREADY OCCURRING
      DO 50 F2=1,CT
      IF (G(F2).NE.FAKE) GO TO 50
C      MAT(FOO) STORES FIRST VALUE OF I FOR SIGNATURE FOO AS -1.
C      ONCE SIGNATURE RECURS, MAT(FOO) IS NUMBER OF DIFFERENT I
C      VALUES FOR WHICH SIGNATURE OCCURS.
      IF (MAT(F2).GT.0) MAT(F2)=MAT(F2)+1
      F3 = -1
      IF ((MAT(F2).LT.0).AND.(MAT(F2).NE.F3)) MAT(F2)=2
      THISC(F2) = THISC(F2)+1
      GO TO 300
50      CONTINUE

C      INSTALLATION OF NEW SIGNATURE
      CT=CT+1
      G(CT)=FAKE
      MAT(CT)=-1
      THISC(CT)=1
      LAST=1
300      CONTINUE
400      CONTINUE

C      CASE BY CASE OUTPUT ROUTINE.
C      IMAX IS THE LARGEST CLASS SIZE FOR FIXED I.
C      ISUM IS THE NUMBER OF CASES OCCURRING FOR FIXED I.
C      CALCULATING IMAX AND ISUM FROM THISC.
      DO 445 FF=1,CT
      IF (THISC(FF).GT.0) ISUM(1)=ISUM(1)+1
      IF (THISC(FF).GT.IMAX(1)) IMAX(1)=THISC(FF)
445      CONTINUE

C      COMPUTE AND PRINT OUTPUT LINE FOR I
      IDUM=(1+1)*(1+1*(1-1)/2)
      IDUM2=100.0*IMAX(1)/IDUM+.5
      TYPE 450,1,IDUM,ISUM(1),IMAX(1),IDUM2
450      FORMAT(5I10)
500      CONTINUE

```

```

C      SUMMARY STATISTICS
      TYPE 520,CT
520    FORMAT (' NUMBER OF SIGNATURES IS ',15)
      F2=0
C      THE SIGNATURE FOR I=0 IS UNIQUE TO THAT I VALUE.
      DO 525 K=2,CT
      IF (MAT(K).GT.0) F2=F2+1
525    CONTINUE
      TYPE 530,F2
530    FORMAT (' NUMBER OF SIGNATURES FOR MULTIPLE I VALUES IS',15)
      F2=CT-F2
      TYPE 540,F2
540    FORMAT (' NUMBER OF SIGNATURES FOR SINGLE I VALUE IS ',15)
      TYPE 550, LAST
550    FORMAT (' LAST NEW SIGNATURE OCCURS AT I = ',15)
      END

```

B.2. L2 Output

The following is an output listing from program L2.

[PHOTO: Recording initiated Tue 28-Dec-82 10:34AM]

LINK FROM EPSTEIN, TTY 114

TOPS-20 Command processor 5(134712)

End of COMAND.COM.2

2@EXE L2.FOR

LINK: Loading

[LNKXCT L2 execution]

VERTICES	CASES	CLASSES	LARGEST	DENSITY
0	1	1	1	100
1	2	2	1	50
2	6	6	1	17
3	16	12	2	13
4	35	33	2	6
5	66	28	8	12
6	112	42	24	21
7	176	29	48	27
8	261	50	76	29
9	370	34	196	53
10	506	36	272	54
11	672	35	400	60
12	871	58	512	59
13	1106	30	792	72
14	1380	36	960	70
15	1696	43	1268	75
16	2057	50	1460	71
17	2466	30	1984	80
18	2926	40	2276	78
19	3440	35	2808	82
20	4011	50	3136	78
21	4642	34	3964	85

22	5336	36	4400	82
23	6096	35	5236	86
24	6925	58	5732	83
25	7826	30	6912	88

NUMBER OF SIGNATURES IS 106

NUMBER OF SIGNATURES FOR MULTIPLE I VALUES IS 58

NUMBER OF SIGNATURES FOR SINGLE I VALUE IS 48

LAST NEW SIGNATURE OCCURS AT I = 12

CPU time 18.47 Elapsed time 1:22.29

2@POP

[PHOTO: Recording terminated Tue 28-Dec-82 10:36AM]

APPENDIX C

INVESTIGATION OF THE LANGUAGE L_2 FOR DIRECTED GRAPHS

C.1. The Program L2DI

The following is a listing of the program L2DI.

```

C      PROGRAM NAME:  L2DI
C      AUTHOR:  SUSAN EPSTEIN
C      THIS PROGRAM CALCULATES SIGNATURES FOR LANGUAGE L2 FOR
C      DIRECTED GRAPHS OF UP TO N = 25 VERTICES

C      GRAPHS ARE DESCRIBED AS CASES.  THE CASE PARAMETERS ARE
C      A = THE NUMBER OF EDGES NOT IN THE GRAPH
C      B = THE NUMBER OF EDGES IN THE GRAPH WITH REVERSALS NOT IN
C      THE GRAPH
C      C = THE NUMBER OF EDGES IN THE GRAPH WITH REVERSALS IN THE
C      GRAPH
C      D = THE NUMBER OF LOOPS IN THE GRAPH
C      E = THE NUMBER OF EDGES NOT IN THE GRAPH WITH REVERSALS IN
C      THE GRAPH
C      F = THE NUMBER OF LOOPS NOT IN THE GRAPH
C      I = THE NUMBER OF VERTICES IN THE GRAPH

C      THE SIGNATURE FOR EACH GRAPH IS CALCULATED AS THE VECTOR
C      S AND THEN PACKED, TO SAVE SPACE, INTO THE VECTOR FAKE.
C      A LIST OF ALL PREVIOUSLY ENCOUNTERED SIGNATURES IS STORED
C      IN THE MATRIX G.
C      THE NUMBER OF SIGNATURES AT ANY TIME IS CT.
C      MAT CONTAINS DATA ABOUT THE SIGNATURES.
C      THE MOST RECENT VALUE OF I AT WHICH A NEW SIGNATURE IS
C      FOUND IS LAST.
C      THISC TALLIES WHICH CASES OCCUR FOR FIXED I GREATER THAN 0.
C      ISUM AND IMAX PRESERVE THE TOTAL NUMBER OF CASES AND MOST
C      FREQUENTLY OCCURRING CASE.
C      INTEGER N,A,C,D,F,I, LAST
C      INTEGER S(90),FAKE(3),G(5000,3),CT,MAT(5000),THISC(5000)
C      INTEGER ISUM(25),IMAX(25)
C      DUMMY VARIABLES
C      INTEGER T,K,F2,F3,HUND,IDUM,IDUM2,FF,ZERO,D2

```

```

DATA N,A,C,D,F,I,CT, LAST/8*0/
DATA HUND,ZERO/100,0/
DATA (S(J),J=1,90)/90*0/
DATA (FAKE(J),J=1,3)/3*0/
DATA ((G(J,K),J=1,5000),K=1,3)/15000*0/
DATA (MAT(J),J=1,5000)/5000*0/
DATA (THISC(J),J=1,5000)/5000*0/
DATA (ISUM(J),J=1,25)/25*0/
DATA (IMAX(J),J=1,25)/25*0/
N=25

C      INITIALIZE CASE FOR GRAPH ON NO VERTICES
      CT = 1
      F2=0
      DO 1 K=1,30
      F2=F2*2+1
1      CONTINUE
      DO 2 J=1,3
      G(1,J)=F2
2      CONTINUE

C      GENERATE HEADER AND FIRST DATA LINE
      TYPE 3
3      FORMAT(' VERTICES  CASES      CLASSES   LARGEST   DENSITY')
      TYPE 450,ZERO,CT,CT,CT,HUND

C      MAJOR LOOP ON I = # VERTICES
      DO 500 I=1,N
      DO 5 K = 1,5000
          THISC(K) = 0
5      CONTINUE
      T=I*(I-1)/2

C      LOOPS ON A,B AND D TO CREATE CASES
      DO 400 A=0,T
      DO 300 D=0,I
          D2=(T-A)/2
      DO 200 B=0,D2

C      VALUES FOR C AND F CALCULATED
          C=T-A-2*B
          F=I-D

C      ZERO OUT SIGNATURE
      DO 10 K=1,90
          S(K)=0
10     CONTINUE

C      SIGNATURE CALCULATION
          IF (A+C.EQ.D+F) S(1)=1
          IF (A.EQ.0) S(2)=1
          IF (C.EQ.0) S(3)=1
          IF (D.EQ.0) S(4)=1
          IF (F.EQ.0) S(5)=1
          IF (A.EQ.C) S(6)=1
          IF (A.EQ.D) S(7)=1

```

```

IF (A.EQ.F) S(8)=1
IF (C.EQ.D) S(9)=1
IF (C.EQ.F) S(10)=1
IF (D.EQ.F) S(11)=1
IF (A+D.EQ.F) S(12)=1
IF (A+F.EQ.C) S(13)=1
IF (A+F.EQ.D) S(14)=1
IF (A+D.EQ.C) S(15)=1
IF (C+D.EQ.A) S(16)=1
IF (C+D.EQ.F) S(17)=1
IF (C+F.EQ.A) S(18)=1
IF (C+F.EQ.D) S(19)=1
IF (D+F.EQ.A) S(20)=1
IF (D+F.EQ.C) S(21)=1
IF (A+C.EQ.F) S(22)=1
IF (A+D.EQ.C+F) S(23)=1
IF (A+F.EQ.C+D) S(24)=1
IF (A.EQ.C+D+F) S(25)=1
IF (C.EQ.A+D+F) S(26)=1
IF (A+C.EQ.D) S(27)=1
IF (B.EQ.O) S(28)=1
IF (A.EQ.B) S(29)=1
IF (B.EQ.C) S(30)=1
IF (B.EQ.D) S(31)=1
IF (B.EQ.F) S(32)=1
IF (A.EQ.B+C) S(33)=1
IF (A.EQ.B+D) S(34)=1
IF (A.EQ.B+F) S(35)=1
IF (B.EQ.A+C) S(36)=1
IF (B.EQ.A+D) S(37)=1
IF (B.EQ.A+F) S(38)=1
IF (B.EQ.C+D) S(39)=1
IF (B.EQ.C+F) S(40)=1
IF (B.EQ.D+F) S(41)=1
IF (C.EQ.A+B) S(42)=1
IF (C.EQ.B+D) S(43)=1
IF (C.EQ.B+F) S(44)=1
IF (D.EQ.A+B) S(45)=1
IF (D.EQ.B+C) S(46)=1
IF (D.EQ.B+F) S(47)=1
IF (A+B.EQ.C+D) S(48)=1
IF (A+B.EQ.C+F) S(49)=1
IF (A+B.EQ.D+F) S(50)=1
IF (A+C.EQ.B+D) S(51)=1
IF (A+C.EQ.B+F) S(52)=1
IF (A+D.EQ.B+C) S(53)=1
IF (A+D.EQ.B+F) S(54)=1
IF (A+F.EQ.B+C) S(55)=1
IF (A+F.EQ.B+D) S(56)=1
IF (B+C.EQ.D+F) S(57)=1
IF (B+D.EQ.C+F) S(58)=1
IF (B+F.EQ.C+D) S(59)=1
IF (A+B+C.EQ.D) S(60)=1
IF (A+B+C.EQ.F) S(61)=1

```

```

IF (A+B+D.EQ.C) S(62)=1
IF (A+B+D.EQ.F) S(63)=1
IF (A+B+F.EQ.C) S(64)=1
IF (A+B+F.EQ.D) S(65)=1
IF (A+C+D.EQ.B) S(66)=1
IF (A+C+D.EQ.F) S(67)=1
IF (A+C+F.EQ.B) S(68)=1
IF (A+C+F.EQ.D) S(69)=1
IF (A+D+F.EQ.B) S(70)=1
IF (A+D+F.EQ.C) S(71)=1
IF (B+C+D.EQ.A) S(72)=1
IF (B+C+D.EQ.F) S(73)=1
IF (B+C+F.EQ.A) S(74)=1
IF (B+C+F.EQ.D) S(75)=1
IF (B+D+F.EQ.A) S(76)=1
IF (B+D+F.EQ.C) S(77)=1
IF (C+D+F.EQ.A) S(78)=1
IF (C+D+F.EQ.B) S(79)=1
IF (A+B+C.EQ.D+F) S(80)=1
IF (A+B+D.EQ.C+F) S(81)=1
IF (A+B+F.EQ.C+D) S(82)=1
IF (A+C+D.EQ.B+F) S(83)=1
IF (A+C+F.EQ.B+D) S(84)=1
IF (A+D+F.EQ.B+C) S(85)=1
IF (B+C+D.EQ.A+F) S(86)=1
IF (B+C+F.EQ.A+D) S(87)=1
IF (B+D+F.EQ.A+C) S(88)=1
IF (C+D+F.EQ.A+B) S(89)=1
S(90)=0

```

```

C -   PACKING SIGNATURE S INTO FAKE TO SAVE SPACE, 3 GROUPS OF 30
      DO 44 L1=1,3
      FAKE(L1)=S((L1-1)*30+1)
      DO 43 L2=(L1-1)*30+2,L1*30
      FAKE(L1)=FAKE(L1)*2+S(L2)
43    CONTINUE
44    CONTINUE

C     TEST FOR SIGNATURE ALREADY OCCURRING
      DO 50 F2=1,CT
      DO 45 FF=1,3
      IF (G(F2,FF).NE.FAKE(FF)) GO TO 50
45    CONTINUE
C     MAT(FOO) STORES FIRST VALUE OF I FOR SIGNATURE FOO AS -1.
C     ONCE SIGNATURE RECURS, MAT(FOO) IS NUMBER OF DIFFERENT I
C     VALUES FOR WHICH SIGNATURE OCCURS.
      IF (MAT(F2).GT.0) MAT(F2)=MAT(F2)+1
      F3 = -1
      IF ((MAT(F2).LT.0).AND.(MAT(F2).NE.F3)) MAT(F2)=2
      THISC(F2) = THISC(F2)+1
      GO TO 200
      CONTINUE
50    CONTINUE
C     INSTALLATION OF NEW SIGNATURE
      CT=CT+1

```

```

DO 60 FF=1,3
G (CT,FF)=FAKE (FF)
60    CONTINUE
      MAT (CT)=-1
      THISC (CT)=1
      LAST=1
200    CONTINUE
300    CONTINUE
400    CONTINUE

C      CASE BY CASE OUTPUT ROUTINE.
C      IMAX IS THE LARGEST CLASS SIZE FOR FIXED I.
C      ISUM IS THE NUMBER OF CASES OCCURRING FOR FIXED I.
C      CALCULATING IMAX AND ISUM FROM THISC.
      DO 445 FF=1,CT
      IF (THISC (FF).GT.0) ISUM (I)=ISUM (I)+1
      IF (THISC (FF).GT.IMAX (I)) IMAX (I)=THISC (FF)
445    CONTINUE

C      COMPUTE AND PRINT OUTPUT LINE FOR I
      F2=T/2*2
      IF (F2.EQ.T) IDUM=(T**2/4+T+1)*(I+1)
      IF (F2.NE.T) IDUM=((T+1)**2/4+(T+1)/2)*(I+1)
      IDUM2=100.0*IMAX (I)/IDUM+.5
      TYPE 450,I,IDUM,ISUM (I),IMAX (I),IDUM2
450    FORMAT (5I10)
500    CONTINUE

C      SUMMARY STATISTICS
      TYPE 520,CT
520    FORMAT (' NUMBER OF SIGNATURES IS ',15)
      F2=0
C      THE SIGNATURE FOR I=0 IS UNIQUE TO THAT I VALUE.
      DO 525 K=2,CT
      IF (MAT (K).GT.0) F2=F2+1
525    CONTINUE
      TYPE 530,F2
530    FORMAT (' NUMBER OF SIGNATURES FOR MULTIPLE I VALUES IS ',15)
      F2=CT-F2
      TYPE 540,F2
540    FORMAT (' NUMBER OF SIGNATURES FOR SINGLE I VALUE IS ',15)
      TYPE 550, LAST
550    FORMAT (' LAST NEW SIGNATURE OCCURS AT I = ',15)
      END

```

C.2. L2DI Output

The following is the output listing from program L2DI.

28-Dec-82 10:38:20

BATCON Version 104(6133)

GLXLIB Version 1(527)

Job FILE Req #40 for EPSTEIN in Stream 2

OUTPUT: Nolog
UNIQUE: Yes
RESTART: Yes

TIME-LIMIT: 10:00:00
BATCH-LOG: Append
ASSISTANCE: Yes
SEQUENCE: 2101

Input from => PS:<EPSTEIN>FILE.CTL.3
Output to => PS:<EPSTEIN>FILE.LOG

```

10:38:21 USER Rutgers/LCSR DEC-20 (Red), TOPS-20 Monitor 5.2(107200)
10:38:21 USER
10:38:21 USER The system is somewhat unstable. Save your work often!
10:38:21 USER Frequent test times 5:30-6:00 pm and after midnight.
10:38:21 USER
10:38:21 MONTR TIME-LIMIT 36000
10:38:21 MONTR @LOGIN EPSTEIN CS-SRIDHARAN
10:38:26 MONTR [Job 15 also logged into PS:<EPSTEIN>]
10:38:26 MONTR Job 12 on TTY254 28-Dec-82 10:38:25
10:38:26 MONTR Last login on 28-Dec-82 at 09:07:50
10:38:26 MONTR End of COMAND.COMD.2
10:38:26 MONTR 10:38:26 MONTR [PS Mounted]
10:38:26 MONTR
10:38:26 MONTR [CONNECTED TO PS:<EPSTEIN>]
10:38:26 MONTR EXE L2DI.FOR
10:38:30 USER FORTRAN: L2DI
10:38:54 USER MAIN.
10:38:58 USER LINK: Loading
10:39:08 USER [LNKXCT L2DI execution]
10:39:09 USER
10:39:09 USER


| VERTICES | CASES | CLASSES | LARGEST | DENSITY |
|----------|-------|---------|---------|---------|
| 0        | 1     | 1       | 1       | 100     |
| 1        | 2     | 2       | 1       | 50      |
| 2        | 6     | 6       | 1       | 17      |
| 3        | 24    | 20      | 2       | 8       |
| 4        | 80    | 78      | 2       | 3       |
| 5        | 216   | 141     | 8       | 4       |
| 6        | 504   | 336     | 24      | 5       |
| 7        | 1056  | 484     | 48      | 5       |
| 8        | 2025  | 956     | 76      | 4       |
| 9        | 3610  | 911     | 196     | 5       |
| 10       | 6072  | 1065    | 416     | 7       |
| 11       | 9744  | 1045    | 1086    | 11      |
| 12       | 15028 | 1750    | 2496    | 17      |


10:40:31 USER
10:42:09 USER
10:44:27 USER
10:48:01 USER

```

10:53:16	USER	13	22400	998	5746	26
11:00:28	USER	14	32430	1098	9758	30
11:10:16	USER	15	45792	1584	16156	35
11:22:45	USER	16	63257	1785	23508	37
11:43:31	USER	17	85698	968	40284	47
12:26:02	USER	18	114114	1438	55838	49
12:59:39	USER	19	149640	1104	77874	52
14:25:18	USER	20	193536	1651	101792	53
15:38:47	USER	21	247192	1255	150060	61
17:33:05	USER	22	312156	1107	189316	61
18:42:17	USER	23	390144	1081	250364	64
20:17:25	USER	24	483025	2104	305916	63
22:36:03	USER	25	592826	1108	413362	70

22:36:03 USER NUMBER OF SIGNATURES IS 4849
 22:36:03 USER NUMBER OF SIGNATURES FOR MULTIPLE I VALUES IS 2572
 22:36:03 USER NUMBER OF SIGNATURES FOR SINGLE I VALUE IS 2277
 22:36:03 USER LAST NEW SIGNATURE OCCURS AT I = 25
 22:36:04 USER CPU time 6:06:56.65 Elapsed time 11:56:54.81
 22:36:04 MONTR 22:36:05 MONTR Killed by OPERATOR, TTY 246
 22:36:05 MONTR Killed Job 12, User EPSTEIN, Account CS-SRIDHARAN, TTY 254,
 22:36:05 MONTR at 28-Dec-82 22:36:04, Used 6:07:09 in 11:57:39

APPENDIX D

INVESTIGATION OF THE LANGUAGE L_3 FOR UNDIRECTED GRAPHS

D.1. The Program L3

The following is a listing of the program L3.

```

C      PROGRAM NAME:  L3
C      AUTHOR:  SUSAN EPSTEIN
C      THIS PROGRAM CALCULATES SIGNATURES FOR LANGUAGE L3 FOR
C      UNDIRECTED GRAPHS OF UP TO N = 25 VERTICES

C      GRAPHS ARE DESCRIBED AS CASES.  THE CASE PARAMETERS ARE
C      A = THE NUMBER OF EDGES NOT IN THE GRAPH
C      C = THE NUMBER OF EDGES IN THE GRAPH
C      D = THE NUMBER OF LOOPS IN THE GRAPH
C      F = THE NUMBER OF LOOPS NOT IN THE GRAPH
C      I = THE NUMBER OF VERTICES IN THE GRAPH

C      THE SIGNATURE FOR EACH GRAPH IS CALCULATED AS THE VECTOR
C      S AND THEN SORTED, TO SAVE SPACE, USING THE VECTOR S1 INTO
C      THE VECTOR S2.  THE VECTOR Q BECOMES THE SIGNATURE,
C      REPRESENTING THE CARDINALITY OF THE REGIONS AND THEIR
C      ORDERING.  THE SIGNATURE FOR EACH GRAPH IS THEN PACKED, TO
C      SAVE SPACE, INTO THE VECTOR FAKE.
C      A LIST OF ALL PREVIOUSLY ENCOUNTERED SIGNATURES IS STORED
C      IN THE VECTOR G.
C      THE NUMBER OF SIGNATURES AT ANY TIME IS CT.
C      MAT CONTAINS DATA ABOUT THE SIGNATURES.
C      THE MOST RECENT VALUE OF I AT WHICH A NEW SIGNATURE IS
C      FOUND IS LAST.
C      THISC TALLIES WHICH CASES OCCUR FOR FIXED I GREATER THAN 0.
C      ISUM AND IMAX PRESERVE THE TOTAL NUMBER OF CASES AND MOST
C      FREQUENTLY OCCURRING CASE.
C      INTEGER N,A,C,D,F,I, LAST
C      INTEGER S(14),S1(14),S2(14),Q(27),G(300),CT,MAT(300)
C      INTEGER THISC(300),FAKE,ISUM(25),IMAX(25)
C      DUMMY VARIABLES
C      INTEGER T,K,F2,F3,HUND,IDUM,IDUM2,FF,ZERO,TEMP,Y,MI,Z,L1

DATA N,A,C,D,F,I,FAKE,CT, LAST/9*0/

```

```

DATA HUND,ZERO/100,0/
DATA (S(J),J=1,14)/14*0/
DATA (S1(J),J=1,14)/14*0/
DATA (S2(J),J=1,14)/14*0/
DATA (Q(J),J=1,27)/27*0/
DATA (G(J),J=1,300)/300*0/
DATA (MAT(J),J=1,300)/300*0/
DATA (THISC(J),J=1,300)/300*0/
DATA (ISUM(J),J=1,25)/25*0/
DATA (IMAX(J),J=1,25)/25*0/
N=25

C      INITIALIZE CASE FOR GRAPH ON NO VERTICES
      CT = 1
      DO 1 J=1,14
        Q(J)=J
1      CONTINUE
      DO 2 J=15,27
        Q(J)=1
2      CONTINUE
      DO 3 J=1,27
        FAKE=FAKE*2+Q(J)
3      CONTINUE
      G(1)=FAKE

C      GENERATE HEADER AND FIRST DATA LINE
      TYPE 4
4      FORMAT(' VERTICES CASES      CLASSES  LARGEST  DENSITY')
      TYPE 450,ZERO,CT,CT,CT,HUND

C      MAJOR LOOP ON I = # VERTICES
      DO 500 I=1,N
      DO 5 K = 1,300
        THISC(K) = 0
5      CONTINUE
      T=I*(I-1)/2

C      LOOPS ON A AND D TO CREATE CASES
      DO 400 A=0,T
      DO 300 D=0,I
C      VALUES FOR C AND F CALCULATED FROM A,D AND I
        C=T-A
        F=I-D
C      ZERO OUT SIGNATURE
      DO 10 K=1,27
        Q(K)=0
10     CONTINUE

C      S VALUES ARE ASSEMBLED
        S(1)=A
        S(2)=C
        S(3)=D
        S(4)=F
        S(5)=A+C

```

```

S(6) = A+D
S(7) = A+F
S(8) = C+D
S(9) = C+F
S(10) = D+F
S(11) = A+C+D
S(12) = A+C+F
S(13) = A+D+F
S(14) = C+D+F

```

C S VALUES MUST BE SORTED INTO S2 TO CREATE SIGNATURE

```

DO 20 Y=1,14
S1(Y) = S(Y)
S2(Y) = Y
20 CONTINUE
DO 30 Y=1,13
MI=Y
DO 25 Z=Y+1,14
IF (S1(MI) .LE. S1(Z)) GO TO 25
MI=Z
25 CONTINUE
TEMP=S2(Y)
S2(Y) = S2(MI)
S2(MI) = TEMP
TEMP=S1(Y)
S1(Y) = S1(MI)
S1(MI) = TEMP
30 CONTINUE
Q(14) = S2(14)
DO 40 J=1,13
Q(J+14) = 0
Q(J) = S2(J)
IF (S(S2(J)) .EQ. S(S2(J+1))) Q(J+14) = 1
40 CONTINUE

```

C PACKING SIGNATURE Q INTO FAKE TO SAVE SPACE, 1 GROUP OF 30

```

FAKE=0
DO 44 L1=1,27
FAKE=FAKE*2+Q(L1)
44 CONTINUE

```

C TEST FOR SIGNATURE ALREADY OCCURRING

```

DO 50 F2=1,CT
IF (G(F2) .NE. FAKE) GO TO 50
C MAT(FOO) STORES FIRST VALUE OF I FOR SIGNATURE FOO AS -1.
-C ONCE SIGNATURE RECURS, MAT(FOO) IS NUMBER OF DIFFERENT I
C VALUES FOR WHICH SIGNATURE OCCURS.
IF (MAT(F2) .GT. 0) MAT(F2) = MAT(F2) + 1
F3 = -1
IF ((MAT(F2) .LT. 0) .AND. (MAT(F2) .NE. F3)) MAT(F2) = 2
THISC(F2) = THISC(F2) + 1
GO TO 300
50 CONTINUE
C INSTALLATION OF NEW SIGNATURE

```

```

          CT=CT+1
          G(CT)=FAKE
          MAT(CT)=-1
          THISC(CT)=1
          LAST=1
300      CONTINUE
400      CONTINUE

C      CASE BY CASE OUTPUT ROUTINE.
C      IMAX IS THE LARGEST CLASS SIZE FOR FIXED I.
C      ISUM IS THE NUMBER OF CASES OCCURRING FOR FIXED I.
C      CALCULATING IMAX AND ISUM FROM THISC.
          DO 445 FF=1,CT
          IF (THISC(FF).GT.0) ISUM(I)=ISUM(I)+1
          IF (THISC(FF).GT.IMAX(I)) IMAX(I)=THISC(FF)
445      CONTINUE

C      COMPUTE AND PRINT OUTPUT LINE FOR I
          IDUM=(I+1)*(1+I*(I-1)/2)
          IDUM2=100.0*IMAX(I)/IDUM+.5
          TYPE 450,I,IDUM,ISUM(I),IMAX(I),IDUM2
450      FORMAT(5I10)
500      CONTINUE

C      SUMMARY STATISTICS
          TYPE 520,CT
520      FORMAT (' NUMBER OF SIGNATURES IS ',I5)
          F2=0
C      THE SIGNATURE FOR I=0 IS UNIQUE TO THAT I VALUE.
          DO 525 K=2,CT
          IF (MAT(K).GT.0) F2=F2+1
525      CONTINUE
          TYPE 530,F2
530      FORMAT (' NUMBER OF SIGNATURES FOR MULTIPLE I VALUES IS ',I5)
          F2=CT-F2
          TYPE 540,F2
540      FORMAT (' NUMBER OF SIGNATURES FOR SINGLE I VALUE IS ',I5)
          TYPE 550,LAST
550      FORMAT (' LAST NEW SIGNATURE OCCURS AT I = ',I5)
          END

```

D.2. L3 Output

The following is the output listing from program L3.

28-Dec-82 22:51:01

BATCON Version 104(6133)

GLXLIB Version 1(527)

Job FILE3 Req #41 for EPSTEIN in Stream 2

OUTPUT: Nolog
UNIQUE: Yes

TIME-LIMIT: 2:00:00
BATCH-LOG: Append

RESTART: Yes

ASSISTANCE: Yes
SEQUENCE: 2102Input from => PS:<EPSTEIN>FILE3.CTL.1
Output to => PS:<EPSTEIN>FILE3.LOG

```

22:51:01 USER Rutgers/LCSR DEC-20 (Red), TOPS-20 Monitor 5.2(107200)
22:51:01 USER
22:51:01 USER The system is somewhat unstable. Save your work often!
22:51:02 USER Frequent test times 5:30-6:00 pm and after midnight.
22:51:02 USER
22:51:02 MONTR TIME-LIMIT 7200
22:51:02 MONTR @LOGIN EPSTEIN CS-SRIDHARAN
22:51:06 MONTR Job 12 on TTY254 28-Dec-82 22:51:06
22:51:07 MONTR Last login on 28-Dec-82 at 13:07:41
22:51:07 MONTR End of COMAND.COMD.2
22:51:07 MONTR 22:51:07 MONTR [PS Mounted]
22:51:07 MONTR
22:51:07 MONTR [CONNECTED TO PS:<EPSTEIN>]
22:51:07 MONTR EXE L3.FOR
22:51:09 USER FORTRAN: L3
22:51:12 USER MAIN.
22:51:13 USER LINK: Loading
22:51:15 USER [LNKXCT L3 execution]
22:51:16 USER

```

	VERTICES	CASES	CLASSES	LARGEST	DENSITY
22:51:16 USER	0	1	1	1	100
22:51:16 USER	1	2	2	1	50
22:51:16 USER	2	6	6	1	17
22:51:16 USER	3	16	16	1	6
22:51:16 USER	4	35	35	1	3
22:51:16 USER	5	66	52	2	3
22:51:16 USER	6	112	90	4	4
22:51:17 USER	7	176	96	6	3
22:51:17 USER	8	261	129	7	3
22:51:17 USER	9	370	112	16	4
22:51:18 USER	10	506	118	28	6
22:51:19 USER	11	672	120	50	7
22:51:20 USER	12	871	149	70	8
22:51:22 USER	13	1106	108	114	10
22:51:24 USER	14	1380	122	144	10
22:51:27 USER	15	1696	128	203	12
22:51:30 USER	16	2057	145	245	12
22:51:34 USER	17	2466	108	336	14
22:51:39 USER	18	2926	126	392	13
22:51:44 USER	19	3440	120	504	15
22:51:52 USER	20	4011	145	576	14
22:52:01 USER	21	4642	112	730	16
22:52:10 USER	22	5336	122	820	15
22:52:20 USER	23	6096	120	1001	16
22:52:30 USER	24	6925	153	1111	16
22:52:41 USER	25	7826	108	1344	17
22:52:41 USER	NUMBER OF SIGNATURES IS		259		

22:52:41 USER NUMBER OF SIGNATURES FOR MULTIPLE I VALUES IS 157
22:52:41 USER NUMBER OF SIGNATURES FOR SINGLE I VALUE IS 102
22:52:41 USER LAST NEW SIGNATURE OCCURS AT I = 12
22:52:41 USER CPU time 1:03.09 Elapsed time 1:25.58
22:52:41 MONTR 22:52:41 MONTR Killed by OPERATOR. TTY 246
22:52:41 MONTR Killed Job 12, User EPSTEIN, Account CS-SRIDHARAN, TTY 254,
22:52:41 MONTR at 28-Dec-82 22:52:41, Used 0:01:06 in 0:01:35

APPENDIX E

INVESTIGATION OF THE LANGUAGE L_3 FOR DIRECTED

GRAPHS

E.1. The Program L3DI

The following is a listing of the program L3DI.

```

C      PROGRAM NAME:  L3DI
C      AUTHOR:  SUSAN EPSTEIN
C      THIS PROGRAM CALCULATES SIGNATURES FOR LANGUAGE  $L_3$  FOR
C      DIRECTED GRAPHS OF UP TO  $N = 25$  VERTICES

C      GRAPHS ARE DESCRIBED AS CASES.  THE CASE PARAMETERS ARE
C      A = THE NUMBER OF EDGES NOT IN THE GRAPH
C      B = THE NUMBER OF EDGES IN THE GRAPH WITH REVERSALS NOT IN
C          THE GRAPH
C      C = THE NUMBER OF EDGES IN THE GRAPH WITH REVERSALS IN THE
C          GRAPH
C      D = THE NUMBER OF LOOPS IN THE GRAPH
C      E = THE NUMBER OF EDGES NOT IN THE GRAPH WITH REVERSALS IN
C          THE GRAPH
C      F = THE NUMBER OF LOOPS NOT IN THE GRAPH
C      I = THE NUMBER OF VERTICES IN THE GRAPH

C      THE SIGNATURE FOR EACH GRAPH IS CALCULATED AS THE VECTOR
C      S AND THEN SORTED, TO SAVE SPACE, USING THE VECTOR S1 INTO
C      THE VECTOR S2.  THE VECTOR Q BECOMES THE SIGNATURE,
C      REPRESENTING THE CARDINALITY OF THE REGIONS AND THEIR
C      ORDERING.  THE SIGNATURE FOR EACH GRAPH IS THEN PACKED, TO
C      SAVE SPACE, INTO THE VECTOR FAKE.
C      A LIST OF ALL PREVIOUSLY ENCOUNTERED SIGNATURES IS STORED
C      IN THE MATRIX G.
C      THE NUMBER OF SIGNATURES AT ANY TIME IS CT.
C      MAT CONTAINS DATA ABOUT THE SIGNATURES.
C      THE MOST RECENT VALUE OF I AT WHICH A NEW SIGNATURE IS
C      FOUND IS LAST.
C      THISC TALLIES WHICH CASES OCCUR FOR FIXED I GREATER THAN 0.
C      ISUM AND IMAX PRESERVE THE TOTAL NUMBER OF CASES AND MOST
C      FREQUENTLY OCCURRING CASE.
C      INTEGER N,A,C,D,F,I, LAST
C      INTEGER S(30),S1(30),S2(30),Q(60),FAKE(7),G(20000,7),CT

```

```

      INTEGER MAT(20000),THISC(20000)
      INTEGER ISUM(25),IMAX(25)
C     DUMMY VARIABLES
      INTEGER T,K,F2,F3,HUND,IDUM,IDUM2,FF,ZERO,TEMP,Y,M1,Z,L1

      DATA N,A,C,D,F,I,CT,LAST/8*0/
      DATA HUND,ZERO/100,0/
      DATA (S(J),J=1,30)/30*0/
      DATA (S1(J),J=1,30)/30*0/
      DATA (S2(J),J=1,30)/30*0/
      DATA (FAKE(J),J=1,7)/7*0/
      DATA ((G(J,K),J=1,20000),K=1,7)/140000*0/
      DATA (MAT(J),J=1,20000)/20000*0/
      DATA (THISC(J),J=1,20000)/20000*0/
      DATA (ISUM(J),J=1,25)/25*0/
      DATA (IMAX(J),J=1,25)/25*0/
      N=25

C     INITIALIZE CASE FOR GRAPH ON NO VERTICES WHERE Q(K)=K FOR
C     K<31 AND Q(K)=1 FOR K>30
      CT = 1
      DO 2 J=1,6
      FAKE(J)=(J-1)*5+1
          DO 1 K=(J-1)*5+2,J*5
          FAKE(J)=FAKE(J)*100+K
1         CONTINUE
2     CONTINUE
      FAKE2=0
      DO 3 J=31,59
      FAKE2=FAKE2*2+1
3     CONTINUE
      FAKE(7)=FAKE2
      DO 4 J=1,7
      G(1,J)=FAKE(J)
4     CONTINUE

C     GENERATE HEADER AND FIRST DATA LINE
      TYPE 5
5     FORMAT(' VERTICES  CASES      CLASSES    LARGEST    DENSITY')
      TYPE 450,ZERO,CT,CT,CT,HUND

C     MAJOR LOOP ON I = # VERTICES
      DO 500 I=1,N
      DO 6 K = 1,20000
          THISC(K) = 0
6     CONTINUE
      T=I*(I-1)/2

C     LOOPS ON A,B AND D TO CREATE CASES
      DO 400 A=0,T
      DO 300 D=0,I
          D2=(T-A)/2
      DO 200 B=0,D2
C     VALUES FOR C AND F CALCULATED

```

```

      C=T-A-2*B
      F=I-D
C     ZERO OUT SIGNATURE
      DO 10 K=1,60
        Q(K)=0
10    CONTINUE

C     SIGNATURE CALCULATION
      S(1)=A
      S(2)=C
      S(3)=D
      S(4)=F
      S(5)=A+C
      S(6)=A+D
      S(7)=A+F
      S(8)=C+D
      S(9)=C+F
      S(10)=D+F
      S(11)=A+C+D
      S(12)=A+C+F
      S(13)=A+D+F
      S(14)=C+D+F
      S(15)=B
      S(16)=A+B
      S(17)=B+C
      S(18)=B+D
      S(19)=B+F
      S(20)=A+B+C
      S(21)=A+B+D
      S(22)=A+B+F
      S(23)=B+C+D
      S(24)=B+C+F
      S(25)=B+D+F
      S(26)=A+B+C+D
      S(27)=A+B+C+F
      S(28)=A+B+D+F
      S(29)=A+C+D+F
      S(30)=B+C+D+F

C     SORTING S VALUES TO CONSTRUCT SIGNATURE
      DO 20 Y=1,30
        S1(Y)=S(Y)
        S2(Y)=Y
20    CONTINUE
      DO 30 Y=1,29
        M1=Y
        DO 25 Z=Y+1,30
          IF (S1(M1).LE.S1(Z)) GO TO 25
          M1=Z
25    CONTINUE
        TEMP=S2(Y)
        S2(Y)=S2(M1)
        S2(M1)=TEMP
        TEMP=S1(Y)

```

```

          S1(Y)=S1(MI)
          S1(MI)=TEMP
30        CONTINUE
          Q(30)=S2(30)
C        COMPARING Q VALUES BY BITS
          DO 40 J=1,29
          Q(J+30)=0
          Q(J)=S2(J)
          IF (S(S2(J)).EQ.S(S2(J+1))) Q(J+30)=1
40        CONTINUE
C        PACKING SIGNATURE Q INTO FAKE TO SAVE SPACE. 6 GROUPS OF 5
C        AND 1 GROUP OF 29
          DO 44 L1=1,6
          FAKE(L1)=Q((L1-1)*5+1)
          DO 43 L2=(L1-1)*5+2,L1*5
          FAKE(L1)=FAKE(L1)*100+Q(L2)
43        CONTINUE
44        CONTINUE
          FAKE2=0
          DO 45 L1=31,59
          FAKE2=FAKE2*2+Q(L1)
45        CONTINUE
          FAKE(7)=FAKE2

C        TEST FOR SIGNATURE ALREADY OCCURRING
          DO 50 F2=1,CT
          DO 46 FF=1,7
          IF (G(F2,FF).NE.FAKE(FF)) GO TO 50
46        CONTINUE
C        MAT(FOO) STORES FIRST VALUE OF I FOR SIGNATURE FOO AS -1.
C        ONCE SIGNATURE RECURS, MAT(FOO) IS NUMBER OF DIFFERENT I
C        VALUES FOR WHICH SIGNATURE OCCURS.
          IF (MAT(F2).GT.0) MAT(F2)=MAT(F2)+1
          F3 = -1
          IF ((MAT(F2).LT.0).AND.(MAT(F2).NE.F3)) MAT(F2)=2
          THISC(F2) = THISC(F2)+1
          GO TO 200
50        CONTINUE
C        INSTALLATION OF NEW SIGNATURE
          CT=CT+1
          DO 60 FF=1,7
          G(CT,FF)=FAKE(FF)
60        CONTINUE
          MAT(CT)=-1
          THISC(CT)=1
          LAST=1
          IF (CT.EQ.20000) GO TO 519
200       CONTINUE
300       CONTINUE
400       CONTINUE

C        CASE BY CASE OUTPUT ROUTINE.
C        IMAX IS THE LARGEST CLASS SIZE FOR FIXED I.
C        ISUM IS THE NUMBER OF CASES OCCURRING FOR FIXED I.

```

```

C      CALCULATING IMAX AND ISUM FROM THISC.
      DO 445 FF=1,CT
      IF (THISC(FF).GT.0) ISUM(I)=ISUM(I)+1
      IF (THISC(FF).GT.IMAX(I)) IMAX(I)=THISC(FF)
445      CONTINUE

C      COMPUTE AND PRINT OUTPUT LINE FOR I
      F2=T/2*2
      IF (F2.EQ.T) IDUM=(T**2/4+T+1)*(I+1)
      IF (F2.NE.T) IDUM=((T+1)**2/4+(T+1)/2)*(I+1)
      IDUM2=100.0*IMAX(I)/IDUM+.5
      TYPE 450,I,IDUM,ISUM(I),IMAX(I),IDUM2
450      FORMAT(5I10)
500      CONTINUE
      GO TO 521

C      SUMMARY STATISTICS
519      TYPE 520
520      FORMAT(' 20000 SIGNATURES DISCOVERED, MATRIX FULL')
521      TYPE 522,CT
522      FORMAT(' NUMBER OF SIGNATURES IS ',I5)
      F2=0

C      THE SIGNATURE FOR I=0 IS UNIQUE TO THAT I VALUE.
      DO 525 K=2,CT
      IF (MAT(K).GT.0) F2=F2+1
525      CONTINUE
      TYPE 530,F2
530      FORMAT(' NUMBER OF SIGNATURES FOR MULTIPLE I VALUES IS ',I5)
      F2=CT-F2
      TYPE 540,F2
540      FORMAT(' NUMBER OF SIGNATURES FOR SINGLE I VALUE IS ',I5)
      TYPE 550, LAST
550      FORMAT(' LAST NEW SIGNATURE OCCURS AT I = ',I5)
      END

```

E.2. L3DI Output

The following is the output listing from program L3DI.

1-Jan-83 9:06:45

BATCON Version 104(6133)

GLXLIB Version 1(527)

Job FILE3D Req #88 for EPSTEIN in Stream 2

OUTPUT: Nolog
 UNIQUE: Yes
 RESTART: No

TIME-LIMIT: 1:00:00
 BATCH-LOG: Append
 ASSISTANCE: Yes
 SEQUENCE: 3037

Input from => PS:<EPSTEIN>FILE3D.CTL.1
 Output to => PS:<EPSTEIN>FILE3D.LOG

```

9:06:46 USER Rutgers/LCSR DEC-20 (Red), TOPS-20 Monitor 5.2(107200)
9:06:46 USER
9:06:46 USER The system is somewhat unstable. Save your work often!
9:06:46 USER Frequent test times 5:30-6:00 pm and after midnight.
9:06:46 USER
9:06:47 MONTR TIME-LIMIT 3600
9:06:47 MONTR @LOGIN EPSTEIN CS-SRIDHARAN
9:06:50 MONTR [Job 10 also logged into PS:<EPSTEIN>]
9:06:50 MONTR Job 24 on TTY254 1-Jan-83 09:06:50
9:06:50 MONTR Last login on 1-Jan-83 at 08:55:45
9:06:50 MONTR End of COMAND.COMD.2
9:06:50 MONTR 9:06:50 MONTR [PS Mounted]
9:06:50 MONTR
9:06:50 MONTR [CONNECTED TO PS:<EPSTEIN>]
9:06:50 MONTR EXE L3DI.FOR
9:06:52 USER FORTRAN: L3DI
9:07:54 USER MAIN.
9:07:55 USER LINK: Loading
9:08:20 USER [LNKPCX Program too complex to load and execute, will
run from file DSK:024LNK.EXE]
9:08:26 USER [LNKXCT L3DI execution]
9:08:34 USER
9:08:34 USER VERTICES CASES CLASSES LARGEST DENSITY
9:08:34 USER 0 1 1 1 100
9:08:34 USER 1 2 2 1 50
9:08:35 USER 2 6 6 1 17
9:08:35 USER 3 24 24 1 4
9:08:35 USER 4 80 80 1 1
9:08:36 USER 5 216 200 2 1
9:08:39 USER 6 504 476 4 1
9:08:50 USER 7 1056 876 6 1
9:09:27 USER 8 2025 1670 9 0
9:11:15 USER 9 3610 2734 16 0
9:16:06 USER 10 6072 4080 28 0
9:26:33 USER 11 9744 5848 50 1
9:46:33 USER 12 15028 7809 73 0
10:07:45 USER 20000 SIGNATURES DISCOVERED, MATRIX FULL
10:07:45 USER NUMBER OF SIGNATURES IS 20000
10:07:46 USER NUMBER OF SIGNATURES FOR MULTIPLE I VALUES IS 5191
10:07:46 USER NUMBER OF SIGNATURES FOR SINGLE I VALUE IS 14809
10:07:46 USER LAST NEW SIGNATURE OCCURS AT I = 13
10:07:46 USER CPU time 53:36.85 Elapsed time 59:11.72
10:07:46 MONTR 10:07:46 MONTR Killed by OPERATOR, TTY 246
10:07:46 MONTR Killed Job 24, User EPSTEIN, Account CS-SRIDHARAN, TTY 254,
10:07:46 MONTR at 1-Jan-83 10:07:46, Used 0:54:53 in 1:00:56

```

REFERENCES

- [75] *An Introduction to Modelling Using Mixed Integer Programming*
third edition, IBM, Amsterdam, 1975.
- [Amarel 81] Amarel, S.
*'Problems of Representation in Heuristic Problem Solving;
Related Issues in the Development of Expert Systems.*
Technical Report CBM-TR-118, Rutgers University, 1981.
- [Anderson 73] Anderson, J. and Bower, G.
Human Associative Memory.
Winston, Washington, D.C., 1973.
- [Angluin 79] Angluin, D.
Reversible Regular Languages and Inductive Inference.
Yale unpublished.
Used for general reference only. Not cited.
- [Bondy 76] Bondy, J. and Murty, U.
Graph Theory with Applications.
North Holland, New York, 1976.
- [Chang 79] Chang, C.
Resolution Plans in Theorem Proving.
Technical Report RJ2469(32420), IBM Research Laboratory,
February, 1979.
- [Cornell 70] Cornell, D. and Gottlieb, C.
An Efficient Algorithm for Graph Isomorphism.
JACM 17(1):51-64, January, 1970.
Used for general reference only. Not cited.
- [Fahlman 77] Fahlman, S.
A System for Representing and Using Real World Knowledge.
PhD thesis, MIT, December, 1977.
- [Feigenbaum 77] Feigenbaum, E.
The Art of Artificial Intelligence: Themes and Case Studies of
Knowledge Engineering.
In *IJCAI-5*. MIT, Cambridge, Mass., 1977.
- [Garey 79] Garey, Michael R. and Johnson, David S.
Computers and Intractability.
W.H. Freeman and Company, San Francisco, 1979.

- [Hadamard 45] Hadamard, Jacques.
The Psychology of Invention in the Mathematical Field.
 Dover Publications, Inc., New York, 1945.
 Used for general reference only. Not cited.
- [Harary 72] Harary, F.
Graph Theory.
 Addison-Wesley, Reading, Mass., 1972.
- [Hardy 40] Hardy, G.H.
A Mathematician's Apology.
 Cambridge University Press, 1940.
- [Hopcroft 79] Hopcroft, John E. and Ullman, Jeffrey D.
Introduction to Automata Theory, Languages, and Computation.
 Addison-Wesley, Reading, Massachusetts, 1979.
 Used for general reference only. Not cited.
- [Knuth 73] Knuth, D.
The Art of Computer Programming. Volume 1: Fundamental Algorithms.
 Addison-Wesley, Reading, Mass., 1973.
 Used for general reference only. Not cited.
- [Lawler 76] Lawler, E.
Combinatorial Optimization: Networks and Matroids.
 Holt, Rinehart and Winston, New York, 1976.
 Used for general reference only. Not cited.
- [Lenat 76] Lenat, Douglas B.
AM: An Artificial Intelligence Approach to Discovery in Mathematics as Heuristic Search.
 PhD thesis, Stanford, July, 1976.
- [Lenat 77] Lenat, D.
 The Ubiquity of Discovery.
 In *IJCAI-5*. IJCAI-5, MIT, Cambridge, Mass., 1977.
 1977 Computers and Thought Lecture, Used for general reference only. Not cited.
- [Lenat 79] Lenat, D. B.
Machine Intelligence 9.
 Ellis Horwood Limited, Chichester, England, 1979, pages
 251-283 chapter On Automated Scientific Theory Formation:
 A Case Study Using the AM Program.
- [Lenat 82] Lenat, D.
 The Nature of Heuristics.
Artificial Intelligence 19(2):189-249, October, 1982.
- [Michener 78] Michener, E.
Understanding Understanding Mathematics.
 Technical Report AI MEMO-488, MIT, August, 1978.
 LOGO MEMO-50.

- [Minsky 63] Minsky, M.
Computers and Thought.
McGraw Hill, New York, 1963, pages 406-450chapter Steps
toward Artificial Intelligence.
Originally in the Proceedings of the IRE, volume 49, January,
1961.
- [Mitchell 83] Mitchell, T., Utgoff, P. and Banerji, R.
Learning by Experimentation: Acquiring and Modifying
Problem-Solving Heuristics.
In Michalski, R., Carbonell, J. and Mitchell, T. (editors), *Machine
Learning*. Tioga Press, 1983.
- [Moses 75] Moses, J.
A MACSYMA Primer.
Technical Report Mathlab Memo No. 2, MIT Computer Science
Lab, October, 1975.
- [Newell 75] Newell, A. and Simon, H.
Computer Science as Empirical Inquiry: Symbols and Search.
CACM 19(3), March, 1975.
1975 ACM Turing Lecture.
- [Newell 76] Newell, A.
CMU Proposal to ARPA,
unpublished.
- [Nilsson 80] Nilsson, N.
Principles of Artificial Intelligence.
Tioga, 1980.
- [Ore 62] Ore, O.
American Mathematical Society Colloquium Publications.
Volume 38: *Theory of Graphs*.
American Mathematical Society, Providence, Rhode Island, 1962.
- [Pascal 64] Pascal, Rene.
Pensees de Pascal.
Editions Garnier Freres, Paris, France, 1964, pages 73-84.
- [Poincare 52] Poincare, Henri.
Science and Hypothesis.
Dover Publications Inc., New York, 1952.
- [Poincare 70] Poincare, Henri.
La Valeur de la Science.
Flammarion, France, 1970, pages 27-40chapter L'Intuition et la
Logique en Mathematiques.
- [Quillian 67] Quillian, J.
Word Concepts: A Theory and Simulation of Some Basic
Semantic Capabilities.
Behavioral Science 12, 1967.
- [Roberts 76] Roberts, F.
Discrete Mathematical Models.
Prentice-Hall, Englewood Cliffs, NJ., 1976.

- [Schank 75] Schank, R.
The Structure of Episodes in Memory.
In Bobrow, D. and Collins, A. (editor), *Representation and Understanding*, pages 237-272. Academic Press, New York, 1975.
- [Slagle 63] Slagle, J.
A Heuristic Program That Solves Symbolic Integration Problems in Freshman Calculus.
In Feigenbaum, E. and Feldman, J. (editors), *Computers and Thought*, pages 191-203. McGraw Hill, New York, 1963.
- [Sowa 79] Sowa, J.
Semantics of Conceptual Graphs.
Technical Report, IBM SYstems Research Institute, 1979.
- [Sridharan 80] Sridharan, N.
Representational Facilities of AIMDS: A Sampling.
Technical Report CBM-TR-119, Rutgers University, May, 1980.
- [Sridharan 81] Sridharan, N., Schmidt, C. and Goodson, J.
Reasoning by Default.
Technical Report CBM-TR-119, Rutgers University, April, 1981.
- [Statman 82] Statman, Richard.
Topological Subgraphs of Cubic Graphs and a Theorem of Dirac.
Journal of Graph Theory 6:419-427, 1982.
Used for general reference only. Not cited.
- [Winston 75] Winston, P.
Learning Structural Descriptions from Examples.
In Winston, P. (editors), *The Psychology of Computer Vision*, McGraw Hill, New York, 1975.
- [Winston 80] Winston, P.
Learning and Reasoning by Analogy.
CACM 23(12):689-703, December, 1980.

INDEX

- Σ_c -language (thesis specific) 167
- Acyclic 82
- ACYCLIC (algorithm) 82
- Adjacent 12
- Biconnected 139
- BICONNECTED (algorithm) 139
- Bipartite 96
- BIPARTITE (algorithm) 156
- Block 176
- Boolean property (thesis specific) 13
- Cardinality 12
- Case (thesis specific) 38
- Center of a star (thesis specific) 97
- Chain 90
- CHAIN (algorithm) 90
- Characteristic (thesis specific) 13
- Circumference 185
- CIRCUMFERENCE-K (algorithm) 185
- Closed walk 82
- Collectively exhaustive description set (thesis specific) 13
- Coloring 166
- Complement 23
- Complete 101
- COMPLETE (algorithm) 101
- Complete (thesis specific) 65
- Complete bipartite 96, 158
- COMPLETE-BIPARTITE (algorithm) 158
- Complex seed set 213
- Component 126
- Connected 85, 126, 129, 138
- CONNECTED (algorithm) 138
- Connected component 126
- CONSTRUCT (algorithm) 60
- Contractible 175
- Correct (thesis specific) 65
- Covering an edge 161
- Cutpoint 176
- Cycle 82
- CYCLE (algorithm) 94

Degree 70
 DEGREE (algorithm) 151
 Description (thesis specific) 13
 Diameter 224
 Directed 12

 Edge 12
 Edge cover 189
 Edge covering number 189
 EDGELESS (algorithm) 62
 EDGES (algorithm) 148
 Elementary edge contraction 175
 Empty graph 12
 Enumerate (thesis specific) 61
 Equal properties (thesis specific) 13
 Equal set cardinality 35
 Equivalent L-expressions (thesis specific) 14
 Equivalent R-properties (thesis specific) 201
 EULERIAN (algorithm) 111
 Eulerian graph 111
 Eulerian walk 111
 EVEN-M (algorithm) 106
 EVEN-N (algorithm) 102
 EVEN-REGULAR (algorithm) 131
 Expressive power (thesis specific) 8

 Floor (thesis specific) 72
 FRONT-END 32

 General description (thesis specific) 13
 GENERATE (algorithm) 61
 Generator algorithm (thesis specific) 74
 Graph 12
 Graph generator 32
 Graph property (thesis specific) 13
 Graph theory (thesis specific) 7

 HAMILTONIAN (algorithm) 213
 Hamiltonian cycle 213
 Hamiltonian graph 213
 Hub of a pinwheel (thesis specific) 123
 Hub of a wheel (thesis specific) 98

 Independence number 182
 INDEPENDENCE-K (algorithm) 182
 Independent vertex set 164
 Inverse of a property (thesis specific) 74
 Isomorphic 12
 Isomorphism 12

 K-chromatic 170
 K-CHROMATIC (algorithm) 171
 K-colorable 166

K-COLORED (algorithm) 168
 K-colored (thesis specific) 166
 K-coloring 166
 K-COMPONENTS (algorithm) 126
 K-connected 142
 K-CONNECTED (algorithm) 142
 K-EDGE-COVER (algorithm) 190
 K-EDGES (algorithm) 115
 K-factor 193
 K-FACTOR (algorithm) 193
 K-factorable 196
 K-FACTORABLE (algorithm) 197
 K-INDEPENDENT (algorithm) 164
 K-regular 129
 K-vertex-coverable 161
 K-VERTEX-COVERED (algorithm) 161
 K-vertex-covered (thesis specific) 161
 K-VERTICES (algorithm) 114

 L-characterization (thesis specific) 14, 15
 L-class 15
 L-expression (thesis specific) 14
 L-property (thesis specific) 15
 L1-GENERATOR 33
 L1-TESTER 34
 L_c -languages 167
 Label (thesis specific) 166
 Labelled graph 166
 Labelling (thesis specific) 165
 Loop 12
 Loop labelling (thesis specific) 157
 Loop marking (thesis specific) 155
 Loopfree 88
 LOOPFREE (algorithm) 88

 MAX (algorithm) 153
 MAX-K (algorithm) 120
 Maximum degree 120
 Merger of graph properties (thesis specific) 204
 MIN-K (algorithm) 119
 Minimum degree 119
 Mutually exclusive description set (thesis specific) 13

 Neighbor 12
 Neighborhood 70
 Node 12
 NON-PLANAR (algorithm) 234
 Numeric (thesis specific) 13

 ODD-M (algorithm) 108
 ODD-N (algorithm) 105
 ODD-REGULAR (algorithm) 135
 Open walk 82

P_c -language (thesis specific) 167
 Partitioning description set (thesis specific) 13
 Path 21, 111
 PINWHEEL (algorithm) 124
 Pinwheel (thesis specific) 123
 Planar 233
 PLANAR (algorithm) 236
 Post-profile (thesis specific) 78
 Pre-profile (thesis specific) 78
 Procedural power (thesis specific) 8
 Profile (thesis specific) 78

 R-property (thesis specific) 64
 R^+ -property (thesis specific) 147
 R^c -property (thesis specific) 167
 R^* -property (thesis specific) 184
 Recursive graph grammar (thesis specific) 64
 Region 26
 Regular 129
 Reversal 23
 Reverse 23
 Rim of a pinwheel (thesis specific) 123
 Rim of a wheel (thesis specific) 98

 Satisfied description (thesis specific) 13
 Seed graph (thesis specific) 64
 Seed set (thesis specific) 64
 Selector (thesis specific) 64
 Set equality 23
 Set inequality 23
 Signature 15
 Signature (thesis specific) 14
 Simple seed set (thesis specific) 213
 Spoke (thesis specific) 97
 Star 96
 STAR (algorithm) 96
 Subgraph 126, 176
 Subsumption 201

 Testing algorithm 34, 74
 Trail 111
 Transitive closure (thesis specific) 51
 Tree 85
 TREE (algorithm) 85

 Undirected graph 12
 Unequal set cardinality 35
 Union of graphs (thesis specific) 196
 Unique description (thesis specific) 13
 Unsatisfiable description (thesis specific) 13

Vertex 12
Vertex cover 161
Vertex covering number 176
VERTEX-COVER (algorithm) 177
VERTICES (algorithm) 148

Walk 82
Weakly-complete (thesis specific) 30
Wheel 98
WHEEL (algorithm) 98

Susan Lynn Epstein

1944	Born April 8, 1944 in New York, New York
1961	Graduated New Rochelle High School, New Rochelle, New York, summa cum laude
1961-64	Attended Smith College, Northampton, Massachusetts
1963	Phi Beta Kappa
1964	Sigma Xi
1964	B.A. in Mathematics, magna cum laude, Smith College
1964-65	Faculty, Amity Regional Senior High School, Woodbridge, Connecticut
1965-67	Faculty, The Lenox School, New York, New York
1965-68	Attended Mathematics Department of Courant Institute, New York University, New York, New York
1967-68	Operations Research Analyst, Chase Manhattan Bank, New York, New York
1968	M.S. in Mathematics, New York University
1969-70	Senior Consultant, Advanced Computer Techniques, New York, New York
1971-75	Independent computer consultant, New York/New Jersey area
1974-79	Instructor, Montclair State College, Montclair, New Jersey
1978-1983	Attended Department of Computer Science, Rutgers University, New Brunswick, New Jersey
1981	Contributor, <i>The Handbook of Artificial Intelligence</i> , edited by Avron Barr and Edward A. Feigenbaum, William Kaufman, Inc., Los Altos, California
1981-83	Graduate Fellowship, Rutgers University
1983	Article entitled "Challenges" published in SIGART newsletter, number 83, January, 1983
1983	Ph.D. in Computer Science, Rutgers University