# KNOWLEDGE REPRESENTATION IN MATHEMATICS A CASE STUDY IN GRAPH THEORY

Susan Lynn Epstein

DCS-TR-134

Department of Computer Science
Rutgers University
New Brunswick, New Jersey 08903

# 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

# LIST OF FIGURES

# LIST OF TABLES

# 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?

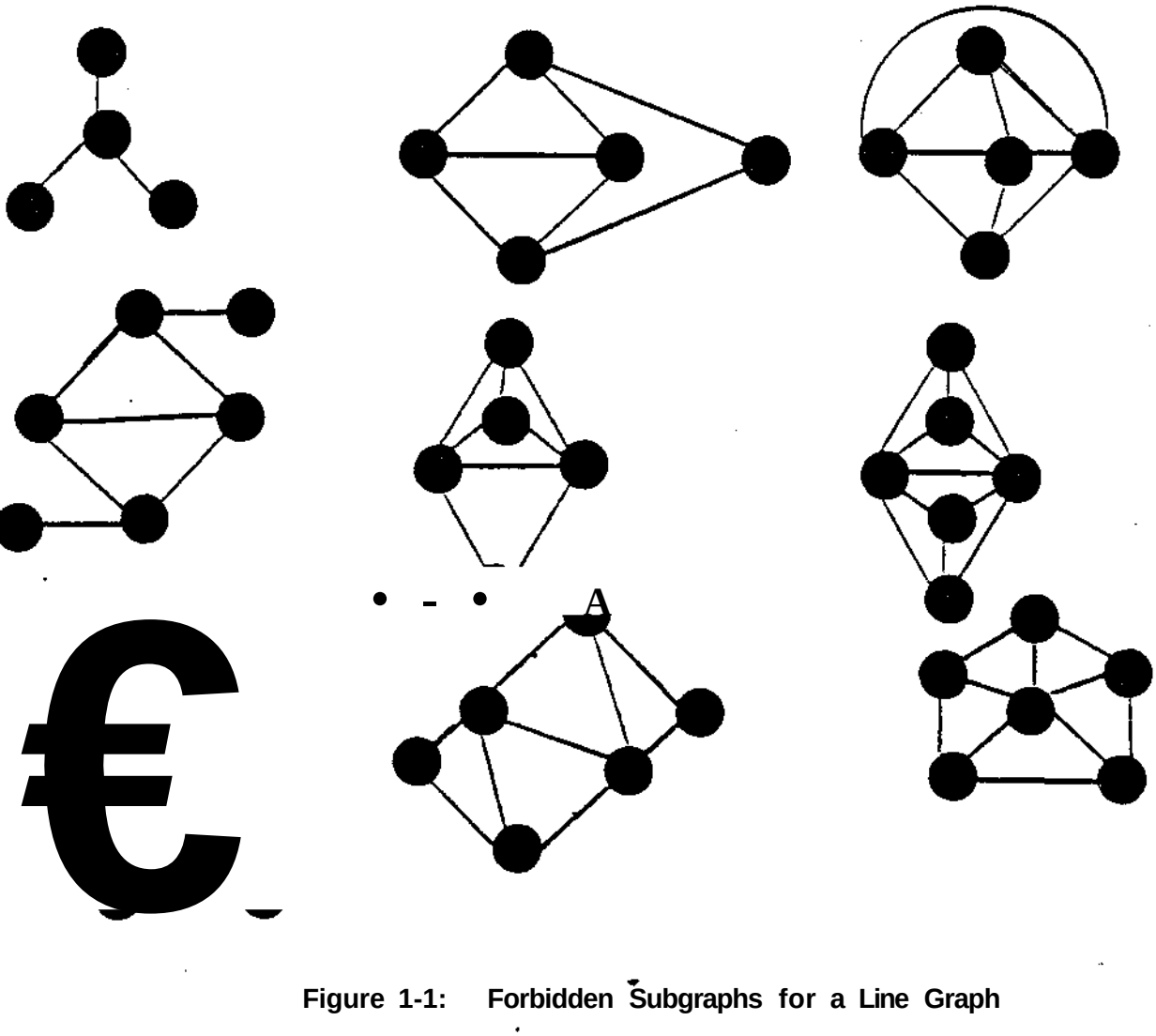


Figure 1-1: Forbidden Subgraphs for a Line Graph

We reflect first on the process of mathematical research. The question 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 [Pasc< 64, Hardy 40], intuitively conjectures additional data [Poincare 70] and thei attempts to satisfy the demands of rigor with formal definitions and proofs. Hov can we even begin to computerize a process so laden with value judgements an< vagueness?

Consider the diverse role of language in mathematical research. Th< mathematician carefully formalizes a research result in the language of definition: and proofs. The promulgation of the result however, is more likely to be in nature

(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 orientatioa

### 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.

### 1.6.1. 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 X V, I = {(x,y) | x,y e v}.
- If E is any subset of I the ordered pair <V,E> is used as the standard representation of a *graph.*
- If (x,y) s E such that x and y are distinct (x,y) is called an *edge* and is abbreviated as xy. If x,y e V and xy e E, vertices x and y are said to be *adjacent* and x is said to be a *neighbor* of y.
- A graph G = <V,E> is *undirected* when xy s E if and only if yx s E Otherwise the graph is *directed.*
- If (x,x) e 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 ail possible loops on V is 1 = {(x,x) | x e 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 = <$,£> is defined to be the *empty graph.*
- Two graphs $G_1$ s <V ,Ej> and $G_2$ = <V $_2$E $_2$> are *isomorphic* to each other if there exists a one-to-one mapping tr : $V_1$ -> $V_2$ such that xy « $E_1$ if and only if ir<x)ir(y) e E-$_2$ The mapping tr 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 ail 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 ail capital letters. The ordered pair <V,E> 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 = {true,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 $d = \{(p_1,s_1),(p_2,s_2),\ldots,(p_k,s_k)\}$ is *satisfied* by a graph G if and only if $p_i(G) = s_i$ for $i = 1,2,\ldots,k$. 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 < 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 < 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, $G_1$ = <{1,2,3},{12,13,23}> and $G_2$ = <{1,2,3,4,5},{12,34,25}> 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 (.-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-deschption 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 satisfiabie 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 satisfiabie. The number of satisfiabie 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 \s $L_y$ $L_2$, $L_3$, $L_{1n}$, $L_{2n}$ and $L_{3n}$ are ail extensions of $L_1$ and closely interrelated The inability of these six to express certain graph properties suggests a somewhat different extension of $t_r$ 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, 1, and 0. I is the Cartesian product V X v,

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

E is any subset of I, 1 is the set of ail loops on V,

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

and 0 is the empty set *j>.* 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 $\underline{S}$). 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 ≠) between any two edge sets constructed with the operators from the original four. Some sample $L_1$-expressions follow:

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

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

$I = E \cup E' \cup 1$

Our interpretation of these expressions will be in terms of the standard representation of the graph G by <V,E>. 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 ~) and cardinal inequality (denoted as *'t-)* 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 $L_r$ Characteristics which can be interpreted from $L_2$-expressions but not from $L_1$-expressions include:

(E)' ~ E

1 u E *h* E

Based again on the standard representation G = <V,E>, 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 (/*). 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:

(E)' < E

E < 1 u E

Based again on the standard representations G = <V,E>, 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:

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

Thus xy is in $E^*$ for G = <V,E> if and only if there is an alternating sequence (a *path*) $x, xv_1, v_1, v_1v_2, \ldots, v_k, v_k y$ 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:

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

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

Using the standard representation G = <V,E>, 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 = <V,E> is

symbol: E | I | 1 | 0

term: symbol | (term)' | (term) | (term ∪ term) | (term ∩ term)

expression: term = term | term ≠ 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 I as the Cartesian product V X V for the vertex set V of the graph:

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

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

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

and 0 as the empty set $\phi$.

We interpret the construction term = 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 term ≠ term is interpreted as *set inequality*. For edges sets $S_1$ and $S_2$, $S_1 \neq S_2$ if and only if $S_1 = S_2$ is false. $E \cap \underline{1} = 0$ 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 $G_2 = \langle\{1,2\},\phi\rangle$. 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 (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 $\underline{(term)}$ as the unary operator *complement*, which replaces an edge set by its complement will 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 "<=>" 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

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:

$\underline{|} \iff \underline{0}$

$\underline{0} \iff \underline{|}$

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

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

We interpret the constructions (term ∪ term) and (term ∩ 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:

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

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

and

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

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

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

term: symbol | (term ∪ term) | (term ∩ term)

expression: term = term | term ≠ term

This is the grammar we will use for $L_r$

We are now interested in simplifying strings such as

«$S_1$ n $S_2$) u ...) n (...)

From set theory we have, for any edge sets $S_y$ $S^\wedge$ $S^\wedge$.

($S_1$ u $S_2$) n $S_3$ <=> ($S_1$ n $S_3$) u ($S_2$ n $S_3$)

($S_1$ n $s_2$) u $S_3$ <=> ($S_1$ u $S_3$) n ($S_2$ u $S_3$)

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

I u S <=> I

I n S <=> S

0 u S <=> S

0 n S <=> 0

Thus we need only consider expressions of the form $S_1$ u $S_2$ and $S_1$ n $S_2$, where $S_1$ and $S_2$ are chosen from among E, $\underline{E}$, 1, $\underline{1}$, E and $\underline{E}$. We also know that T includes:

S u $\underline{S}$ <=> I

S n $\underline{S}$ <=> 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 ail possible unions and intersections of sets $S_r S_2 \_ . S_k$ 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.

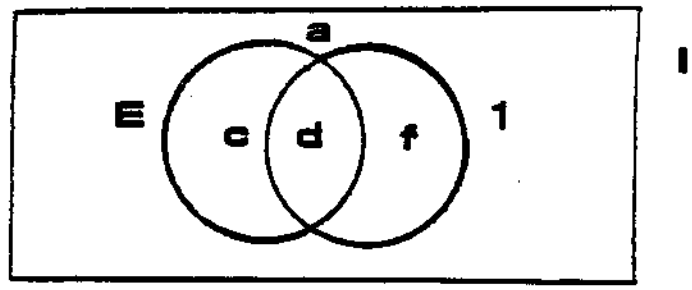



Figure 2-1: A Venn Diagram for Undirected Graphs

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

$$I \wedge E u i u 0 u \underline{E} u i$$

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 $\underline{E}$ n $\underline{i}$, the non-loop edges not in the graph

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

d denotes E n 1, the loops in the graph

f denotes $\underline{E}$ n 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^*-terms. Since these relations are symmetric and complementary, there *are* at most $2(^1| ) = 240$ $L_1$-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 $\underline{1} = 1 \cup \underline{E}$ is viewed in the Venn diagram representation as a ∪ c = a ∪ d ∪ f. Because we always have a = a, the non-trivial portion of this is c = d ∪ 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 $\langle\phi,\phi\rangle$, 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 | $V = \{1\}$, $E = \phi$ |
| 11 | acf | $V = \{1\}$, $E = \{11\}$ |
| 12 | acdf | $V = \phi$, $E = \phi$ |

**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 $\langle V,\underline{1}\rangle$; 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 $\langle V,\phi\rangle$; 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\ E, E, 1. Once again E. B, 1, I and 0 have inherent interpretations in the Venn diagram, leaving us with three sets (E, 1 and B) to explore in Figure 2-2.

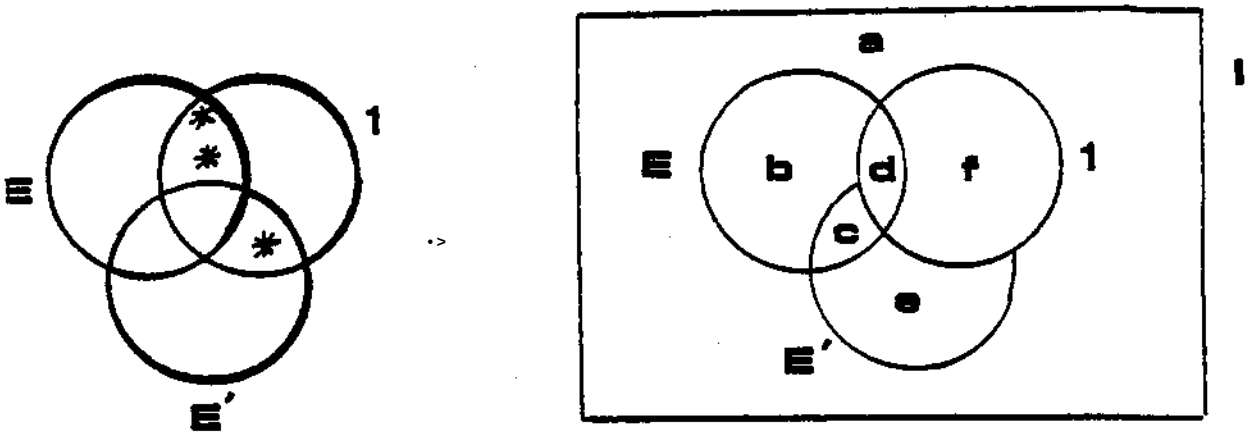



**Figure 2-2:** A Venn Diagram for Directed Graphs

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 • (E n E n 1) then xy *m* 1 and x = y. If xx e B then xx « E and xx * E Thus (E n E n i) is empty.

For the *» region: If xy « (E n B n 1) then xy e 1 and x * y. If xx *m* E then xx s B and xx * B. Thus (E n B n 1) is empty.

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

- a denotes E n B n 1, the non-loop edges not in the graph whose reverses are not in the graph either
- b denotes E n B n 1, the non-loop edges in the graph whose reverses are not in the graph

- c denotes $E \cap E' \cap \underline{1}$, 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 $\underline{E} \cap E' \cap \underline{1}$, the non-loop edges not in the graph whose reverses are in the graph
- f denotes $(\underline{E} \cup \underline{E}') \cap 1$, 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 = $\underline{1}$ | | | |
| 19 | abef | E = I | | | |
| 20 | bcde | E = 0 | | | |
| 21 | bcef | E = 1 | | | |
| 22 | abcde | V = {1}, E = $\phi$ | | | |
| 23 | abcef | V = {1}, E = {11} | | | |
| 24 | abcdef | V = $\phi$, E = $\phi$ | | | |

**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. Fo

n = 2 there are 7 finite non-isomorphic directed graphs, 5 of which have distinct signatures in $L_1$ and two of which (<{1,2},{12,11}> and <{1,2},{12,22}>) 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., <1 0 u u u>) 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. L1-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 i
value of n.

**L1-GENERATOR**

```
Read L1-characterization S
Case: /*minimal case knowledge*/
        abcdef in S:  output <ϕ,ϕ>, 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
```

```
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 <V,E>
```

### 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 L1-TESTER is as follows:

L1-TESTER

```
Read graph G » <V,E>
Read L1-description S
Create all loops and edges and label them u
Do for each xy 6 E, x and y distinct
        Relabel edge xy "T"
Do for each xx e E
        Relabel loop xx "T"
If (I xy, distinct x and y, labelled u and yx labelled u, a in S)
 or (i xy, distinct x and y, labelled u and yx labelled T, be in S)
 or (3 xy, distinct x and y, labelled T and yx labelled T, c in S)
 or (3 xx labelled u and f in S)
 or (3 xx labelled T and d in S)
        then print, "false"
        else print "true"
```

This testing algorithm is quadratic in n.

### 2.2.6. 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 = <V,E> is

| | |
|---|---|
| symbol: | E \| I \| 1 \| 0 |
| term: | symbol \| (term)' \| (term) \| (term ∪ term) \| (term ∩ term) |
| expression: | term = term \| term ≠ term \| term ~ term \| term ≁ term |

We interpret the construction term ~ 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 term ≁ term is interpreted as *inequality of set cardinality*. For edge sets $S_1$ and $S_2$, $S_1 \not\sim S_2$ 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: $E \mid I \mid 1 \mid 0 \mid E' \mid \underline{E} \mid \underline{E}' \mid \underline{1}$

term: symbol | (term $\cup$ term) | (term $\cap$ term)

expression: term = term | term $\neq$ term | term $\sim$ term | term $\not\sim$ 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 $\not\sim$. 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 $\sim$. 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 ~ 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 ~ c |
| 2 | c | 16 | af ~ d |
| 3 | d | 17 | cd ~ a |
| 4 | f | 18 | cd ~ f |
| 5 | a ~ c | 19 | cf ~ a |
| 6 | a ~ d | 20 | cf ~ d |
| 7 | a ~ f | 21 | df ~ a |
| 8 | c ~ d | 22 | df ~ c |
| 9 | c ~ f | 23 | ac ~ df |
| 10 | d ~ f | 24 | ad ~ cf |
| 11 | ac ~ d | 25 | af ~ cd |
| 12 | ac ~ f | 26 | a ~ cdf |
| 13 | ad ~ c | 27 | c ~ adf |
| 14 | ad ~ 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} | $\phi$ |
| acf | {1} | {11} |
| acdf | $\phi$ | $\phi$ |
| | | |
| a, c ~ d ~ f | {1,2} | {12,11} |
| c, a ~ d ~ f | {1,2} | {11} |
| ad, c ~ f | {1,2,3} | {12,13,23} |
| | | |
| af, c ~ d | {1,2,3} | {12,13,23,11,22,33} |
| cd, a ~ f | {1,2,3} | $\phi$ |
| cf, a ~ 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 $c \sim f$ and $cf \sim d$. 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 ~ 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 ~ 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 L2OI on Directed Graphs

for L1-CGENERATOR. 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 <a,b,c,d,e,f,n> 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 • <V,£>
    Calculate a, b, c, d, e, f and n
    For I • 1 to k
        Read S(l)
        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.

### 2.3.5. 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 ail 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 3, $L_2$-characterization may be an adequate categorization

$L_2$ is certainly an improvement on $L_r$ 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 | I | 1 | 0

term: symbol | (term)' | (term) | (term ∪ term) | (term ∩ term)

expression: term = term | term ≠ term | term ~ term | term ≁ term | term < term

We interpret the construction term < 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 | I | 1 | 0 | E' | $\underline{E}$ | $\underline{E}$' | $\underline{1}$

term: symbol | (term ∪ term) | (term ∩ term)

expression: term = term | term ≠ term | term ~ term | term $\not\sim$ term | term < 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:

$\underline{E} < E'$

$1 \cap E < \underline{E}'$

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 term $\not\sim$ term, $L_3$ uses term < 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 $term_1 < term_2$ and $term_2 < term_1$ are refinements on the $L_2$-expression $term_1 \neq term_2$. Some of these, such as ac < a, would be mathematically impossible. If we restrict $term_1 < term_2$ 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 | Largest Class 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 |

Tabla 2-8: Characterizations for Undirecti3d GraDhs 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 a 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 ^-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 $t_3$ 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: $E \mid E^* \mid I \mid 1 \mid 0$

term: symbol | (term)' | (term) | (term ∪ term) | (term ∩ term)

expression: term = term | 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: $E \mid E^* \mid I \mid 1 \mid 0 \mid E' \mid \underline{E} \mid \underline{E}' \mid \underline{1} \mid E^{*'} \mid \underline{E^*} \mid \underline{E^{*'}}$

term: symbol | (term ∪ term) | (term ∩ term)

expression: term = term | term ≠ term

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

$\underline{E^*} = 0$

$E^* \cap \underline{1} = E$

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 = <V,E>, 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, $E^*$, I, 1, 0, $\underline{1}$ 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 = <V,E>. 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 $\underline{E^*} \cap \underline{1}$, non-loops not in the transitive closure of the edge set
- c denotes $E \cap \underline{1}$, 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 $E^* \cap \underline{E} \cap \underline{1}$, edges not in the graph but in the transitive closure of its edge set

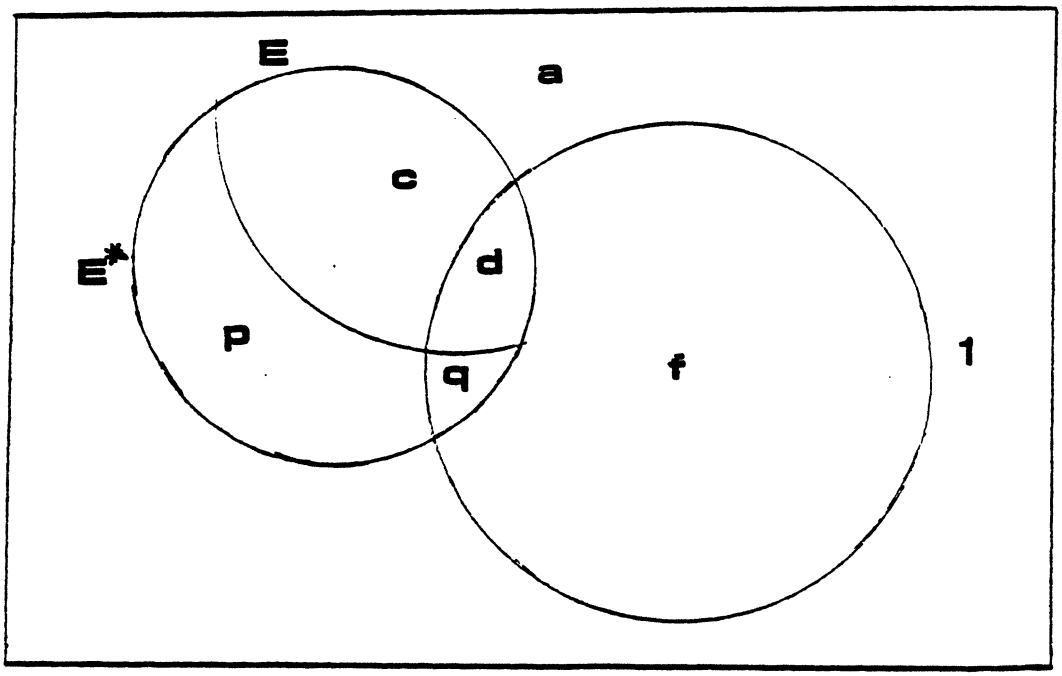



**Figure 2-3:** A Venn Diagram for Undirected Graphs in $L_1^*$

- q denotes $E^* \cap \underline{E} \cap 1$, 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 = <V,E> 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":

i. 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).

ii. If c then pq.

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

iii. If dfq then acdfpq.

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

iv. 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).

v. 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).

vi. 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,<br>some isolated vertices,<br>not every connected component complete,<br>not every unisolated vertex has a loop |
| 2 | f | 4 | disconnected, some edges, some loops,<br>no isolated vertices,<br>not every connected component complete,<br>not every unisolated vertex has a loop |
| 3 | p | 3 | disconnected, some edges, some loops,<br>some isolated vertices,<br>every connected component complete,<br>not every unisolated vertex has a loop |
| 4 | q | 4 | disconnected, some edges, some loops,<br>some isolated vertices,<br>not every connected component complete,<br>every unisolated vertex has a loop |
| 5 | af | 3 | connected, some edges, some loops,<br>no isolated vertices,<br>not every connected component complete,<br>not every unisolated vertex has a loop |
| 6 | df | 5 | disconnected, some edges, loopfree,<br>no isolated vertices,<br>not every connected component complete,<br>not every unisolated vertex has a loop |
| 7 | dp | 3 | disconnected, some edges, loopfree,<br>some isolated vertices,<br>every connected component complete,<br>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,<br>no isolated vertices,<br>every connected component complete,<br>not every unisolated vertex has a loop |
| 9 | fq | 4 | disconnected, some edges, some loops,<br>no isolated vertices,<br>not every connected component complete,<br>every unisolated vertex has a loop |
| 10 | pq | 3 | disconnected, some edges, some loops,<br>some isolated vertices,<br>every connected component complete,<br>every unisolated vertex has a loop |
| 11 | adf | 3 | connected, some edges, loopfree,<br>no isolated vertices,<br>not every connected component complete,<br>not every unisolated vertex has a loop |
| 12 | afp | 2 | connected, some edges, some loops,<br>no isolated vertices,<br>every connected component complete,<br>not every unisolated vertex has a loop |
| 13 | afq | 3 | connected, some edges, some loops,<br>no isolated vertices,<br>not every connected component complete,<br>every unisolated vertex has a loop |
| 14 | cpq | 3 | disconnected, edgeless, some loops,<br>some isolated vertices,<br>every connected component complete,<br>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.<br>no isolated vertices.<br>every connected component complete.<br>not every unisolated vertex has a loop |
| 16 | fpq | **4** | disconnected, some edges, some loops.<br>no isolated vertices.<br>every connected component complete.<br>every unisolated vertex has a loop |
| 17 | adfp | **2** | 1 |
| 18 | afpq | **2** | 1 |
| 19 | cfpq | **2** | 1 |
| **20** | **acdpq** | **1** | <{1}.*> |
| **21** | **acfpq** | **1** | <{1},{11}> |
| **22** | **acdfpq** | **0** | **<ϕ,ϕ>** |

**Table** 2-10: Undirected Graph Signatures in $L_r$ continued

if c = 1 then f £ n-2

if k(kTi)/2 < c & k(k+1)/2 then f £ n-k for k » 2,3,4,.~

O S q ^ min(2c,n-d)

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

From these we observe that the values for ad and n are independent variables and will determine the possible values for the dependent 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 E, E, E*, E£, E*, and E£. 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

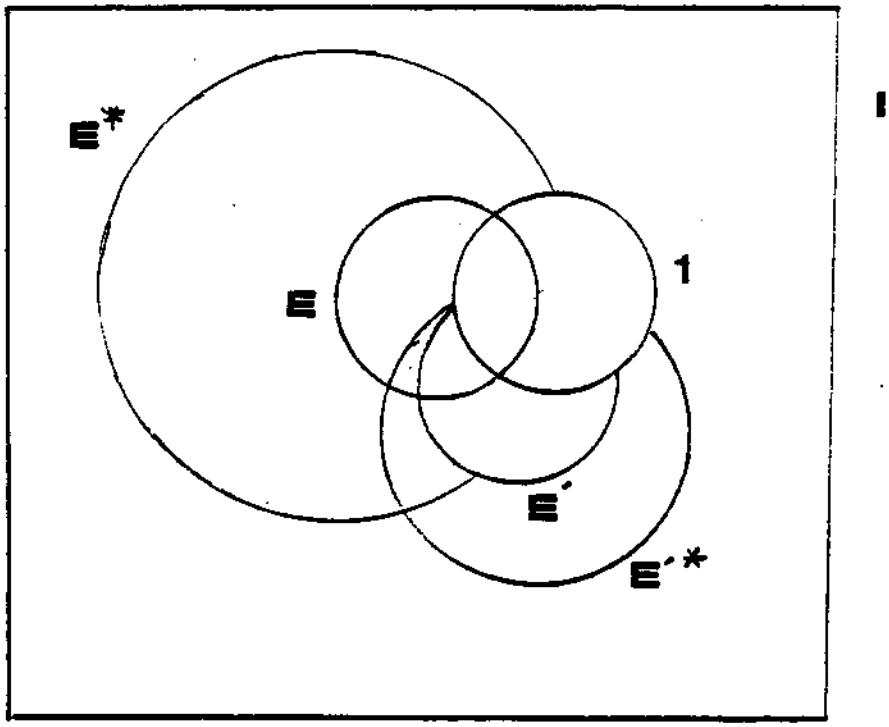



**Figure 2-4:** A Preliminary Venn Diagram for Directed Graphs in $L_1^*$

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(<V_T,E_T>,<V,E>)
Either V <- V ∪ {x} for x ∈ V_T, x ∉ V
    or E <- E ∪ {yz} for  y,z ∈ V, yz ∈ E_T, yz ∉ E
If G = G_T
then halt
else  CONSTRUCT(<V_T,E_T>,<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

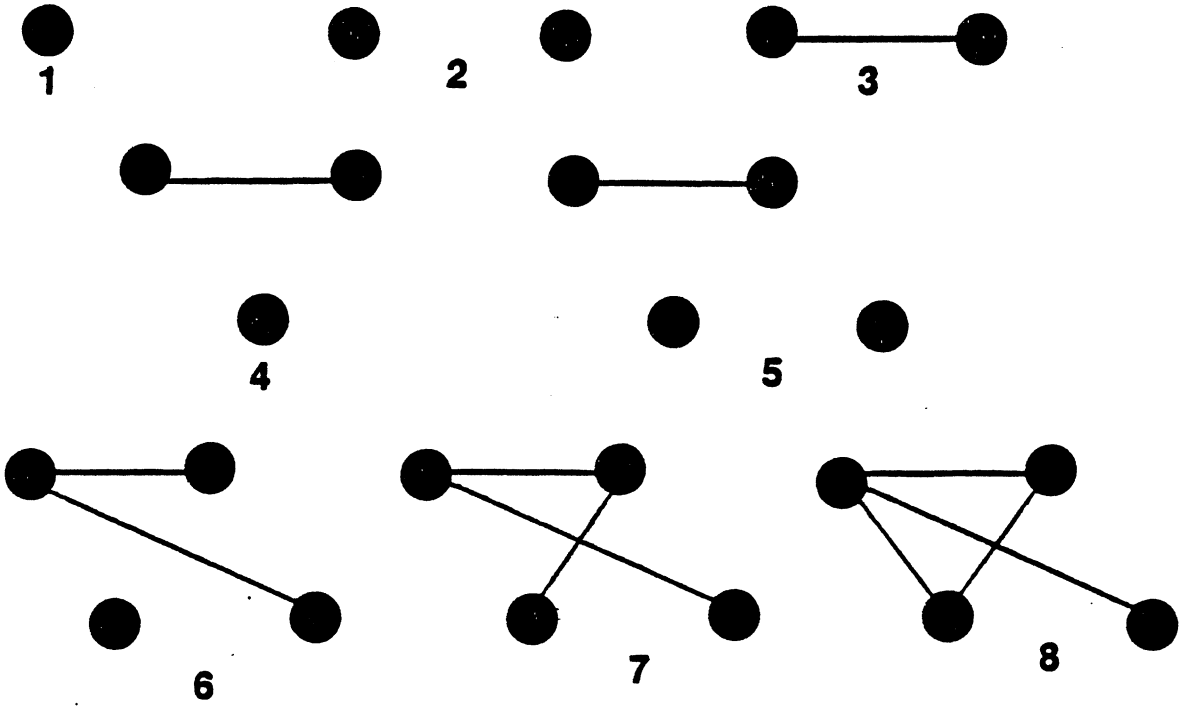



**Figure 3-3:** A Sample Run of GENERATE

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:

$$G_E = \{(<1>,\phi),(<1,2>,\phi),(<1,2,3>,\phi),\ldots\}$$

or if we let $E_k = <\{1,2,3,\ldots,k\},\phi>$, 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«V,E>)**

V < - V U {x} **for x * V**

**Either output <V,E>**

**or EDGELESS(<V,E>)**

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

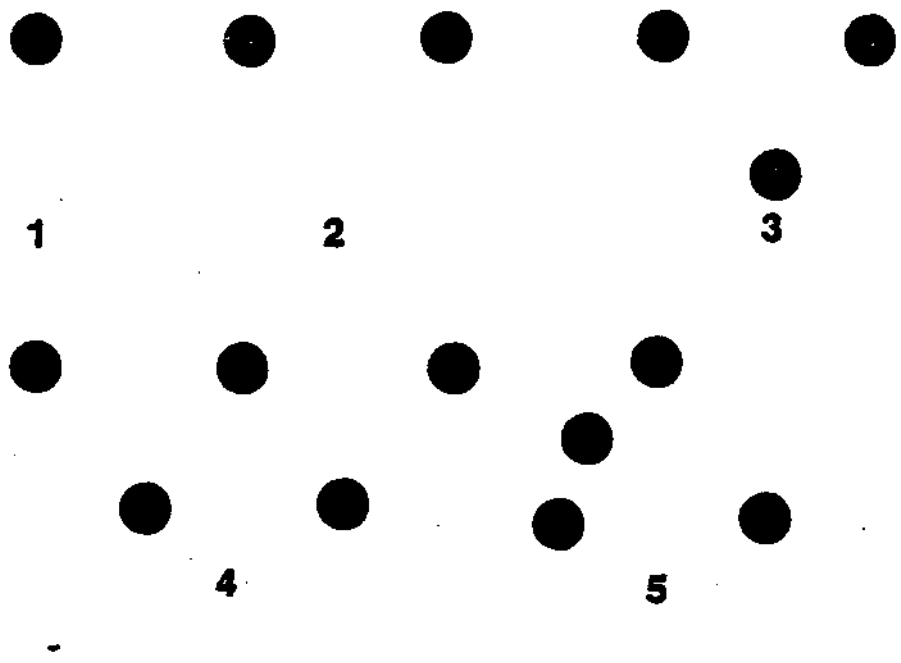



**Figure 3-5: A Sample Run of EDGELESS**

**EDGELESS never changes E, E will remain empty, and every element of $0_{edgeless}$ will be of the form <*V,$*>, La, edgeiess. Thus the algorithm produces only edgeless graphs and $0_{edgeless}$ £ $G_{£}$. Since any (<1,2,3,.«,k>,$> in $G_E$ may be constructed by inserting 1, then 2, and so on, up to k, EDGELESS produces all edgeless graphs, i.e., $G_E$ £ $0_{edgeless}$. Therefore $G_E$ = $0_{edgeless}$ 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.**

### 3.2. Recursive Graph Grammars

Now that we have clarified iterative graph construction as a definition technique, this section defines recursive graph grammars to implement it. GENERATE and EDGELESS are reformulated in this context.

A recursive grammar for graph properties has concise terminal expressions whose semantic interpretations are algorithms similar to GENERATE and EDGELESS. There are three key components in such a grammar:

- the primitive operations permitted on the graph (such as adding an edge)
- the seed graphs on which the algorithm may be called initially (such as $K_1$)
- the selector conditions under which choices are made during execution (such as "for $x \notin V$").

More formally, let a *recursive graph grammar* R be an ordered triple $R = \langle P,L,\Sigma \rangle$ where P is the language for primitive operators permitted on the graph, L is a seed language used to specify the *seed set* (graphs on which the algorithm may be called initially), and $\Sigma$ is a selector language in which the selector conditions are formulated. A terminal R-expression will be of the form $p = \langle f,S,\sigma \rangle$, where f is a terminal P-expression, S is a terminal L-expression and $\sigma$ is a terminal $\Sigma$ expression. The semantic interpretation of this R-expression is an algorithm which iterates an unspecified number of times. On each iteration the *selector* $\sigma$ chooses one or more vertices and/or edges with respect to the current graph G, and then f modifies G, using those choices, to produce a new G. Initially G is a *seed graph* selected from the set of graphs which is the semantic interpretation of the expression S. More formally, an *R-property* is the following semantic interpretation of the triple $\langle f,S,\sigma \rangle$ as a recursive algorithm called on any graph described by S:

f(G) = G if enough
= f(G') where G' = f(G) using elements from G or new to it selected by $\sigma$ in order to apply f

Any G for which S is true has the R-property, and any output from the algorithm

on such a G has the R-property. In the event that no vertices or edges satisfy $\sigma$, 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 $\sigma$." 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), $f^5(G)$ and $f^{17}(G)$ under $\sigma$ 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 $\sigma$ 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

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

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 {Kj and *a* is "y,z **s** 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 E.

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 modificatioa

| **Primitive** | **Effect** | **Representation** |
|---|---|---|
| N | No change | N<V,E> = <V,E> |
| $A_x$ | **Add** vertex x | $A_x$<V,E> a <Vu{x},E> |
| $A_{xy}$ | **Add edge** xy | $A_{xy}$<V,E> « <V,EU{xy}> |
| $D_x$ | **Delete vertex** x | $D_x$<v,E> a <V-{x},E> |
| $D_{xy}$ | **Delete edge** xy | $D_{xy}$<V,E> = <V,E-{xy}> |
| $I_{xv_1 \sim v_k}$ | **Identify vertices** | $I_{xv,...v,}$<V,E> = <V-{v.},E\|$_{*v.<-x}$> |
| $F_{xyv_1 \sim v_k}$ | **Fragment vertex** x into vertices x **and** y | $F_{xyv\text{-}.\text{-}.v,}$<V,E> a <VU{y},E\|$_{xv.->yv.}$> |
| L | Loop on all vertices | L<V,E> a <v,E u 1> |
| L | Unloop on all vertices | L<V,E> = <V,E n 1> |

**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 ) or an edge (A , D ), for the merger of k+1 vertices (I ), for the splitting of one vertex into two (F ), and for the introduction (U and removal (L) of loops on all the vertices.

Now we postulate some possible Ps for the R-grammar <P,L,I>. Each P has a set n = {$ir_r jr_2$,.-,$jr_k$} of terminal symbols and the following grammatical rules:

I -> I + I | I I

This P grammar permits both primitive operators (members of II) 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}

$p_2$ = $p_1$ U <$^D x$ ' $^o xy$}

$p_3$ = $p_2$ u {U L}

$p_4$ s $p_2$ u {'xv•V

$p_S$ = $p_3$ u {'xv<$^F xv$}

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 + $gg^1$ 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 *a* (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 | $A_{v_1x}A_{v_2x}...A_{v_nx}A_{x}$ |
| $Y_{u_1...u_k}$ | Add cycle $u_1u_2...u_ku_1$ | $A_{u_1u_2}A_{u_2u_3}...A_{u_{k-1}u_k}A_{u_ku_1}A_{u_1}A_{u_2}...A_{u_k}$ |
| $\underline{Y}_{u_1...u_k}$ | Delete cycle $u_1u_2...u_ku_1$ | $D'_{u_1}D'_{u_2}...D'_{u_k}D_{u_1u_2}D_{u_2u_3}...D_{u_{k-1}u_k}D_{u_ku_1}$ |
| | | where $D'_i$ = $D_i$ if degree of i is 0 |
| | | = N else |

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

$u_1,u_2,...,u_k$" 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 $\sigma$ and the resources needed to update the graph.

For L in the R-grammar <P,L,$\Sigma$>, 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 $L_\Omega$, the language which precisely lists the vertices and edges of a graph.

The $\Sigma$ in the R-grammar <P,L,$\Sigma$> 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 $\Sigma$ vertices are $v_1, v_2, \ldots$ and edges are ordered pairs of vertices. We offer the following selector languages with their generating grammars. Many others are, of course, possible:

$\Sigma_1$

A formal grammar for $\Sigma_1$ is

| | |
|---|---|
| I | -> I,I \| vertex sign V \| edge sign E |
| vertex | -> $v_1$ \| $v_2$ \| $v_3$ \| ... |
| edge | -> (vertex,vertex) |
| sign | -> $\in$ \| $\notin$ |

Selector expressions such as

$x \in V$

or

$yz \notin E$

are possible in $\Sigma_1$. Note that we could produce the selector for GENERATE in $\Sigma_1$ as follows:

I -> I,I

-> vertex sign V, vertex sign V

-> $y \in V$, $z \in V$

$\Sigma_2$

A formal grammar for $\Sigma_2$ is

| | |
|---|---|
| I | -> I,I \| vertex sign V \| edge sign E \| vertex $\neq$ vertex \| edge $\neq$ edge |
| vertex | -> $v_1$ \| $v_2$ \| $v_3$ \| ... |
| edge | -> (vertex,vertex) |
| sign | -> $\in$ \| $\notin$ |

$\Sigma_2$ contains all the terminal expressions of $\Sigma_1$. In addition, selector expressions such as

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

are possible in $\Sigma_2$, but not in $\Sigma_1$. 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 ≠ vw" for undirected graphs means that neither "y = v and z = w" nor "y = w and z = v" is true. The expression "yz ≠ vw" for directed graphs means "y ≠ v and z ≠ w."

$\Sigma_3$

A formal grammar for $\Sigma_3$ is

| | |
|---|---|
| I | -> I,I \| vertex sign V \| edge sign E \| vertex ≠ vertex \| edge ≠ edge \| d(vertex) rel number |
| vertex | -> $v_1$ \| $v_2$ \| $v_3$ \| ... |
| edge | -> (vertex,vertex) |
| sign | -> ∈ \| ∉ |
| rel | -> = \| > \|≥ \| < \| ≤ |
| number | -> 0 \| 1 \| 2 \| ... |

$\Sigma_3$ contains all the terminal expressions of $\Sigma_1$ and $\Sigma_2$. In addition, selector expressions such as

x ∈ V, d(x) > 1

are possible in $\Sigma_3$, but not in $\Sigma_1$ or $\Sigma_2$. 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 | y ∈ V, xy ∈ E, x ≠ 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 ∈ 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.

$\Sigma_4$

A formal grammar for $\Sigma_4$ is

| | |
|---|---|
| I | -> I,I \| vertex sign V \| edge sign E \| vertex ≠ vertex \| edge ≠ edge \| d(vertex) rel number |
| vertex | -> $v_1$ \| $v_2$ \| $v_3$ \| ... |
| edge | -> (vertex,vertex) |
| sign | -> ∈ \| ∉ |
| rel | -> = \| > \| ≥ \| < \| <= |
| number | -> max \| n \| 0 \| 1 \| 2 \| ... |

$\Sigma_4$ contains all the terminal expressions of $\Sigma_1$, $\Sigma_2$ and $\Sigma_3$. In addition, selector expressions such as

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

are possible in $\Sigma_4$, but not in $\Sigma_1$, $\Sigma_2$ or $\Sigma_3$. 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.

$\Sigma_5$

A formal grammar for $\Sigma_5$ is

| | |
|---|---|
| I | -> ~(I) \| I,I \| vertex sign V \| edge sign E \| |
| | vertex ≠ vertex \| edge ≠ edge \| |
| | d(vertex) rel number \| d(vertex) rel variable sum number \| |
| | \| {vertexset} ∩ V \| rel number \| |
| | \| {edgeset} ∩ E \| rel number \| variable rel number \| |
| | \| {vertexset} \| = \|V\| |
| vertex | -> $v_1$ \| $v_2$ \| $v_3$ \| ... |
| edge | -> (vertex,vertex) |
| sign | -> ∈ \| ∉ |
| rel | -> = \| > \| ≥ \| < \| ≤ |
| number | -> n \| 0 \| 1 \| 2 \| ... |
| vertexset | -> vertex \| vertex,vertexset \| {vertex\|I} |
| edgeset | -> edge \| edge,edgeset \| {edge\|I} |
| variable | -> i \| j \| k \| ... |
| sum | -> + \| - |

The expression ~(I) is interpreted semantically as "not I," providing the negation of any expression. The expression |A ∩ B| is interpreted semantically as the cardinality of the intersection of the sets A and B. An expression of the form variable k ≥ n is intended to refer to the index numbers on the vertices. (See the property EULERIAN for an example.) $\Sigma_5$ contains all the terminal expressions of $\Sigma_1$, $\Sigma_2$, and $\Sigma_3$, but not $\Sigma_4$, because $\Sigma_5$ lacks max.. Selector expressions such as

$k \geq 5$

or

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

are possible in $\Sigma_5$, but not in $\Sigma_1$, $\Sigma_2$, $\Sigma_3$ or $\Sigma_4$.

$\Sigma_6$

A formal grammar for $\Sigma_6$ is the $\Sigma_5$ grammar with one additional production:

I -> statement

"Statement" is any English language sentence. $\Sigma_6$ is a deliberate catchall, to be used, like $L_\Omega$, when all else fails.

In the event that the selector makes no restrictions at all, $\Sigma$ is designated as $\phi$.

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 $\Sigma$. (The reader is invited to define additional appropriate languages.)

### 3.4. The Floor of a Graph Property

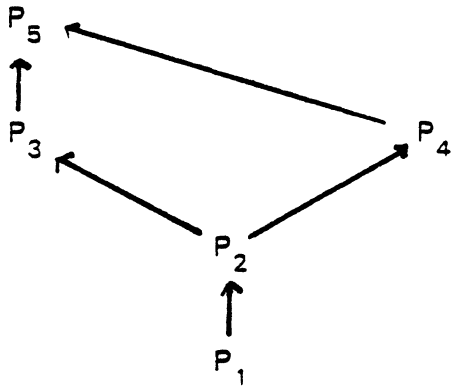
Part of the challenge in writing a graph property recursively is choosing an appropriate recursive language. This section explores a minimality condition for R-properties. GENERATE and EDGELESS are again used as examples.

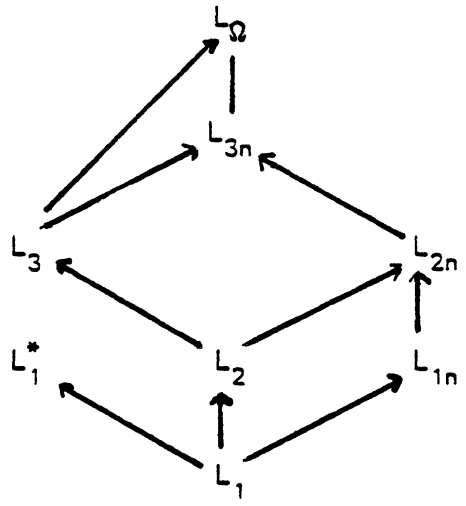
We return now to GENERATE, which we had identified as

$(A_x + A_{yz})^*(K_1)$ 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 $\Sigma$'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 $\Sigma$'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 $p = \langle f,S,\sigma \rangle$ 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



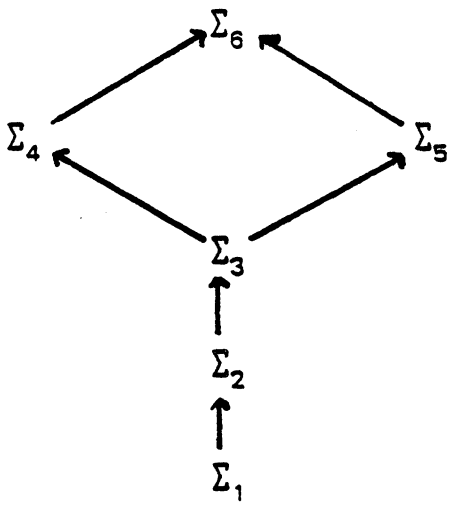




**Figure 3-6:** Orderings for Property Languages

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 $\Sigma_1$ are adequate. Since the expressions $\underline{E} \cap \underline{1} = 0$ and $E = 0$ 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 $\langle P_1, L_1, \phi \rangle$ 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) */ess* 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 = <f,S,a> 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] » <f~[1],S,a~[1]> is said to be the *inverse* of another terminal R-expression p = <f,S,a> if and only if the *testing* semantic interpretation of p~[1] returns 'TRUE* on ail outputs of the *generator* which is the R-property defined by p, and "FALSE" on ail other graphs. The testing semantic interpretation of p~[1] = <f"[1],S,a"[1]> is the following recursive algorithm:

f~[1](G) = TRUE if G is described by S

= f~[1](G') where <*5* = f~[1](G) using elements from G selected by a""[1] in order to apply f""[1]

[3] FALSE if G is not described by .S and a"[1] is not applicable

Note that *every* selection suiting the requirements in *o* guarantees the correct results, not simply *some* selectioa 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 $p\sim^{1}$ is $(f\sim^{1}a\sim^{1})^{*}(G)$. The seed set S is implicit in this representation. Thus $(f\sim^{1}a\sim^{1})^{*}(G)$ is interpreted as "apply $f\sim^{1}$ until an answer is reached, La, G is described by S or $f"^{1}$ cannot be applied because $a"^{1}$ fails." The number of iterations required for testing may vary with $a"^{1}$ 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 constructioa 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 ail 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 ail 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

$A^*_x(K_1)$ where $x * V$

and was modified to

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

$A_x^{-1}$ = $D_x$ if $x * V$ before $A_x$
« N else

and

$A''^{1}_{xy}$ = $D_{xy}$ if $x$ «*i* E before $A_{xy}$
= N else

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

$(A_x + A\text{^ffl^})$ where y,z e V

it is easier to invert as

$(A_x + A_y\text{^(K^}$ where $x * V$, y,z e V, $yz * E$

$A_x$ may be applied to a graph G = <V,E> 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

$$A_x^{-1} = D_x$$
$$A_{xy}^{-1} = D_{xy}$$
$$D_x^{-1} = A_x$$
$$D_{xy}^{-1} = D_{xy}$$
$$N^{-1} = N$$

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

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

The construction of $\sigma^{-1}$ from $\sigma$ 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 ∈ V, d(x) = 1." A *pre-profile* is a profile immediately before the application of an operator. Although $\sigma$ initially constitutes a profile, we expand $\sigma$ to $\sigma_{pre}$. This new pre-profile excludes ineffectual (equivalent to N) operations. $\sigma_{pre}$ 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 $\sigma^{-1}$ from $\sigma$ is embodied in

**RULE 5**

Let $\sigma$ be a pre-profile of those variables involved. Expand $\sigma$ to $\sigma_{pre}$. Compute the changes to $\sigma_{pre}$ caused by f. The new description, $\sigma^{-1}$, is in $\Sigma_4$. $\sigma^{-1}$ is now a post-profile of the variables after f. (If the selector language for p is $\Sigma_5$ or $\Sigma_6$, the new description must also be constructed in $\Sigma_5$ in $\Sigma_6$.)

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 $\Sigma_4$. 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 $\Sigma_1$, $\Sigma_2$, $\Sigma_3$ or $\Sigma_4$ 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 $\sigma$ is empty. We expand $\sigma$ to $\sigma_{pre} = x \notin V$ 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 $\sigma^{-1}$ 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 $\sigma^{-1}$. Therefore, the following algorithm tests to see if an arbitrary graph is edgeless:

$f^{-1}(G)$ = TRUE if G is $K_1$

= $f^{-1}(D_x(G))$ where $x \in V$, $d(x) = 0$

= FALSE if G is not $K_1$

and there does not exist $x \in V$ such that $d(x) = 0$

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 $G \notin G_p$.

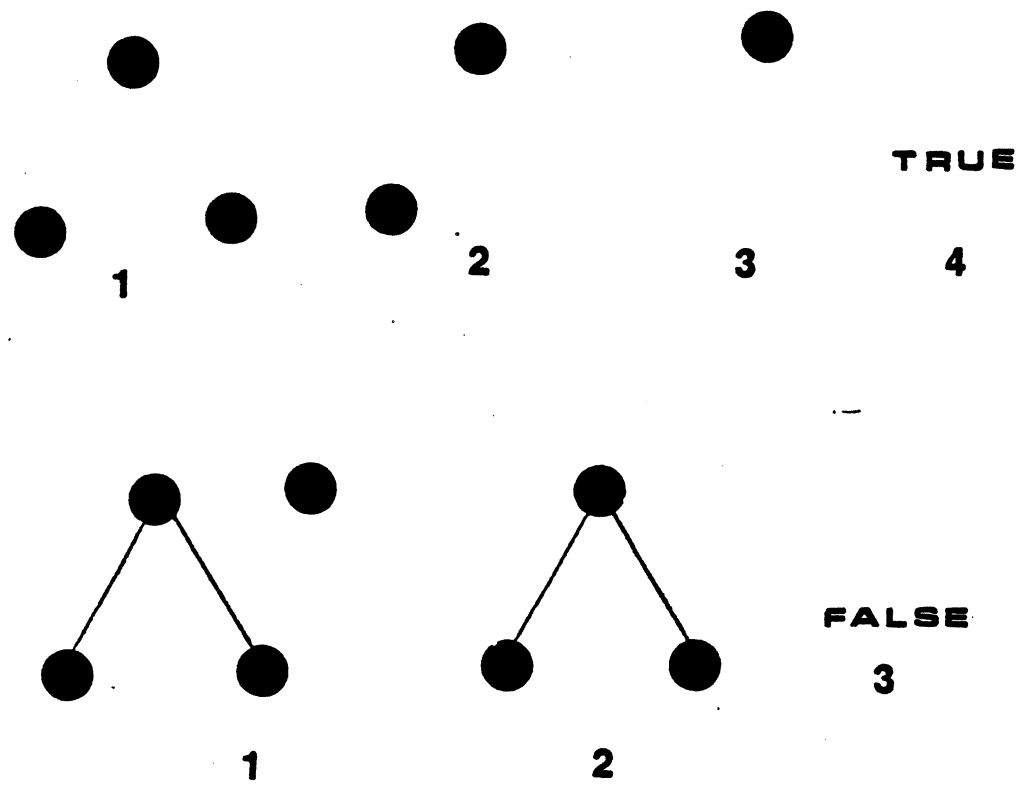



Figure 3-7: EDGELESS$^{-1}$ in Operation

For GENERATE we have

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

The pre-profile $\sigma$ is $yz \in E$, which we expand to $x \notin V$, $y,z \in V$, $yz \notin E$. The post-profile $\sigma^{-1}$ 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 $\sigma^{-1}$. 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 $K_1$

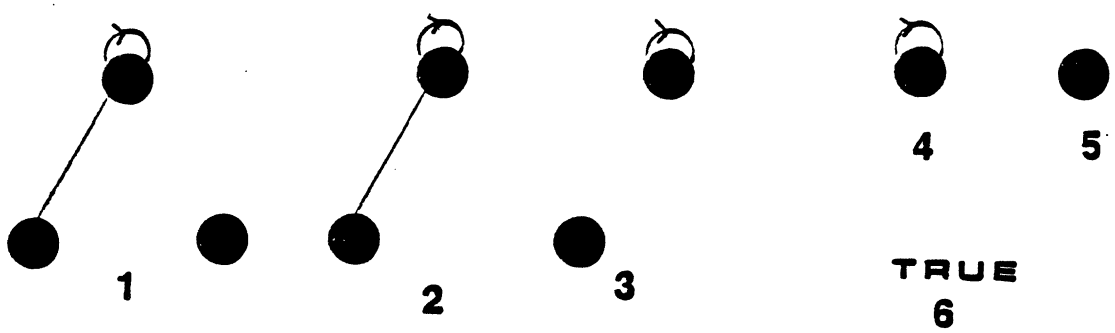
= $f^{-1}((D_x + D_{yz})(G))$ where $x,y,z \in V$, $yz \in E$, $d(x) = 0$

= 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 $\sigma^{-1}$ to succeed, i.e., we need to find x *or* yz but not both, as distinguished by separate lines in the FALSE selector $\sigma^{-1}$. In Figure 3-8 we show the algorithm operating on a graph $G \in G_p$ and a graph $G \notin G_p$.




<{1,2},{12,13}> <{1,2},{13}> <{1},{13}> FALSE

1 2 3 4

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

From now on we will describe inverses merely by stating $f^{-1}$, $\sigma_{pre}$ and $\sigma^{-1}$.

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 $\Sigma_4$ whenever possible, we expect the floor for $p^{-1}$ to involve $\Sigma_4$, regardless of the $\Sigma$ used in p. To the extent that features of $\Sigma_4$ are not applicable, lower $\Sigma$'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 $G \notin G_p$ into $f^{-1}(G) \notin G_p$, 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

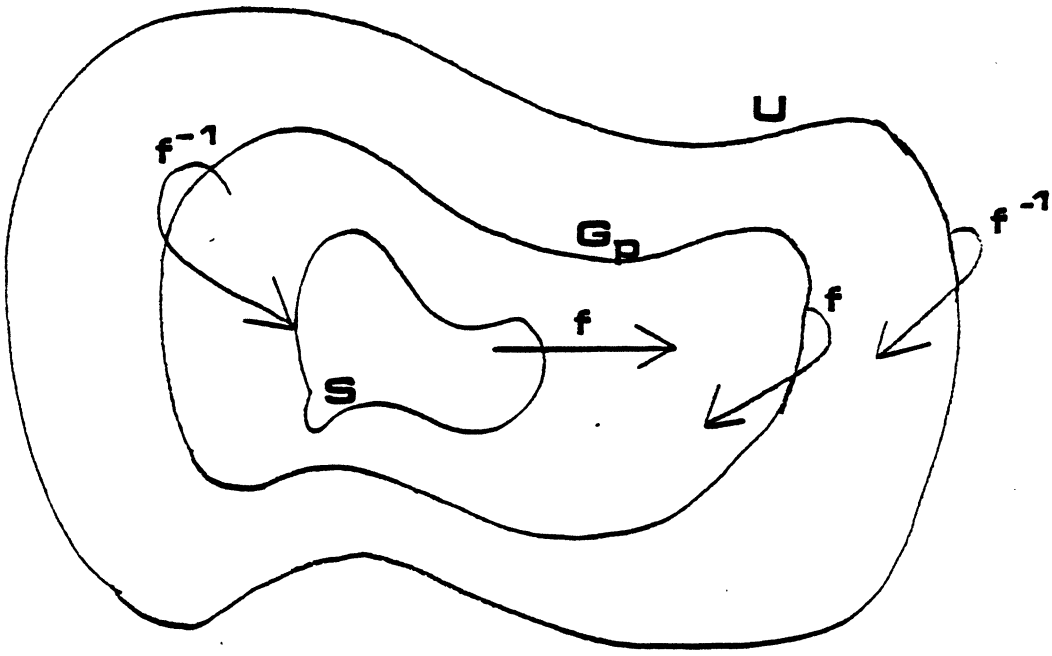



**Figure 3-9:** The Behavior of f and $f^{-1}$ on the Set of All Graphs

chapter and the next, we strive to work in the simplest floor possible. Because our selection languages $\Sigma$ are reasonably limited (e.g., there is no notion of a path before $\Sigma_6$), 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, $v_1, v_1v_2, v_2, v_2v_3, v_3, \ldots, v_{k-1}, v_{k-1}v_k, v_k$ 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 $v_1v_2\ldots v_k$.) 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 $C_{v_1v_2\ldots v_k}$. 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

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

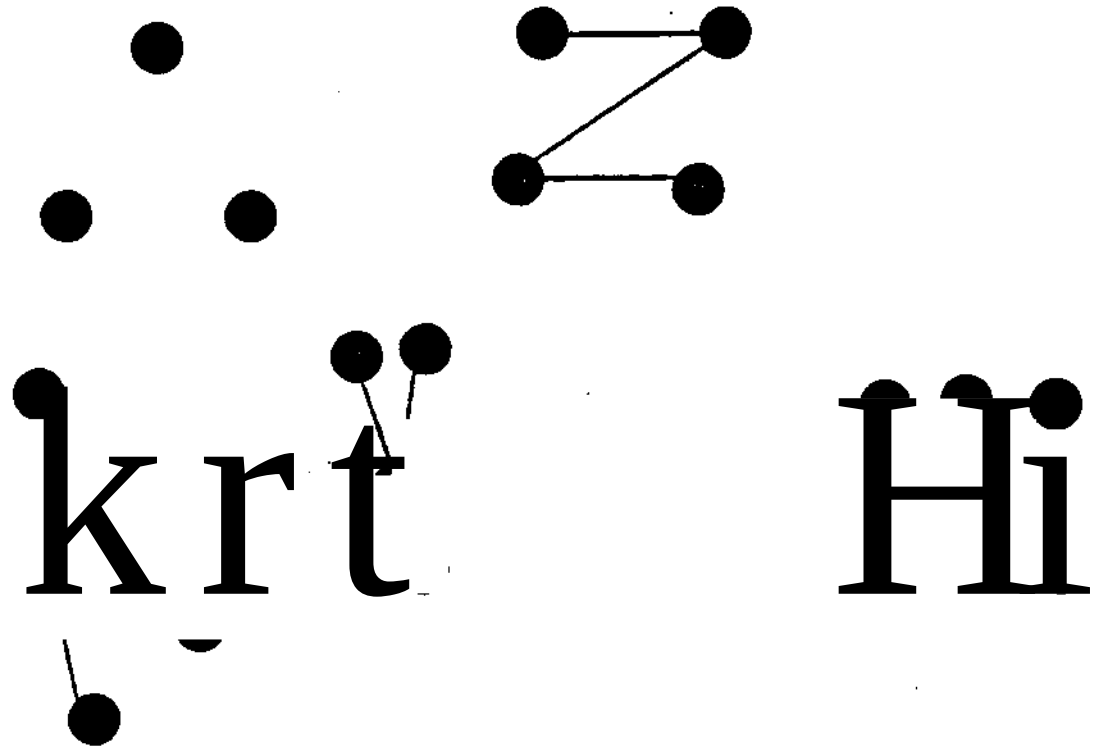


Figure 3-10: Some Acyclic Graphs

The seed set is intended *not* to include <+,$>. Figure 3-11 shows the iterative steps in a sample run of ACYCLIC

Since precisely all graphs of the form <V,£> other than <^>> are identified in $L_1$ by E = 0, and since $B_{xy} = A_{xy} A_x$, we have a floor for acyclic graphs of $\langle P_r L_r 2_1 \rangle$.

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

$B_{x_y}$*(<V,*>) where x e V, y * V

The inverse of ACYCLIC is computed from:

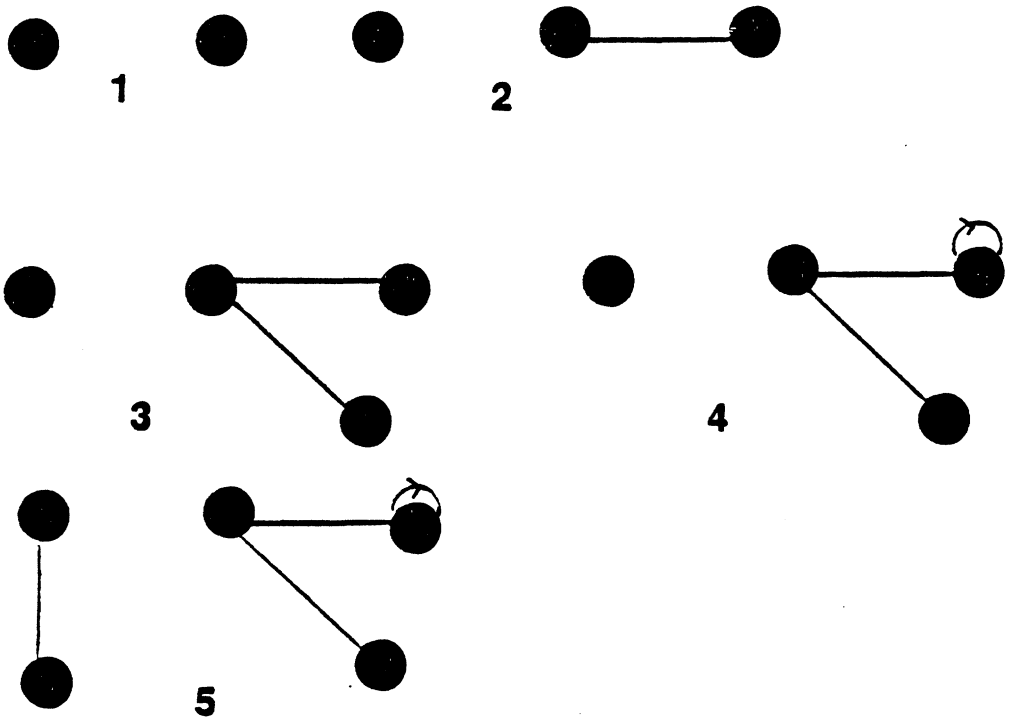



**Figure 3-11:** A Sample Run of ACYCLIC

$$f^{-1} = (B_{xy} + A_{zz})^{-1}$$
$$= B_{xy}^{-1} + A_{zz}^{-1}$$
$$= (A_{xy}A_y)^{-1} + A_{zz}^{-1}$$
$$= A_y^{-1}A_{xy}^{-1} + A_{zz}^{-1}$$
$$= D_y D_{xy} + D_{zz}$$

$$\sigma_{pre} = x \in V,\ y \notin V$$
$$z \in V,\ zz \notin E$$

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

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 $G \notin G_p$. In order to show that an inverse is correct, we must demonstrate that it behaves properly both on $G \in G_p$ and on $G \notin G_p$. 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 $\langle V, \phi \rangle$, with one

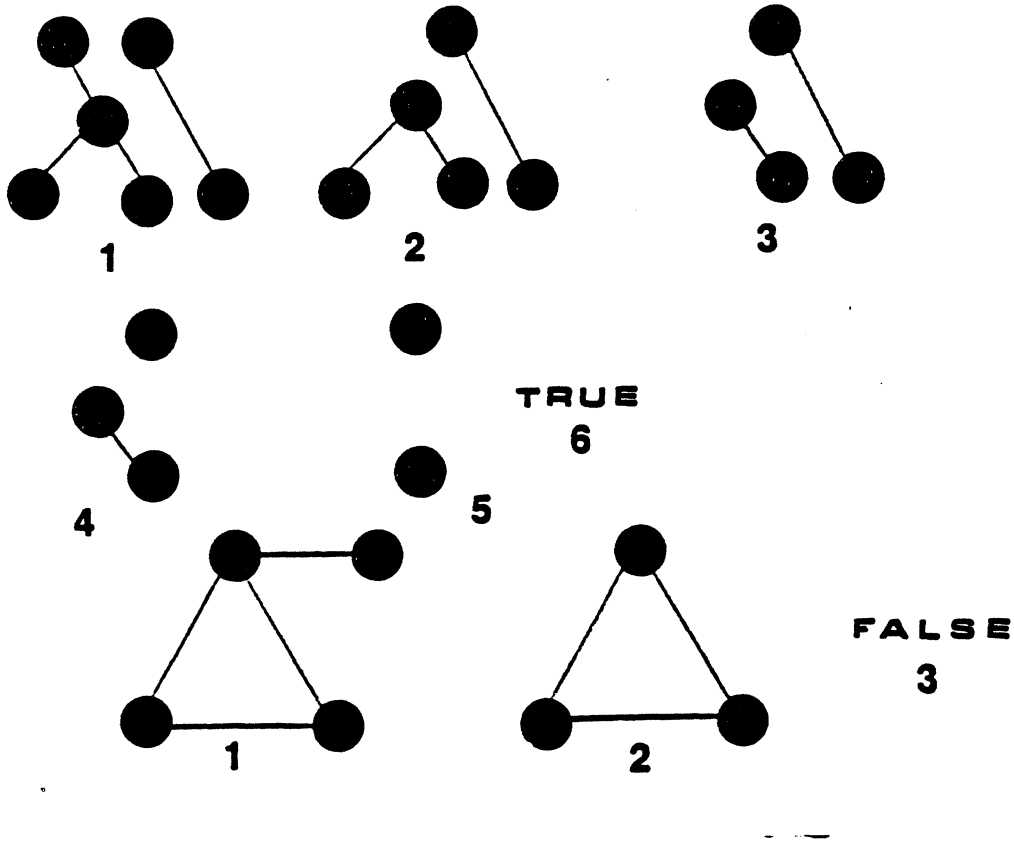



**Figure 3-12:** ACYCLIC$^{-1}$ in Operation

element remaining in V for each mutually accessible set of vertices originally in V. If $G \notin G_p$, $p^{-1}$ 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 $\sigma^{-1}$ 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

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

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

Since the only graph matching the $L_1$ characterization $E \cup \underline{1} = 0$ is $K_1$, the floor for trees is $\langle P_1, L_1, \Sigma_1 \rangle$.

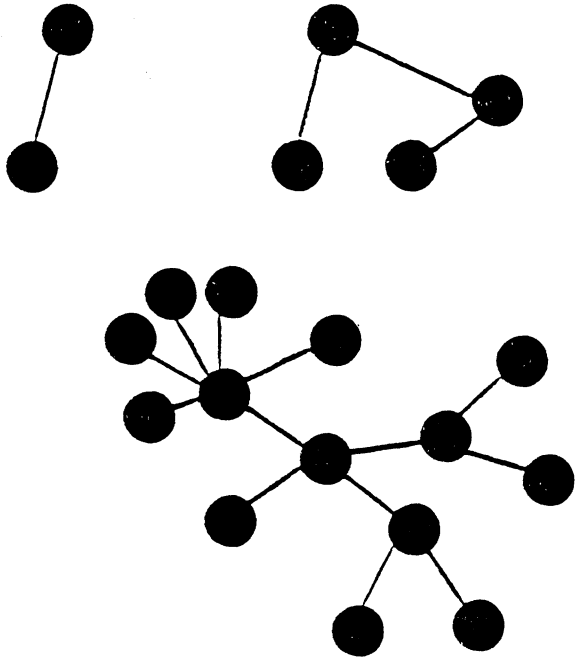


**Figure 3-13:** Some Trees

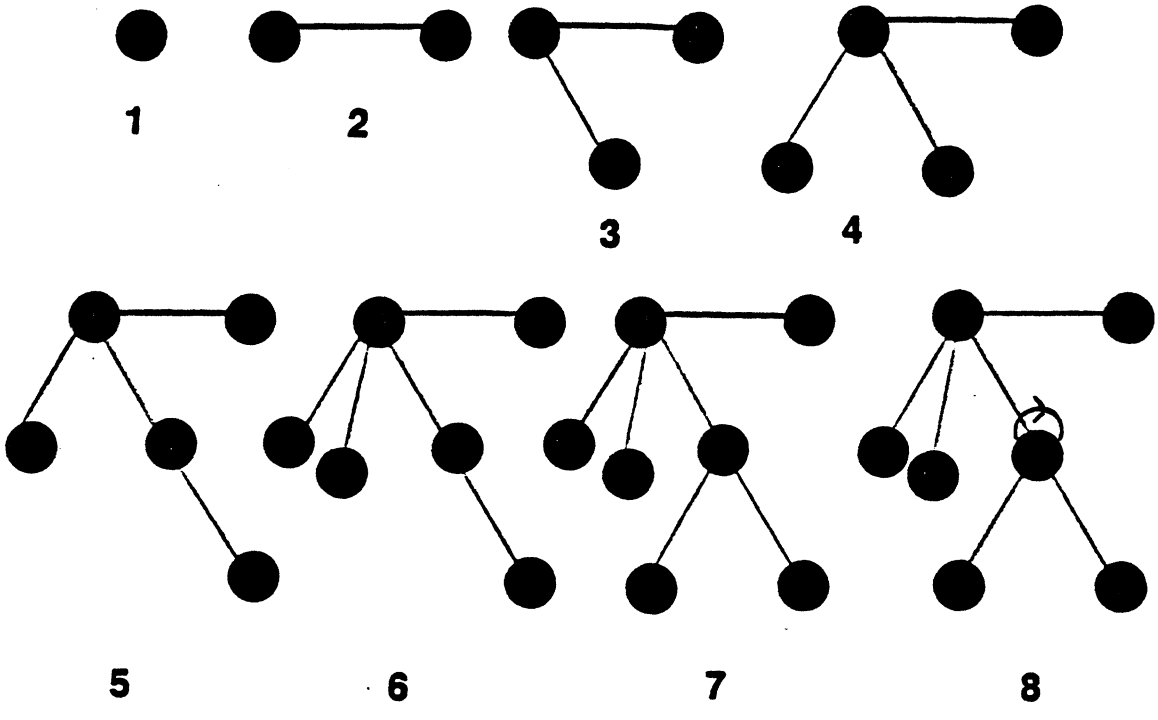



**Figure 3-14:** A Sample Run of TREE

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 $a'''^{1}$ as ACYCLIC, namely

$f\sim^{1} = D_y D_{xy} + D_{22}$

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

$\mathbf{2 \in V,\ Z2 \in E}$

Again there is a shift in the floor to $<P_r L_r E_3>$. 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 $G * G_p$.

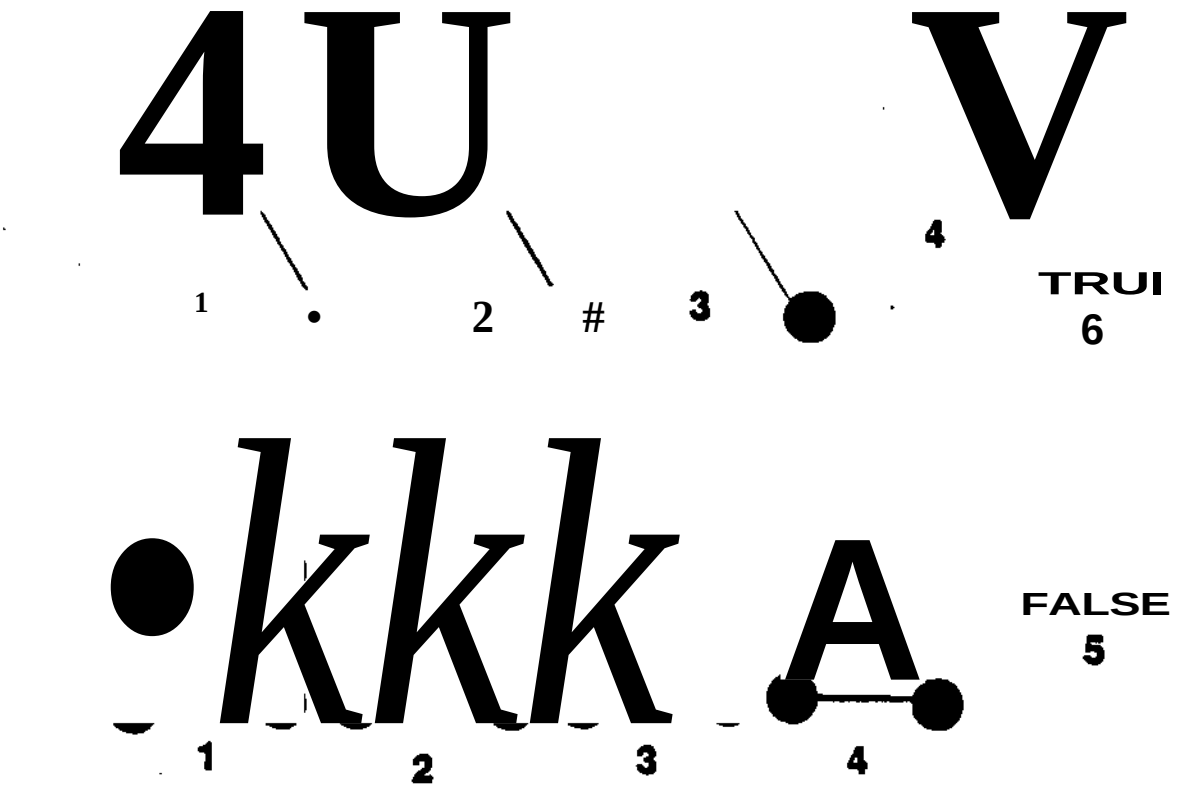



**Figure 3-15:** $TREE''^{1}$ in Operation

$TREE''^{1}$ reduces any tree to, ultimately, a single, isolated vertex isomorphic to $K_r$ $TREE'''^{1}$ on a graph which is not a tree will remove all tree-like protuberances and then $a''^{1}$ 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.

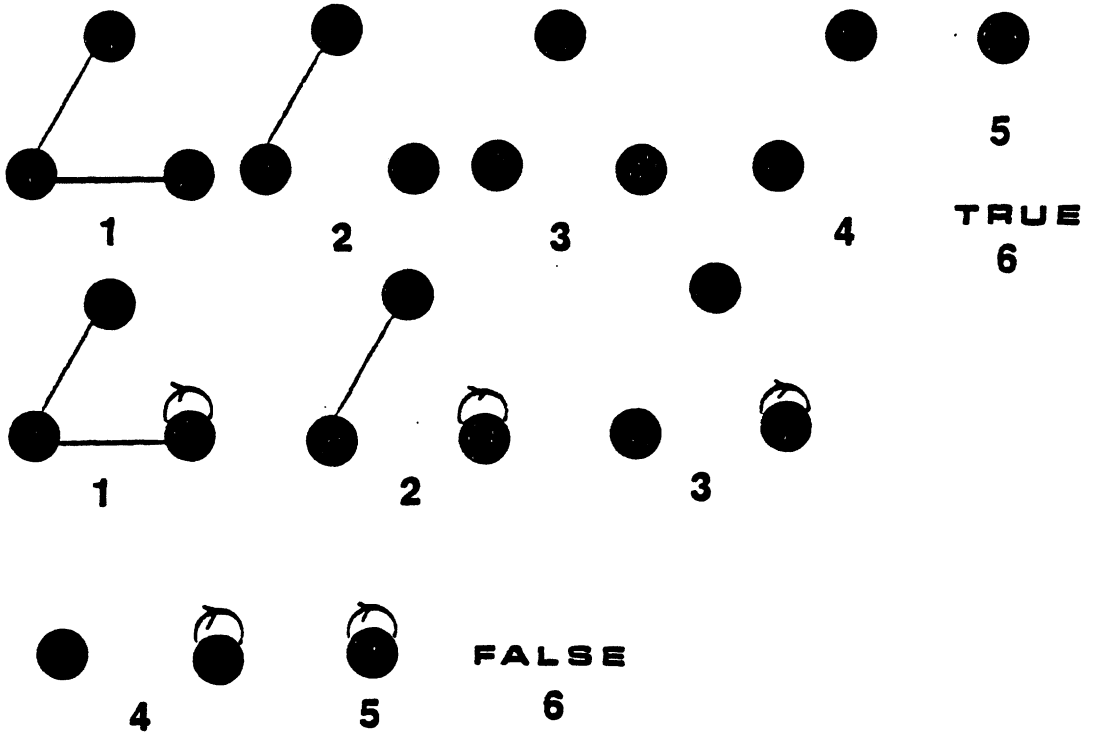



**Figure 3-18:** LOOPFREE$^{-1}$ in Operation

### 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.

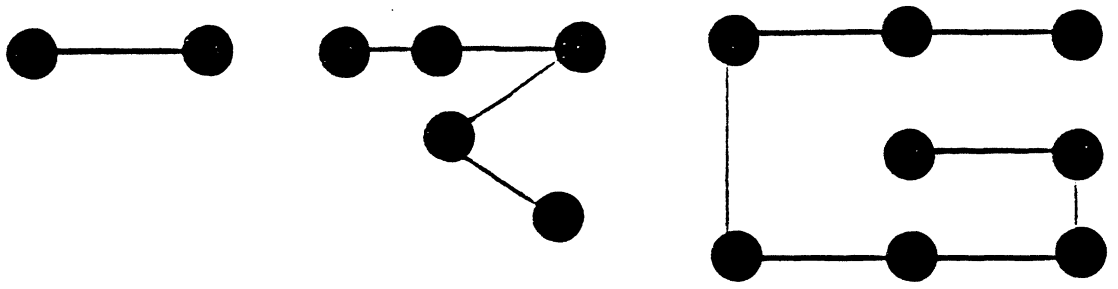


**Figure 3-19:** Some Chains

The R-property CHAIN is

$S^{*}_{xvy}(K_2)$ where $x,y \in V$, $v \notin V$, $xy \in E$

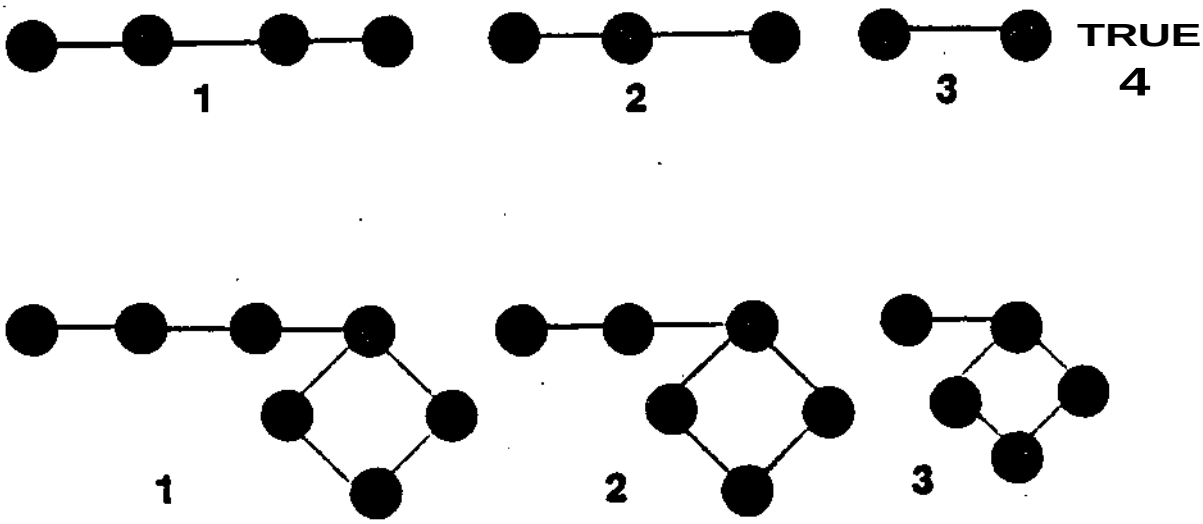



**Figure 3-21:** CHAIN$^{-1}$ in Operation

$= D_y D_{xy}$

$\sigma_{pre}$ $\quad a\ X \in V, y\ O\ V, d(X) > a\ 1$

$\sigma^{-1}$ $\quad \ll x,y\ e\ V, xy\ e\ E, d(y) = 1, d(x) = 2$

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

$CHAIN_2^{*1}$ 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^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.

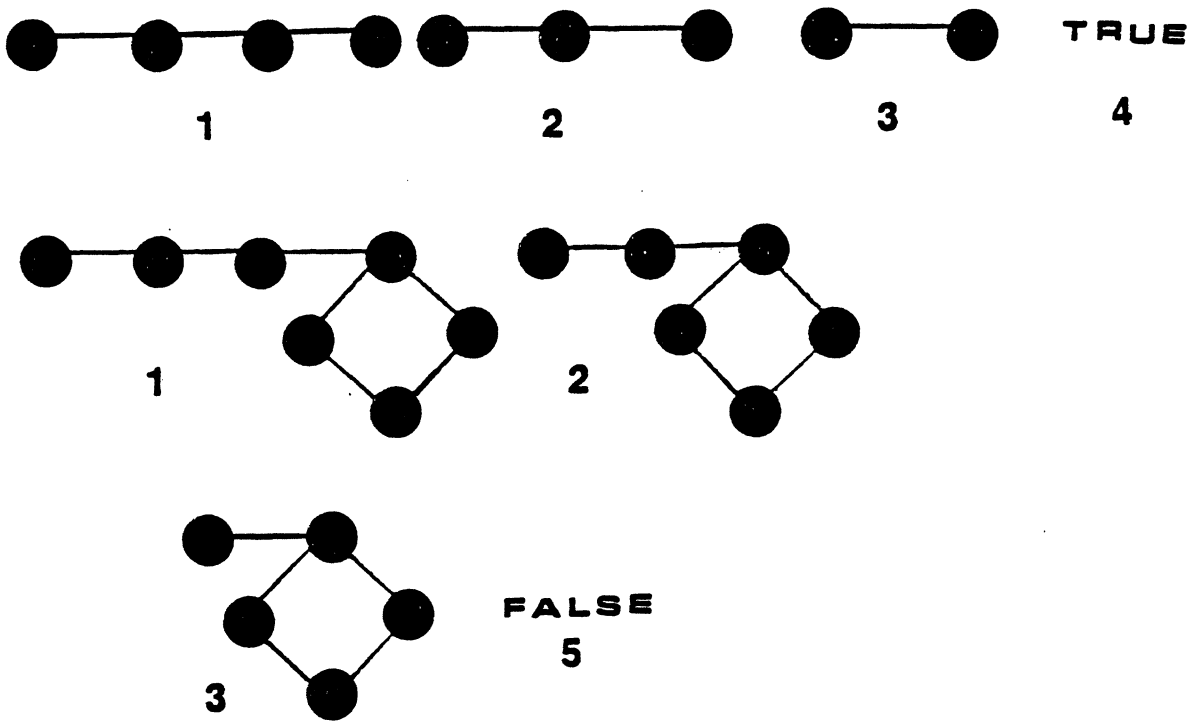



**Figure 3-22:** $CHAIN_2^{-1}$ in Operation

### 3.7.5. Cycles

Several examples of cycles appear in Figure 3-23.

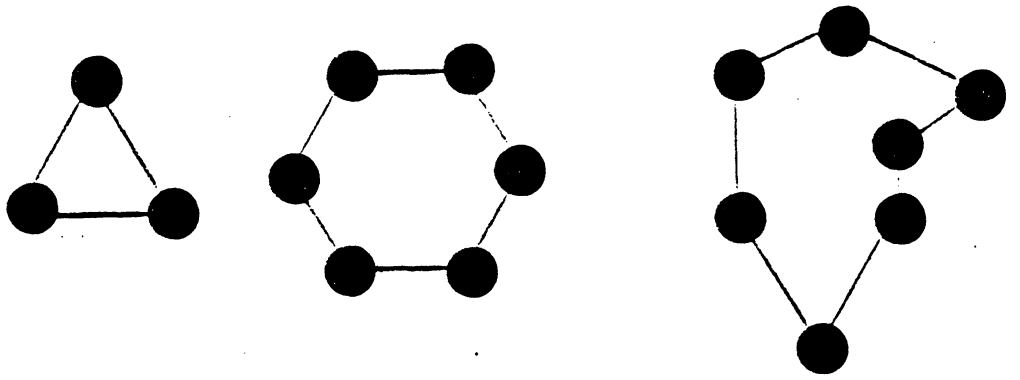


**Figure 3-23:** Some Cycles

The R-property CYCLE is

$S^*_{xvy}(K_3)$ where $x,y \in V$, $v \notin V$, $xy \in E$.

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

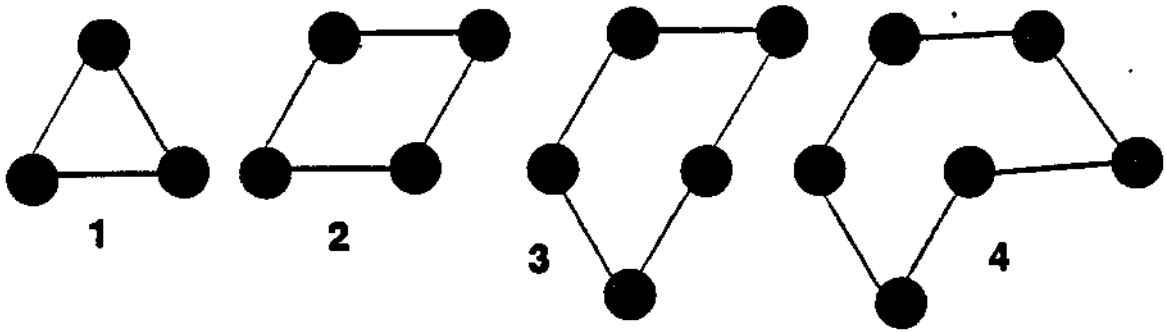



Figure 3-24: A Sample Run of CYCLE

In $L_1$ the characterization of $K_3$ is the same as $K_{2>}$ In $L_2$. however, $\{K_3\}$ is uniquely defined by:

$\underset{\sim}{E} \cap \underset{\sim}{1} = 0$

$E \cap 1 = o$

$|E \cap \underset{\sim}{i}| = |\underset{\sim}{E} \cap 1|$

It is also possible to reach $K_3$ in $L_{1n}$ as $\underset{\sim}{E} \cap j = 0$, $E \cap 1 = 0$ and $n = 3$. Thus the floors of CYCLE are $<P_2 X_r E_1>$ and $<P_2, L_{jn} Z,>$.

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 $a"^{*1}$ as CHAIN, namely

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

$a"^1 = x,y,v \in V,\ xv,\ yv \in E,\ xy \notin E,\ d(v) = 2$

Again, this is not accidental. There is a shift in the floors to $<P_2, L_2, ^3>$ and ^*ihyt*^^ Figure 3-25 shows $CYCLE"^1$ operating on a graph $G \in G_p$ and a graph $G \notin G_p$.

$CYCLE""^1$ will contract any simple cycle until it is isomorphic to $«_3$. 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.

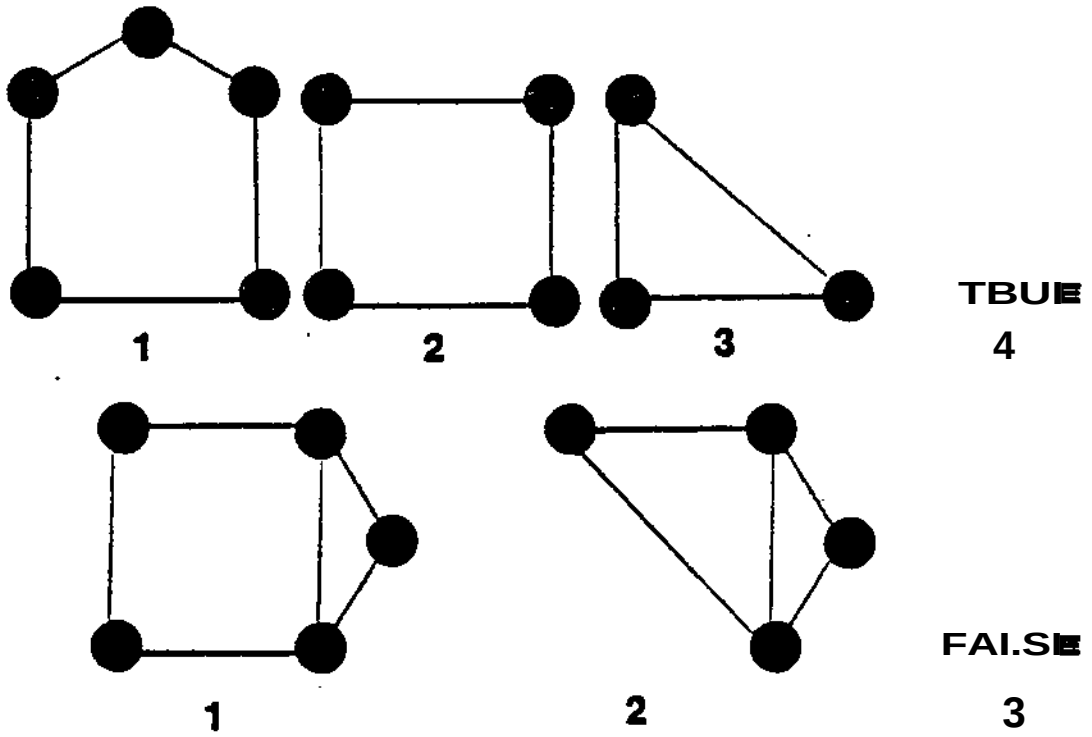



**Figure 3-25:** CYCLE"[1] in Operation

### 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 = Vj X $V_2$ then G is a *complete bipartite* graph. If $|V_1|$ = a, $|V_2|$ = b, the complete bipartite graph on V = $V_1$ u $V_2$ where $V_1$ n $V_2$ = +, is denoted $K_{a,b}$. A *star* is a complete bipartite graph $K_{1,n}$ for n ≥ 3. Several examples of stars appear in Figure 3-26. The R-property STAR is

$B^{*}_{x,y}(K_{1,3}^{*}$ where x e V, y 0 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 n i = 0, but so *are* all loopfree graphs. In $L_2$, $K_{1,3}$ is characteri2ed by E n 1 = 0 and |E| = |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 $<P_1, L_Q, E_4>$. It is

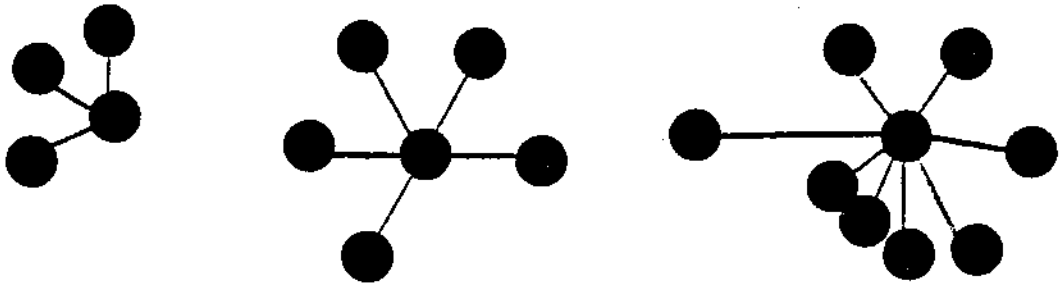


**Figure 3-26:** Some Star Graphs

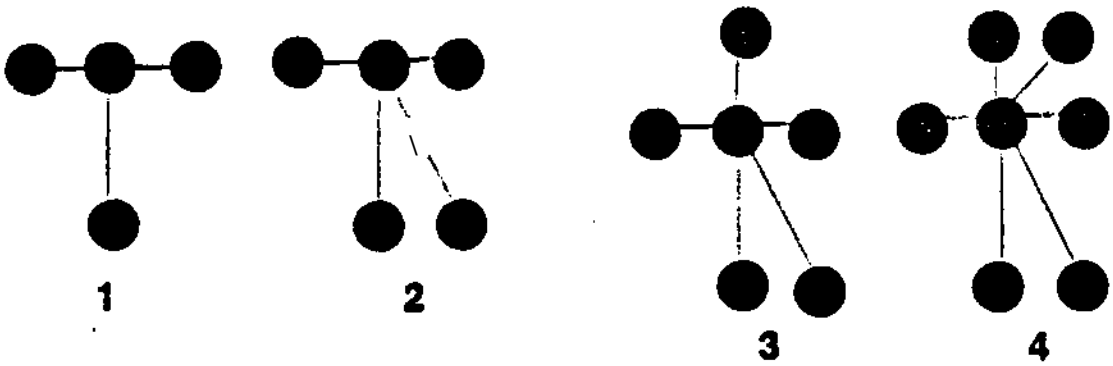



**Figure** 3-27: A Sample Run of STAR

possible to define star so that $K_{1,2}$ or $K_{1,1}$ is considered a star. This would require some *a* 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 E postulated thus far. This is an example of the potential tradeoff between L and *L*

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:

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

$$= DD_{xy}$$

$$\sigma_{pre} = x^{y} \text{« } \mathbf{v, y} \text{ « } V,\ d(x) = \max$$

$$a''^{1} = x,y \leq V,\ d(x) \Rightarrow \max.\ d(y) = 1$$

The floor shifts to $^{<P_2/L}Q'^{\wedge}4^{>}$- $^{pi}9^{ure}$ 3-28 shows STAR$''''^{1}$ operating on a

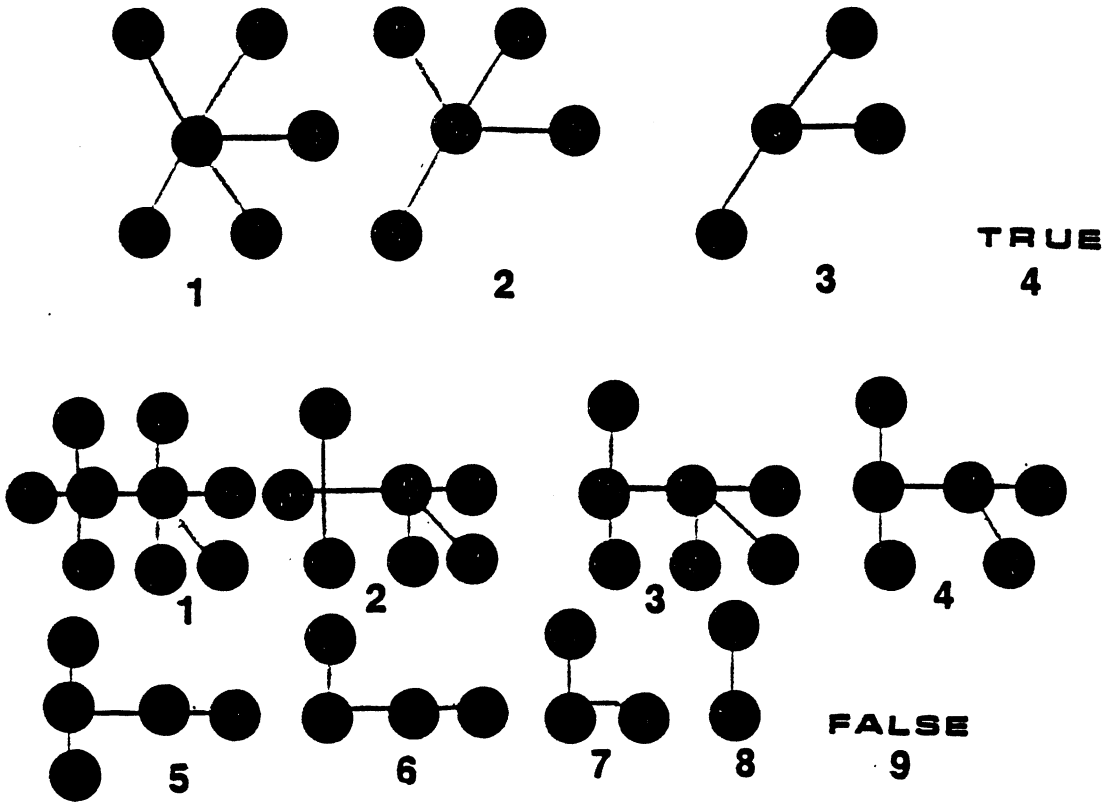



**Figure 3-28:** $STAR^{-1}$ in Operation

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

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

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

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

A wheel is composed of a *rim* ($C_{v_1 v_2 \cdots v_n}$) 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

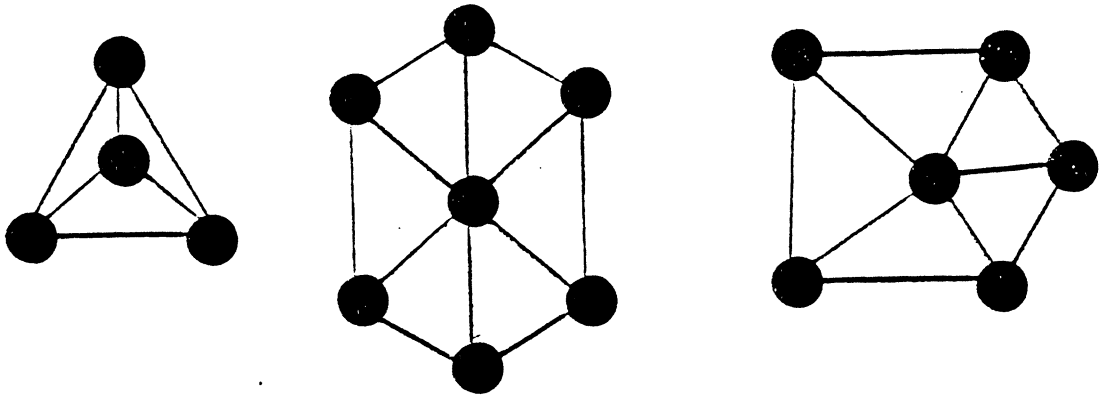


**Figure 3-29:** Some Wheels

$(A_{zv}S_{xvy})^*(K_4)$ where distinct x,y,z ∈ V, v∉ V, xy ∈ 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.

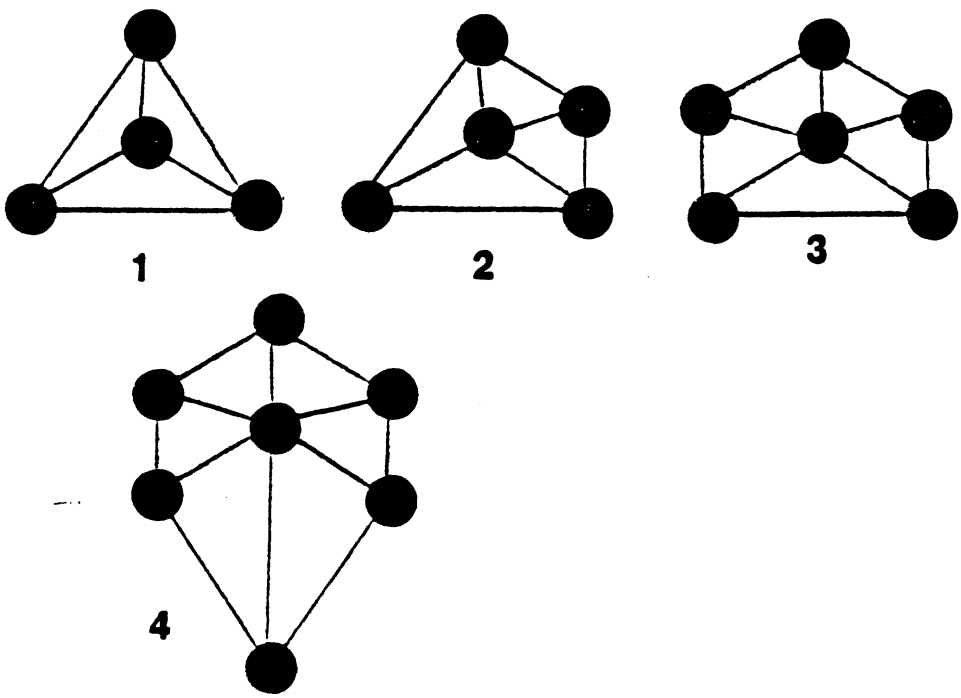



**Figure 3-30:** A Sample Run of WHEEL

We use "distinct x,y,z" here as an abbreviation for the $\Sigma_2$ notation "x ≠ y, x ≠ z, y ≠ 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:

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

$E \cap 1 = 0$

$n = 4$

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:

$f^{-1} = (A_{zv} S_{xvy})^{-1}$

$= S_{xvy}^{-1} A_{zv}^{-1}$

$= D_v D_{vy} D_{xv} A_{xy} D_{zv}$

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

$\sigma^{-1}$ = distinct $v,x,y,z \in V$, $xv, vy \in E$, $xy \notin E$, $d(z) = max$, $d(v) = 3$

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

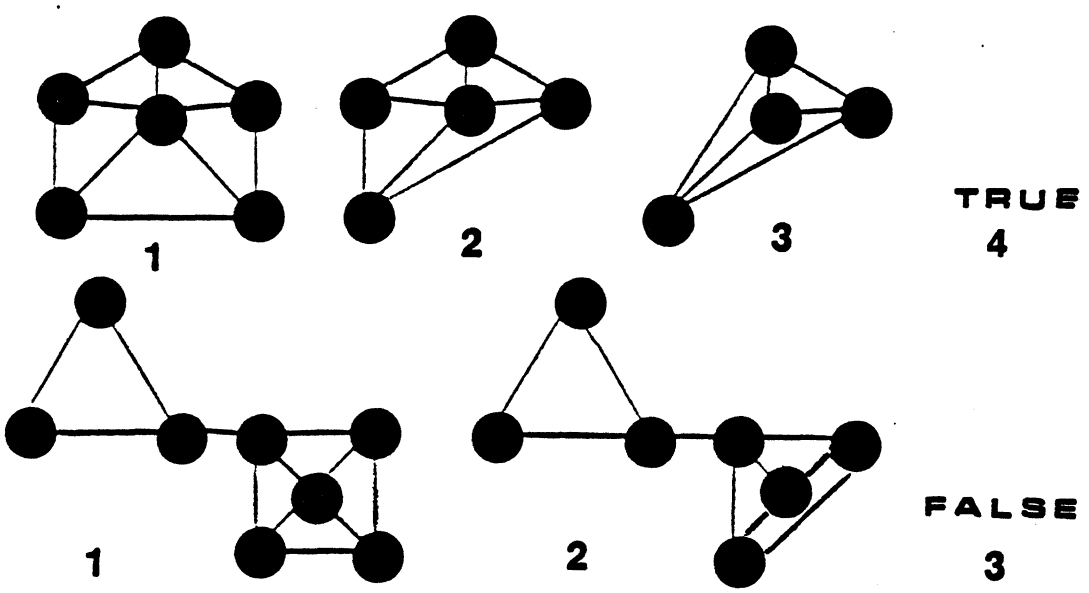



**Figure 3-31:** $WHEEL^{-1}$ in Operation

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.

### 3.7.3. Complete Graphs

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

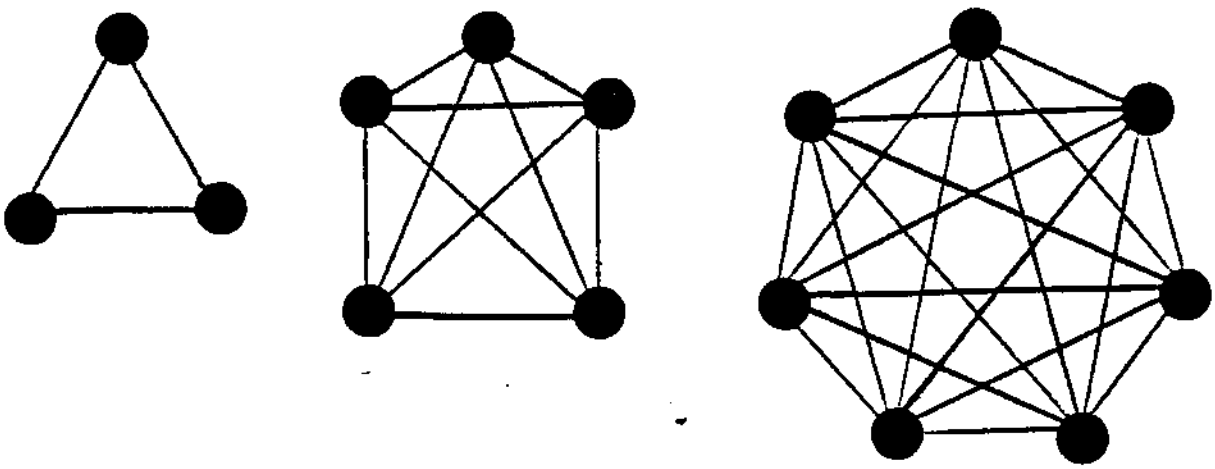


**Figure 3-32:** Some Complete Graphs

The R-property COMPLETE is

FJK,) where x * V, distinct $v_r v_{2W} v_n$ * V, |{$v_r v_r, v_n$}| « |V|

Figure 3-33 shows the iterative steps in a sample run of COMPLETE Since $F_x$ = $A_{xv_1} A_{xv_2} \cdots A_{xv_n} A_{X'}$ the floor for complete graphs is <$P_r L_r Z_5$>.

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

The inverse is computed by:

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

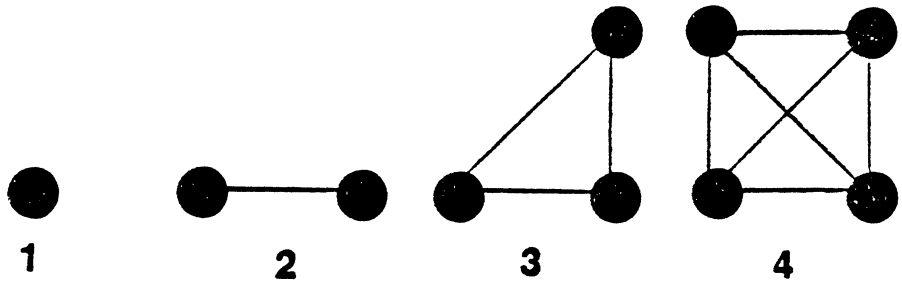



**Figure 3-33:** A Sample Run of COMPLETE

$$= (A_{xv_1} A_{xv_2} \ldots A_{xv_n} A_x)^{-1}$$

$$= A_x^{-1} A_{xv_n}^{-1} A_{xv_{n-1}}^{-1} \ldots A_{xv_1}^{-1}$$

$$= D_x D_{xv_n} D_{xv_{n-1}} \ldots D_{xv_1}$$

$\sigma_{pre} = \sigma$ $= x \notin V$, distinct $v_1, v_2, \ldots, v_n \in V$, $|\{v_1, v_2, \ldots, v_n\}| = |V|$

$\sigma^{-1}$ $=$ distinct $x, v_1, v_2, \ldots, v_n \in V$, $|\{x, v_1, v_2, \ldots, v_n\}| = |V|$, $d(x) = n$

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 $G \notin G_p$.

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

$(A_{xy} + A_w A_z)^*(E_2)$ where $x, y \in V$,

$w, z \notin V$, $w \neq z$

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 $\underline{E} \cap \underline{1} < \underline{E} \cap 1$. These properties imply $n(n-1)/2 < n$ which requires $n < 3$. Since

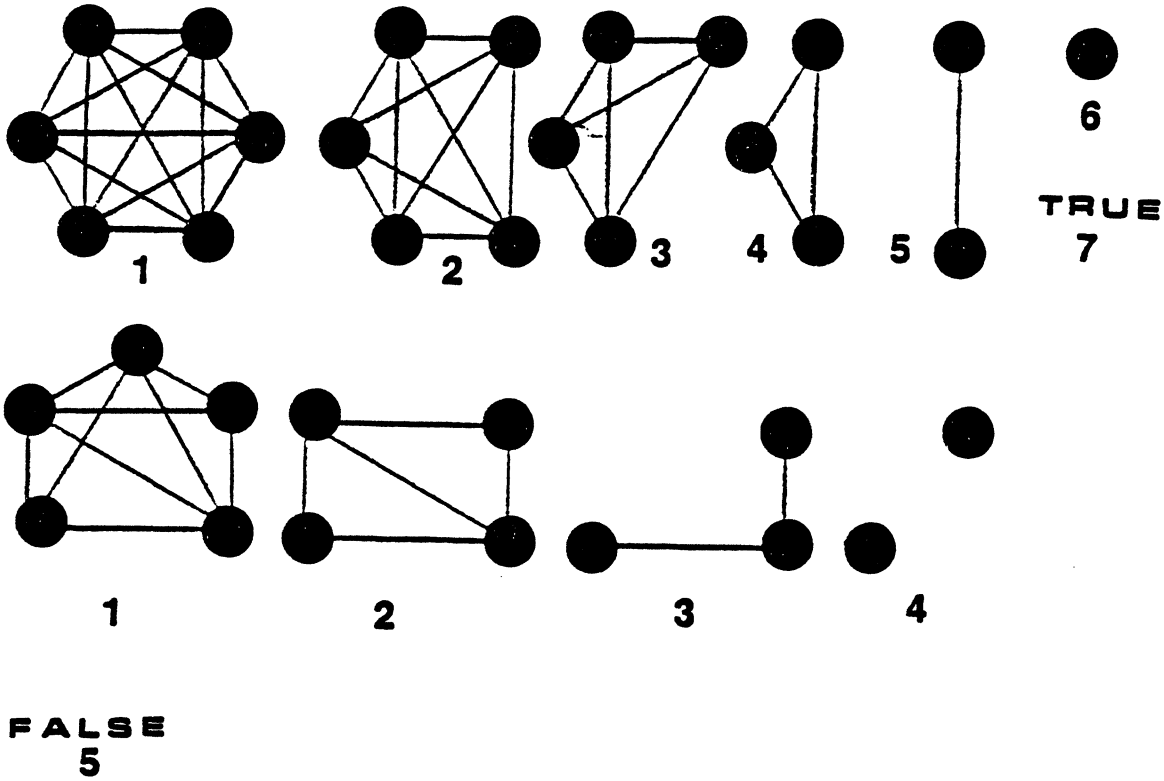



**Figure 3-34:** COMPLETE$^{-1}$ in Operation

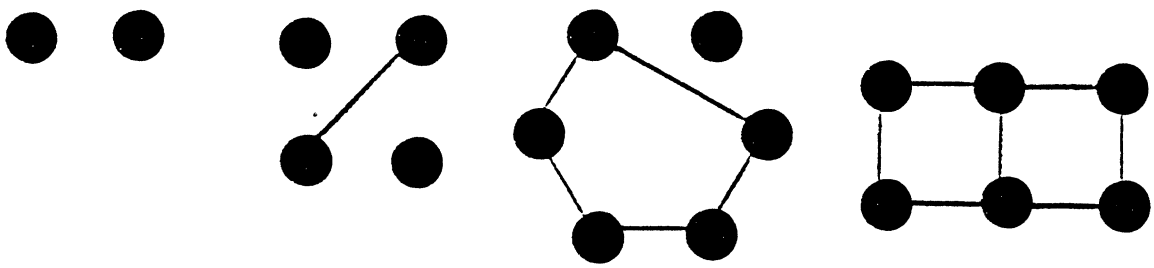


**Figure 3-35:** Some Graphs with an Even Number of Vertices

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.

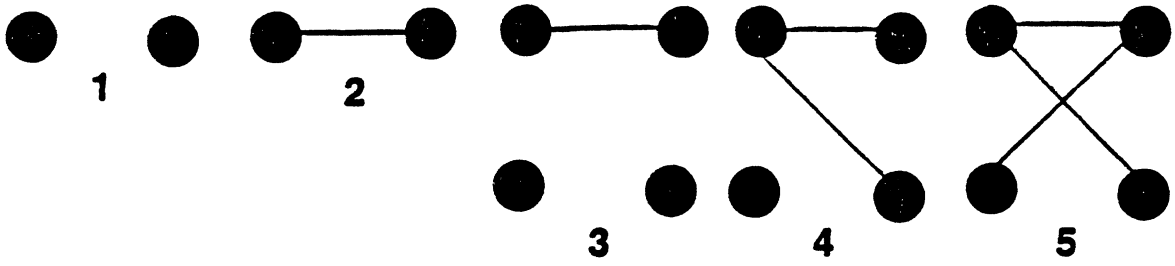



**Figure 3-36:** A Sample Run of EVEN-N

The inverse is computed by:

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

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.

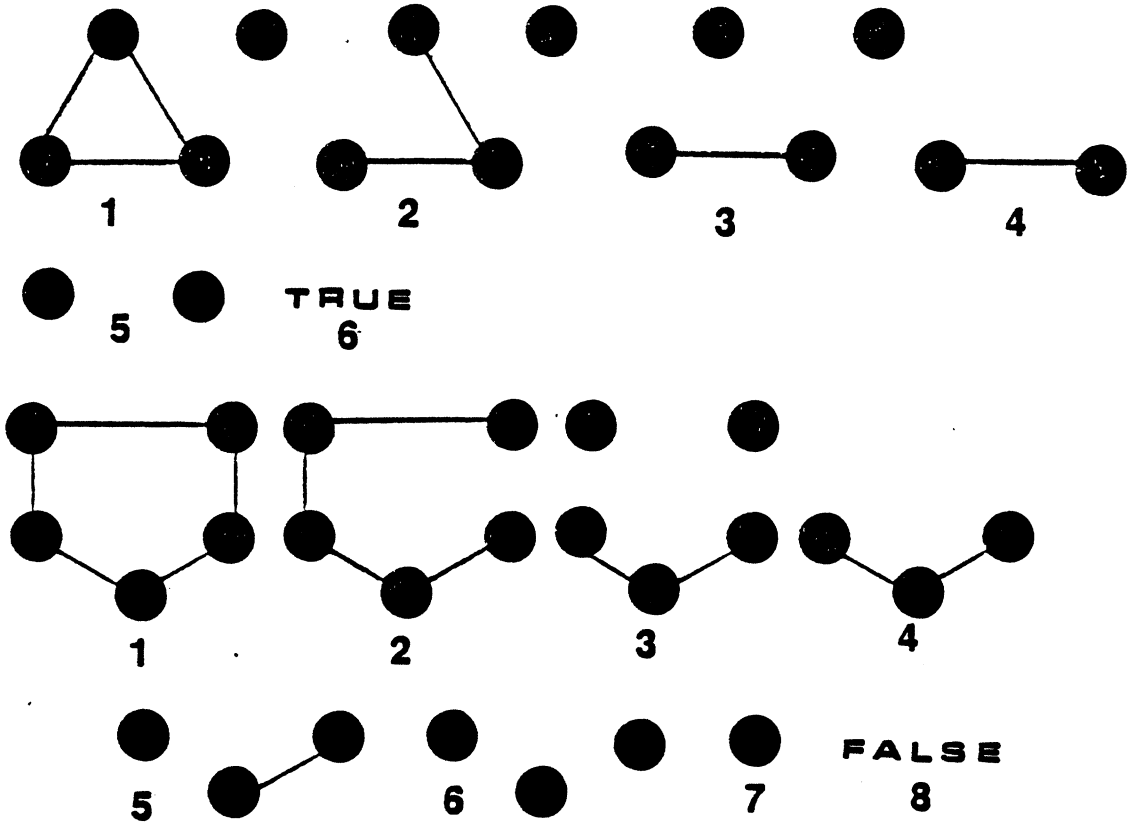



**Figure 3-37:** EVEN-$N^{-1}$ in Operation

### 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.

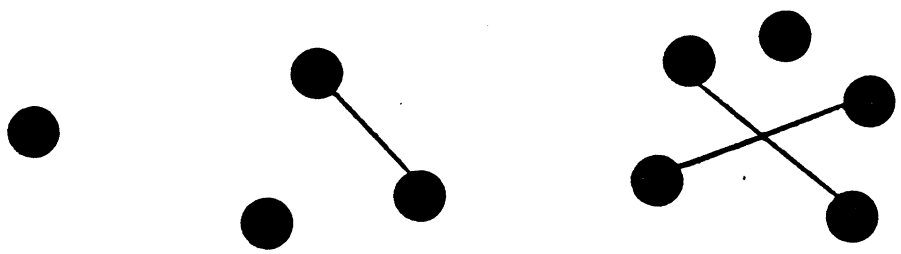


**Figure 3-38:** Some Graphs with an Odd Number of Vertices

The R-property ODD-N is

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

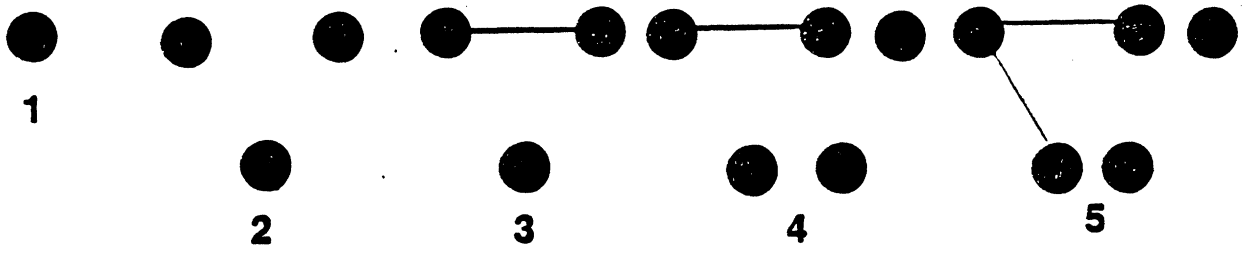



**Figure 3-39:** A Sample Run of ODD-N

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 $\sigma^{-1}$ as those for EVEN-N, namely,

$$f^{-1} = D_{xy} + D_z D_w$$

$$\sigma^{-1} = x,y \in V,\ xy \in E$$
$$w,z \in V,\ w \neq z,\ d(w) = 0,\ d(z) = 0$$

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 $G \notin G_p$. 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 $\langle \phi, \phi \rangle$ 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

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

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 $\langle P_1, L_1, \Sigma_2 \rangle$.

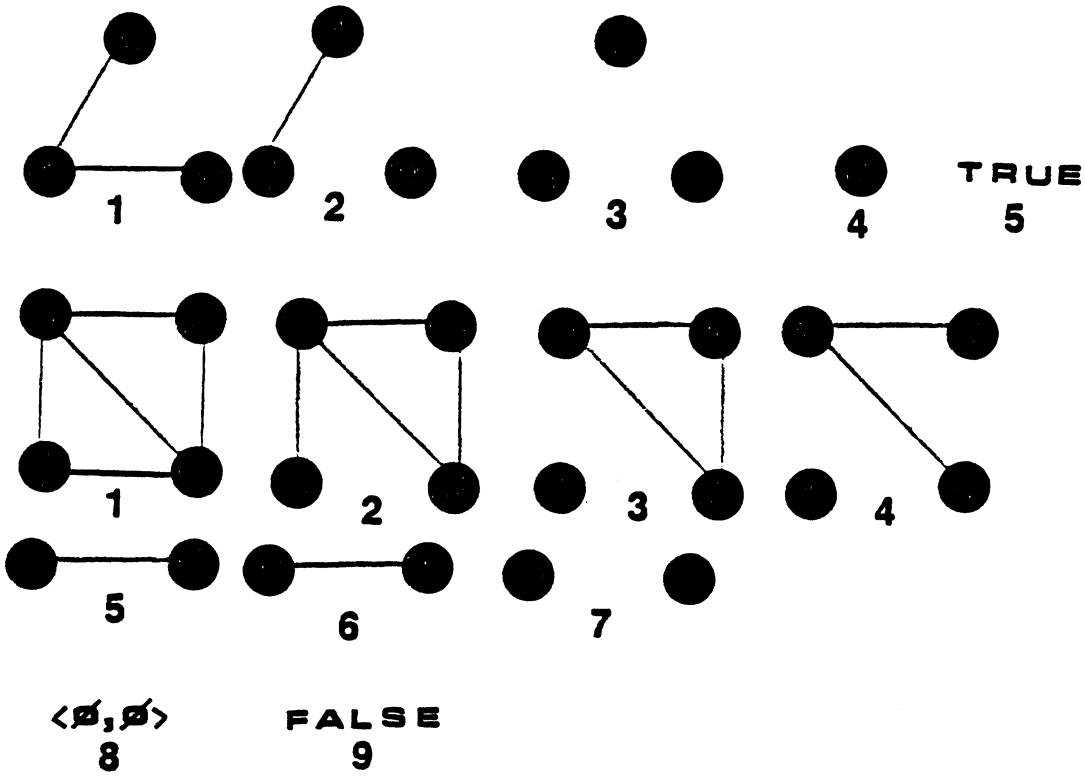



**Figure 3-40:** ODD-$N^{-1}$ in Operation

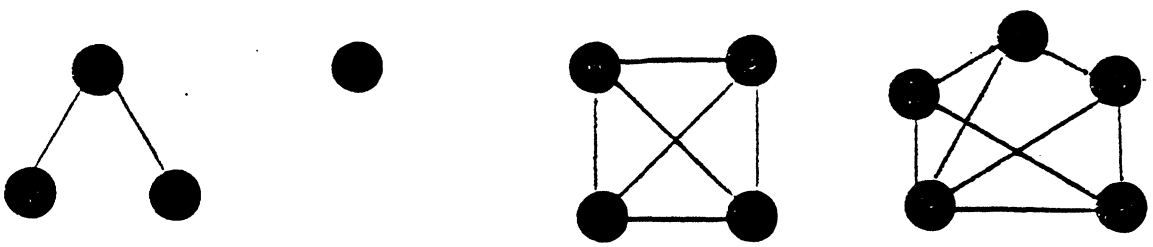


**Figure 3-41:** Some Graphs with an Even Number of Edges

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

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

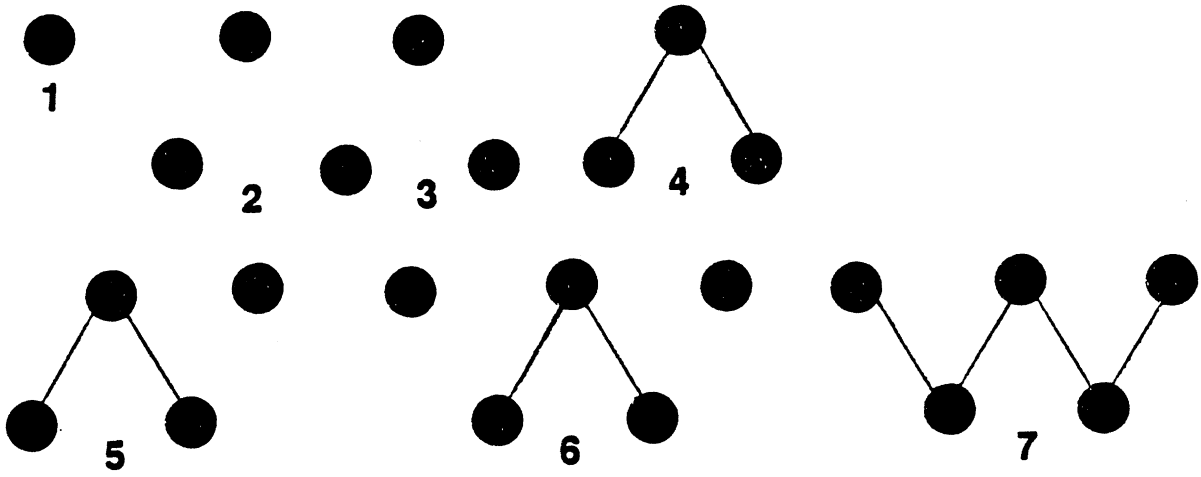



**Figure 3-42:** A Sample Run of EVEN-M

$\sigma_{pre}$ $= x \notin V$

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

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

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

The floor shifts to $<P_2,L_1,\Sigma_3>$. Figure 3-43 shows EVEN-$M^{-1}$ operating on a graph $G \in G_p$ and a graph $G \notin G_p$. 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

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

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 $<P_1,L_{1n},\Sigma_2>$ and $<P_1,L_3,\Sigma_2>$.

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 $\sigma^{-1}$ as those for EVEN-M,

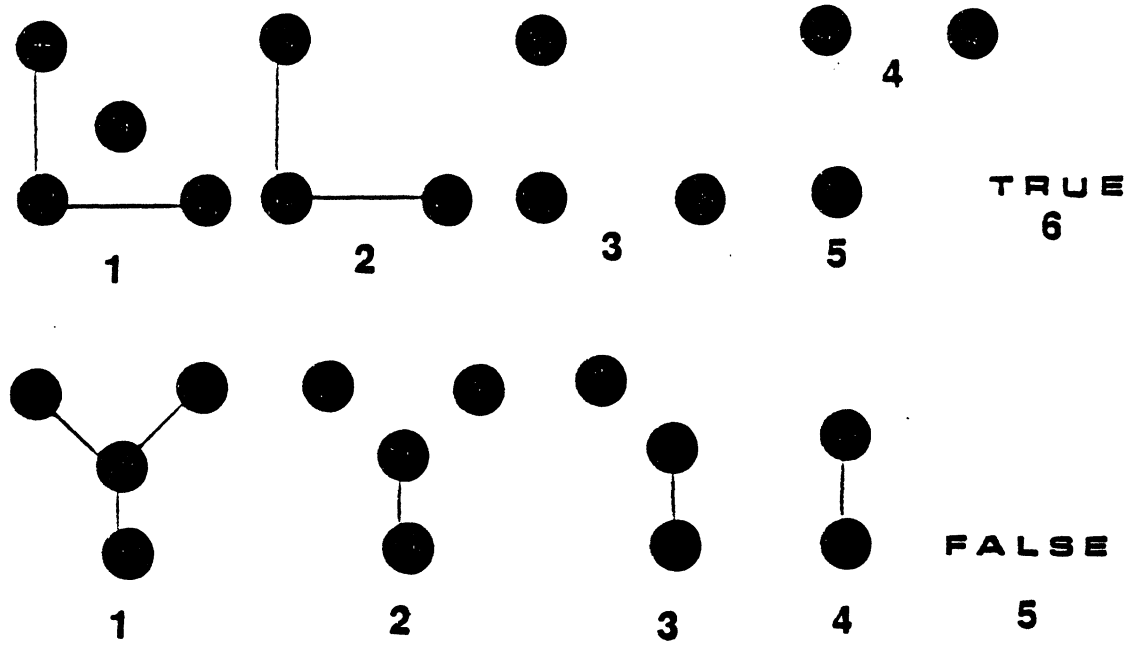



**Figure 3-43:** EVEN-$M^{-1}$ in Operation

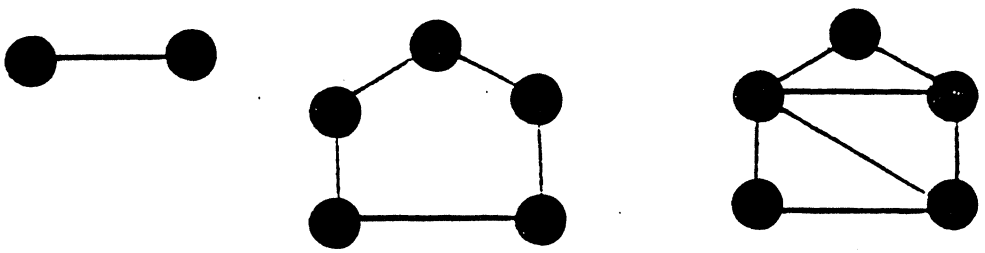


**Figure 3-44:** Some Graphs with an Odd Number of Edges

namely

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

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

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

The floors shift to $\langle P_2, L_{1n}, \Sigma_3 \rangle$ 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 $G \notin G_p$. 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 $\langle \phi, \phi \rangle$ and then fail. Thus ODD-$M^{-1}$ is correct and ODD-M is complete.

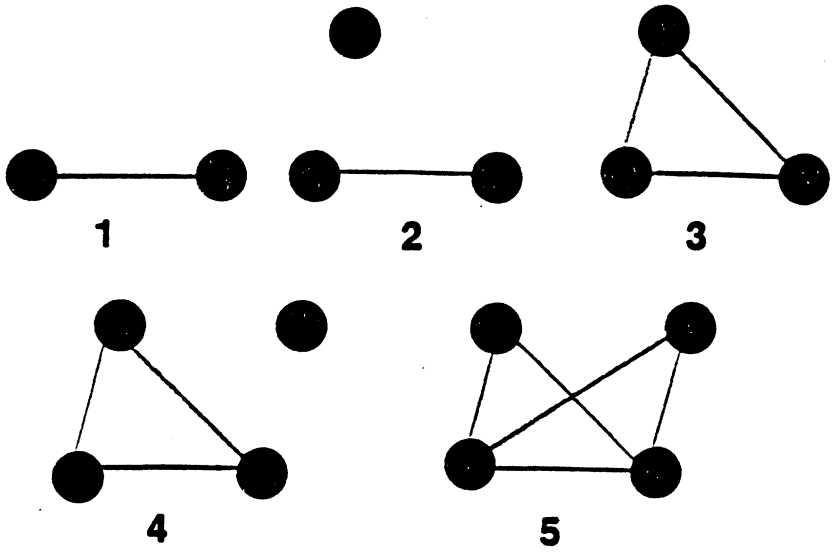



**Figure 3-45:** A Sample Run of ODD−M

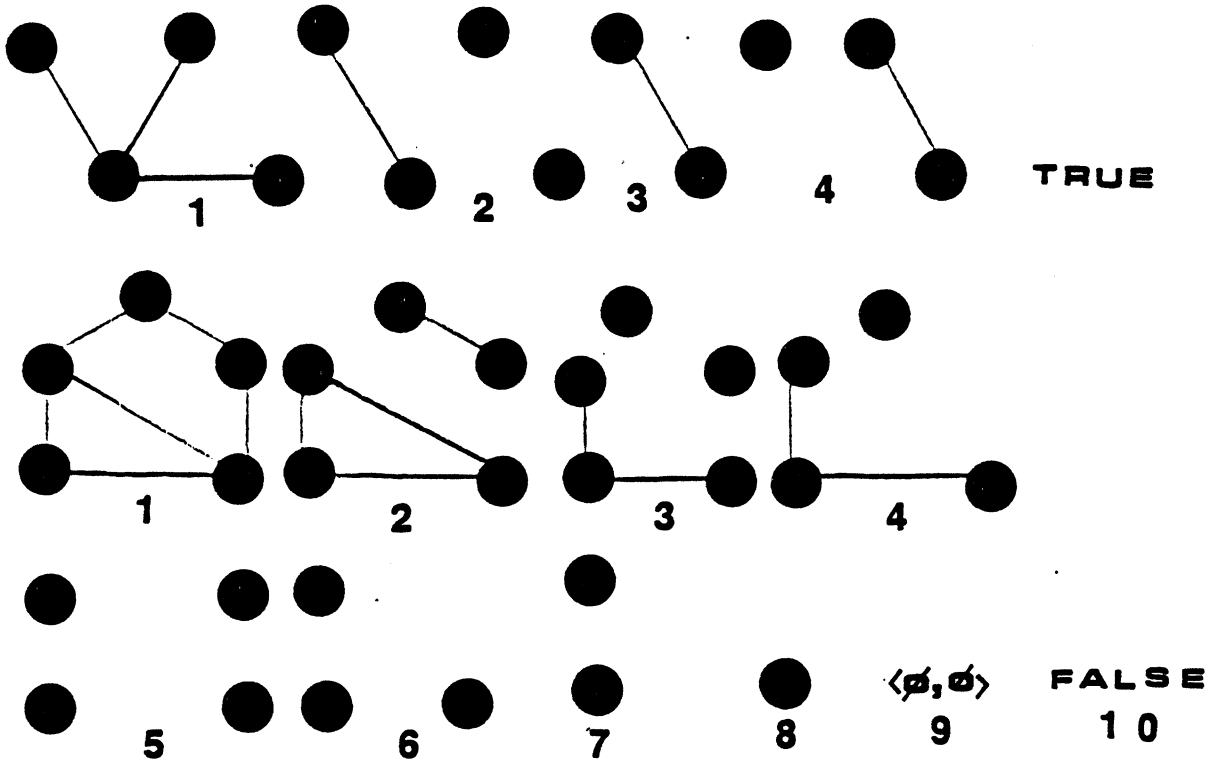



**Figure 3-46:** ODD−$M^{-1}$ in Operation

### 3.7.13. Euiertan 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.

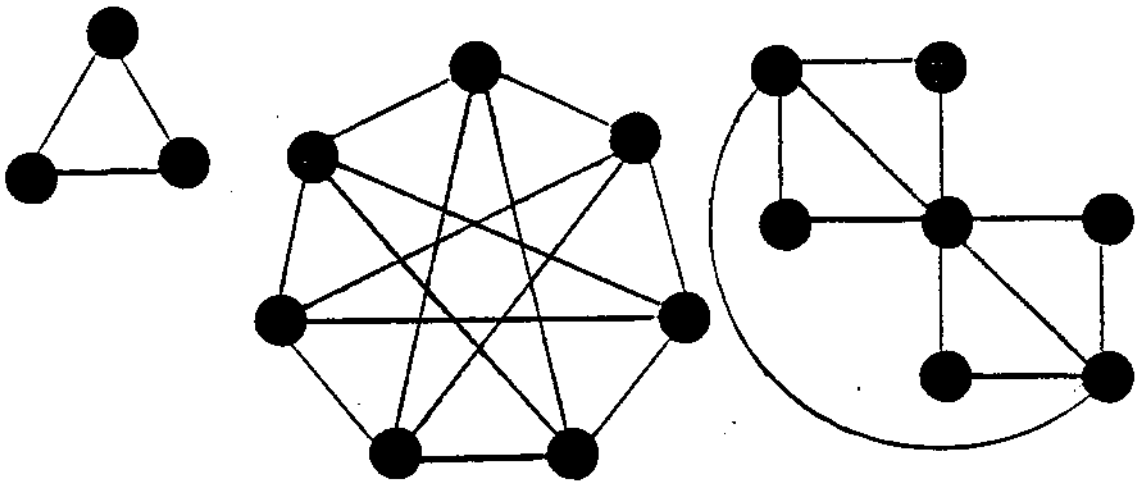


Figure 3-47: Some Eulerian Graphs

The R-property EULERIAN is

$$(S_{wvz} + Y_{v_1 v_2 \cdots v_k})^*(KJ \text{ where } w,z \cdot V,\ v * V,\ wz \cdot E,$$
$$|\{v_1 v_2 \cdots v_k\} \cap V| \geq 1,$$
$$v^{\wedge}{}_k * E,\ v_i v_{i+1} * E,\ i = 1,2 \_ k - 1,\ \text{distinct } v_i,\ i = 1,2,\ldots k \leq 3$$

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

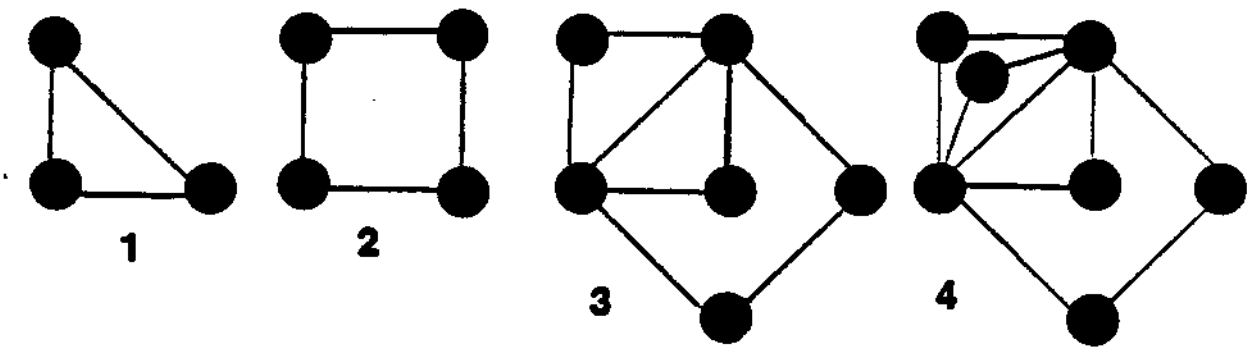



Figure 3-48: A Sample Run of EULERIAN

The algorithm either subdivides an existing edge or appends a cycle $C_{v_i y_r v_k}$. This

cycle must have at least one vertex and no edges in common with the current graph. Since $Y_{v_1v_2'''\backslash} = A_{v_1v_2} A_{v_1v_2} \ldots A_{v_i \backslash v_k} A_{v_1v_2} A_{v_1v_2} A_{v_k} \ldots A_{v_1v_2}$ and $K_3$ is describable in $L_{1n}$ or $L_2$, the floors for Eulerian graphs are $\langle P_2, L_{in}, 5 \rangle$ and $\langle P_2, L_2, 5 \rangle$.

Given a graph G which is Eulerian, there is an Euierian 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 $v_i$ s V occurred in the Eulerian walk, we substitute the cycle

$$v_i, v_i v_{i+1}, v_{i+1}, \ldots, v_n, v_n v_1, v_1, v_1 v_2, v_2, \ldots, v_i$$

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:

$r^1$ a is $+ Y_{wvz} r^1_{v_i v_{rt}}$

$* S_{wvz}^{-1} + Y_{v_1 v_2 * " v_k}^{"1}$

$= S_{wvz}^{"1} + Y_{"\wedge\wedge"\wedge ic}$

$= D_v D_{vy} D_{xv} A_{xy} + Y_{\sim"^v i^v 2^{\#} \sim^v i_c}$

$a_{pre} = a$ = w,z 6 V, wz e E

$i\{v_i V^v k\} n v_l - \geq 1 - V_k * E -$

$v_t v_{i-hl}$ 0 E, i= 1,2,-.,k-1, distinct $v_i$, i « 1,2,.~,k,

k > 3

$a^{*1}$ = distinct v,w,z € V, wv,vz s E, wz * E, d(v) = 2

distinct $v_{1\#} v_2 \wedge_t v_k$ e V,

$v_t v_{j-M}$ s E , i = 1,2,w,k-1, $v_t v_k$ s E, k > 3,

not all $^v_r v_2, \ldots v_k$ of degree s 2

The floors shift to the less powerful $\langle P\wedge X, , Z\wedge \rangle$ and $\langle P_n, L\sim, Z\sim \rangle$- Figure 3-49 shows EULERIAN$\wedge^1$ operating on a graph G s $G^p$ and a graph G * $G^p$. Note our first application of tentative deletion (Rule 4) in $f\sim^1$ via $\overline{Y}^{v_1 v_2 \cdots v_k}$. It is not clear whether all (or even any) of the vertices $^v_r v_2, \ldots, v_k$ were present before the cycle was added After the edges of the cycle are deleted however, those vertices of

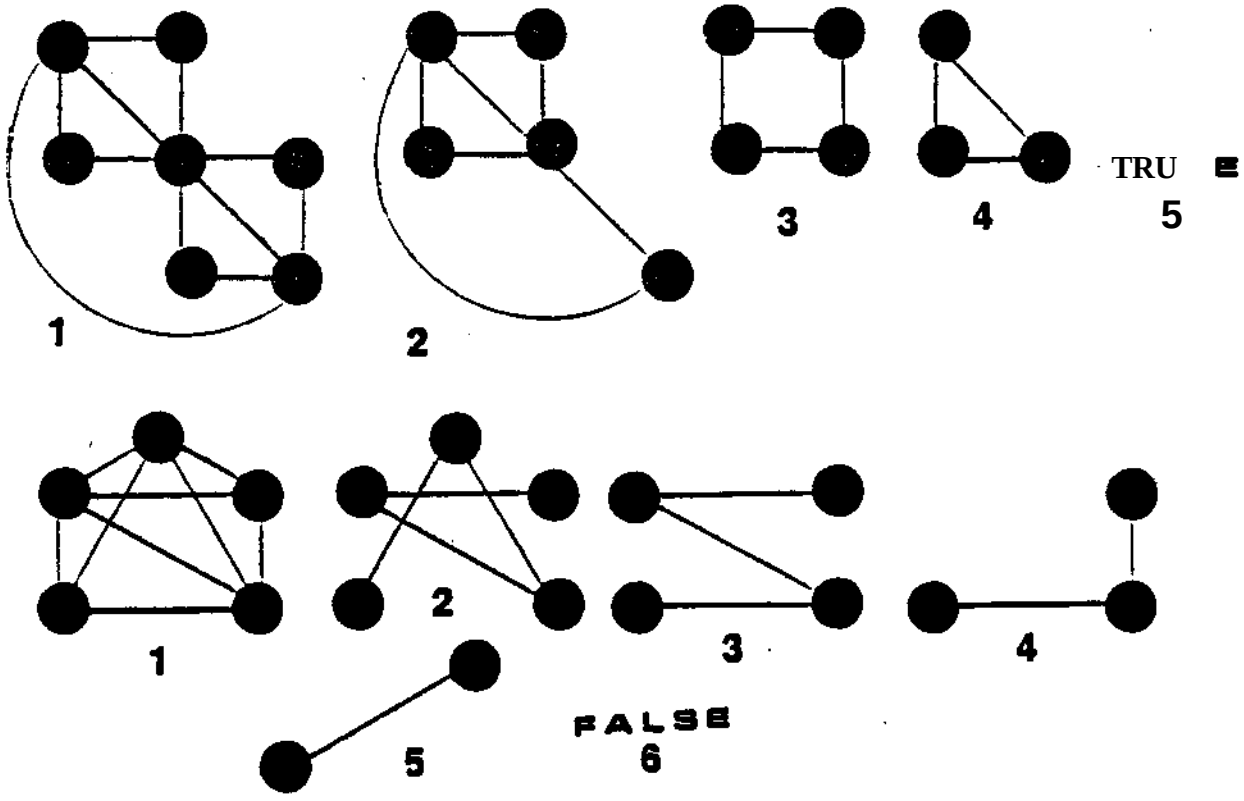



**Figure 3-49:** EULERIAN"[1] in Operation

degree zero will ciearly be the ones introduced by the cycle and deleted afterwards. If all of $v_r v_2$"*"$V_k$ were introduced to the graph by the cycle addition, they would all be of degree two on its completion, hence the "not air statement

On an Eulerian graph EULERIAN[11] detaches closed path "detours" from the underlying Euierian walk without ever deleting the entire graph when it is a cycle, thanks to "not all $v_r v_2$'—$v_k$ of degree £ 2." Once the Eulerian graph being tested has no further closed path "detours[11], it is a simple cycle, to be contracted by $S^{-1}_{wvz}$ until it is isomorphic to $K_3$. On a non-Euierian 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.

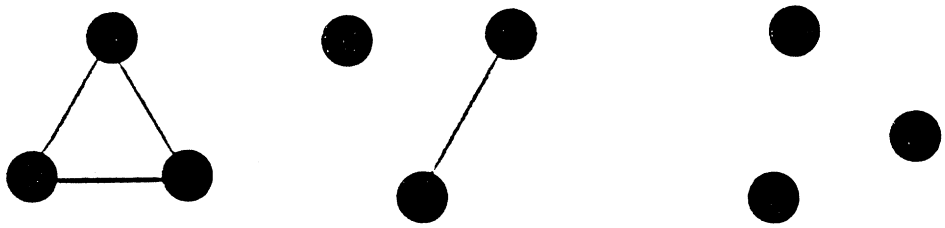


**Figure 3-50:** Some Graphs with 3 Vertices

The R-property K-VERTICES is:

$A^{*}_{xy}(E_k)$ where x,y ∈ V

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

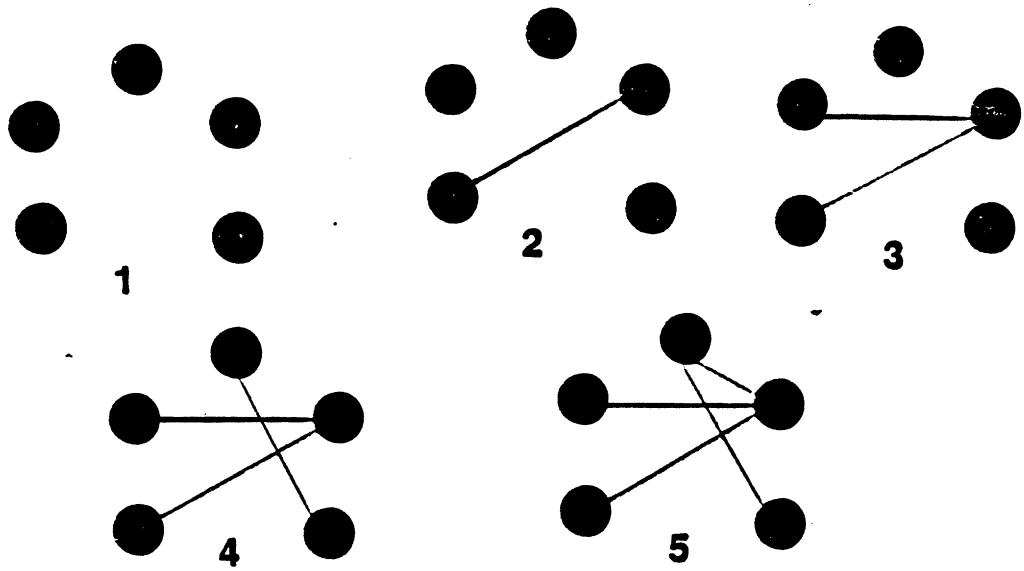



**Figure 3-51:** 5-VERTICES in Operation

$L_1$, $L_2$ 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 $<P_1, L_{1n}, \Sigma_1>$.

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:

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

$= D_{xy}$

$\sigma_{pre}$ $= x,y \in V,\ xy \notin E$

$\sigma^{-1}$ $= x,y \in V,\ xy \in E$

The floor shifts to $<P_2, L_{1n}, \Sigma_1>$. Figure 3-52 shows K-VERTICES$^{-1}$ operating on a graph $G \in G_p$ and a graph $G \notin G_p$ for k = 4.

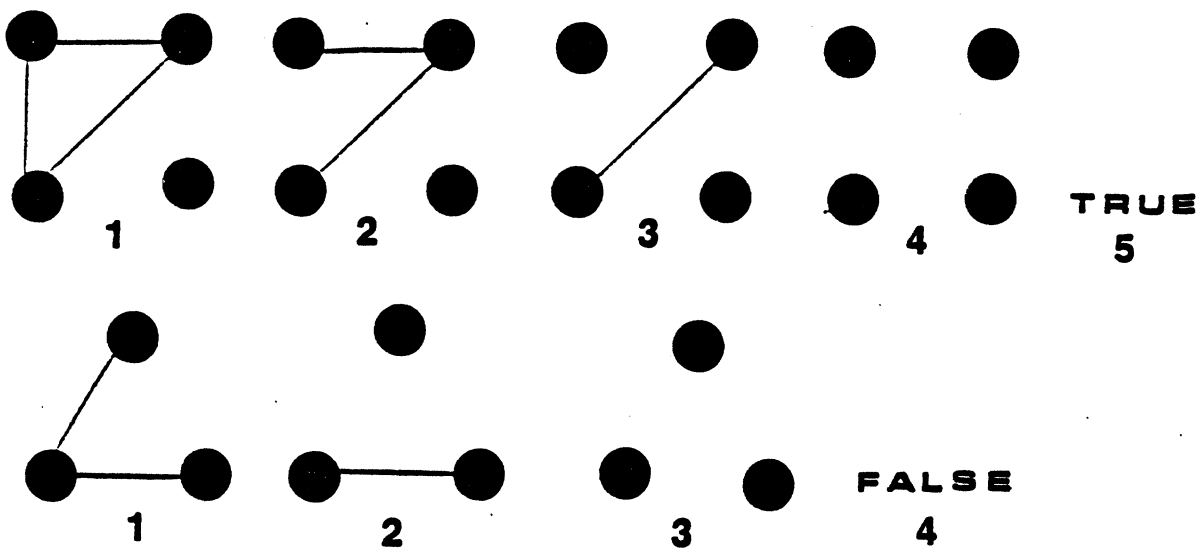



**Figure 3-52:** 4-VERTICES$^{-1}$ in Operation

K-VERTICES$^{-1}$ is correct; it deletes all edges and then tests for isomorphism between two sets of isolated vertices. Thus K-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 $M_k = <\{v_1, v_2, \ldots, v_{2k}\}, \{v_1v_2, v_3v_4, \ldots, v_{2k-1}v_2\}>$ 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-EDGES is:

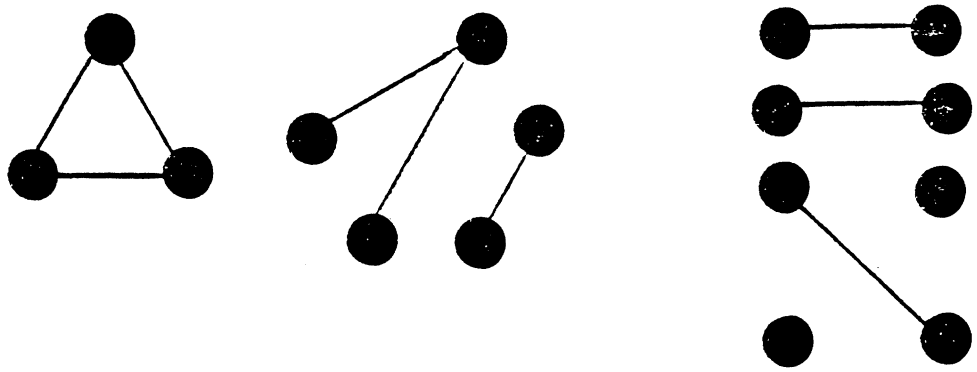


**Figure 3-53:** Some Graphs with 3 Edges

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

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

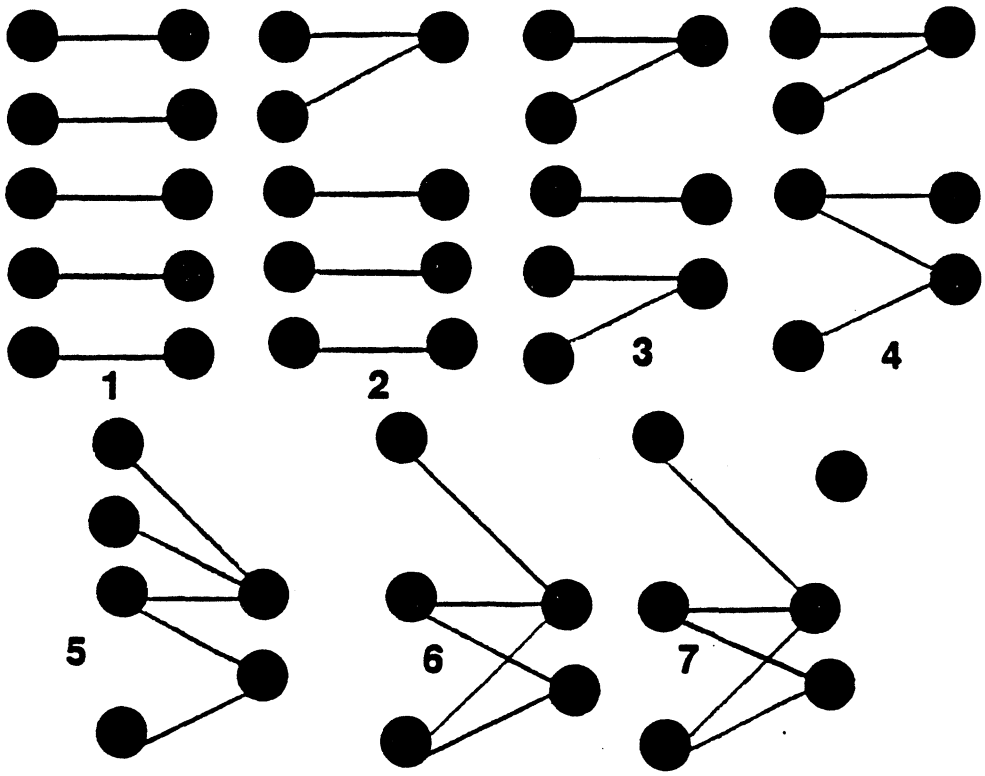



**Figure 3-54:** 5-EDGES in Operation

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 ∈ V such that yw,zw ∈ E" prevents this. We prevent the transformation of an edge into a loop by "yz ∉ 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:

$$f^{-1} = (A_x + I_{yz})^{-1}$$
$$= A_x^{-1} + I_{yz}^{-1}$$
$$= D_x + F_{yzv_1 \cdots v_k}$$

$\sigma_{pre}$ = $x \notin V$

distinct $y,z \in V$, $d(y) > 0$, $d(z) > 0$, $yz \notin E$,

$\sim[w \in V,\ yw,zw \in E]$

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

$y \in V,\ z \notin V,\ yw_i \in E,\ i = 1,2,\ldots,k;\ 2 \le d(y),\ d(y) > k$

The floor remains constant. The use of the fragmentation technique $F_{yzv_1 \cdots v_k}$ 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 $v_1,\ldots,v_k$.) 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-EDGES$^{-1}$ operating on a graph $G \in G_p$ and a graph $G \notin G_p$ for k = 4.

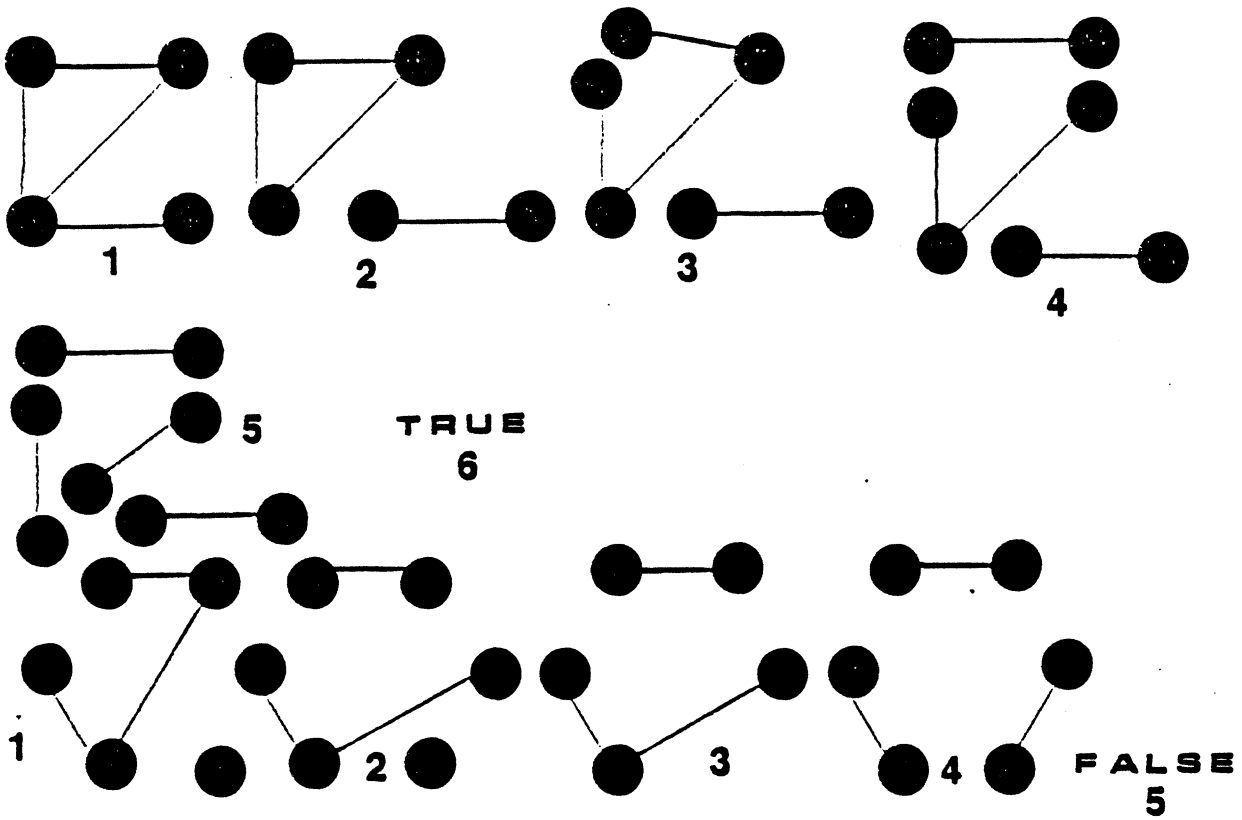



**Figure 3-55:** 4-EDGES$^{-1}$ in Operation

On a graph with k edges, K-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-EDGES$^{-1}$ will create $M_m$ and then fail. Thus K-EDGES$^{-1}$ is correct and K-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.

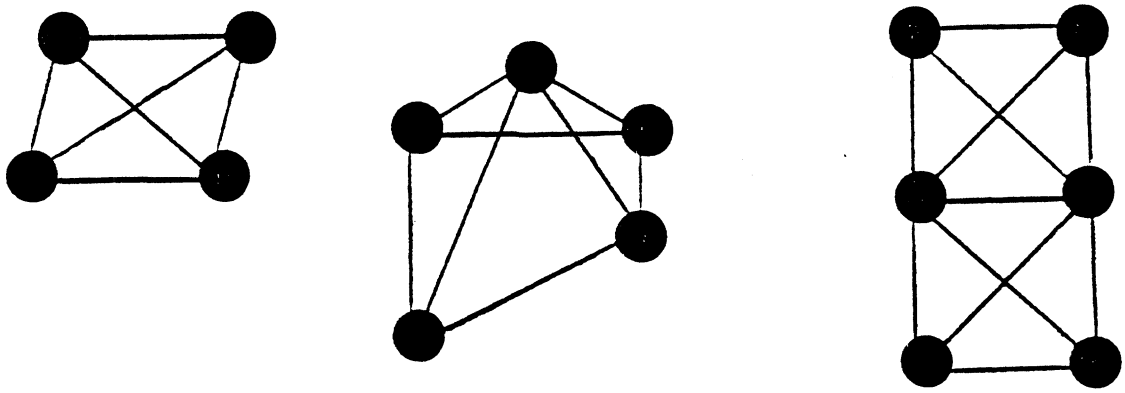


**Figure 3-56:** Some Graphs of Minimum Degree 3

The R-property MIN-K is:

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

distinct $x_1, x_2, \ldots, x_k \in V$, $v \notin V$

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 $\langle P_1, L_{1n}, \Sigma_3 \rangle$.

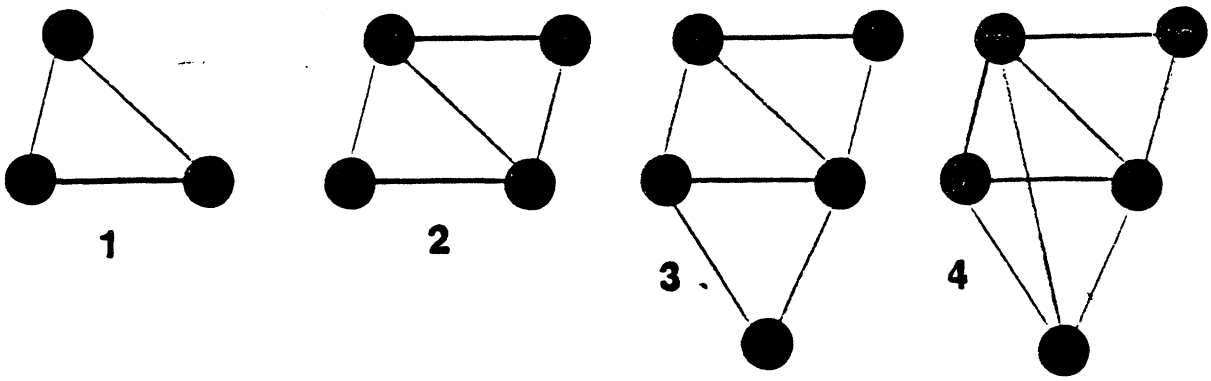



**Figure 3-57:** MIN-2 in Operation

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:

$$f^{-1} = (A_{xy} + A_{vx_1} A_{vx_2} \ldots A_{vx_k} A_v f^{-1}$$

$$= A_{xy}^{-1} + (A_{vx_1} A_{vx_2} \ldots A_{vx_k} A_v f^{-1}$$

$$= A_{xy}^{-1} + A_v^{-1} A_{vx_k}^{-1} A_{vx_{k-1}}^{-1} \ldots A_{vx_1}^{-1}$$

$$= D_{xy} + D_v D_{vx_k} D_{vx_{k-1}} \ldots D_{vx_1}$$

$a_{pre}$ = distinct x,y,w e V, xy * E, d(w) = k

distinct w,$x_1$,$x_2$,..$x_k$ e V, v * V

$or^{-1}$ = distinct x,y,w 6 V, xy 6 E, d(x) > k, d(y) > k, d(w) = k

distinct v,$x_1x_2$,...,$x_k$ e V, $xv_1xv_2$,.-,$xv_k$ € E, d(v) = k

The floor shifts to <p2'Lin'^3>> Rgure 3~58 shows MIN-K~1 operating on a graph G « $G_p$ and a graph G * $G_p$ 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 $K_{R+1}$ 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 s 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

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

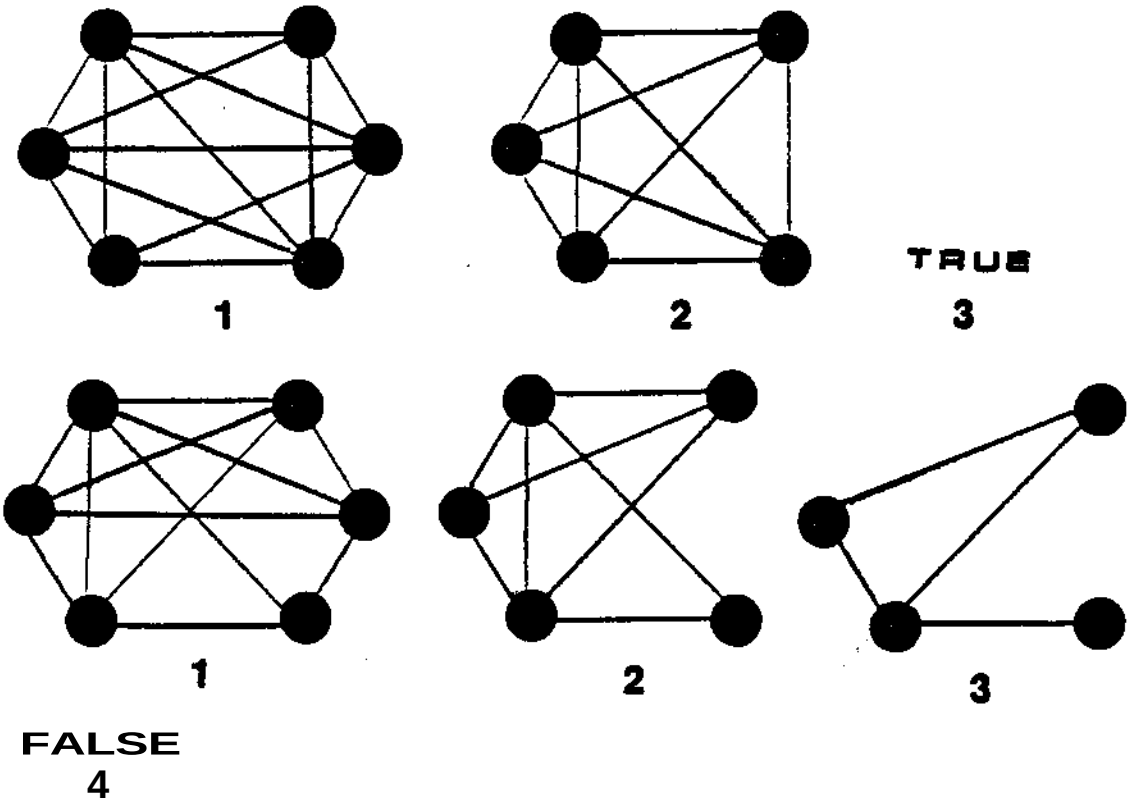



**Figure 3-58:** MIN-4"[1] in Operation

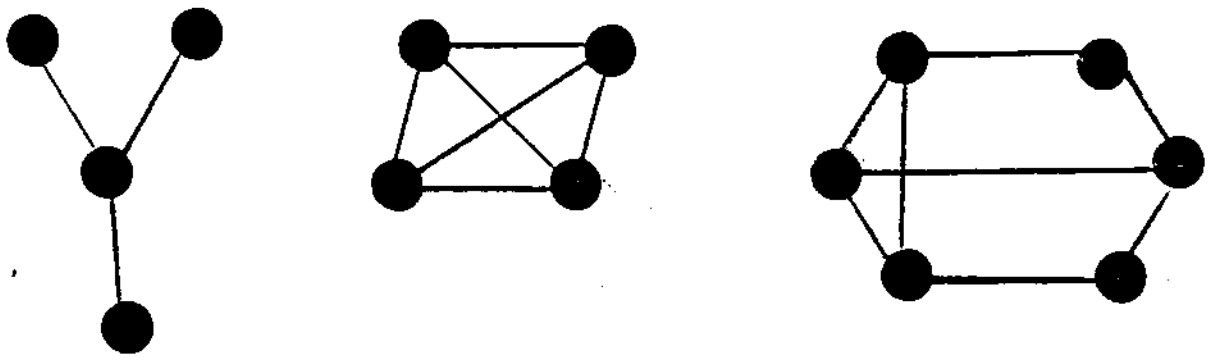


**Figure 3-59:** Some Graphs of Maximum Degree 3

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 $\langle P_1, L_\Omega, \Sigma_3 \rangle$.

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.

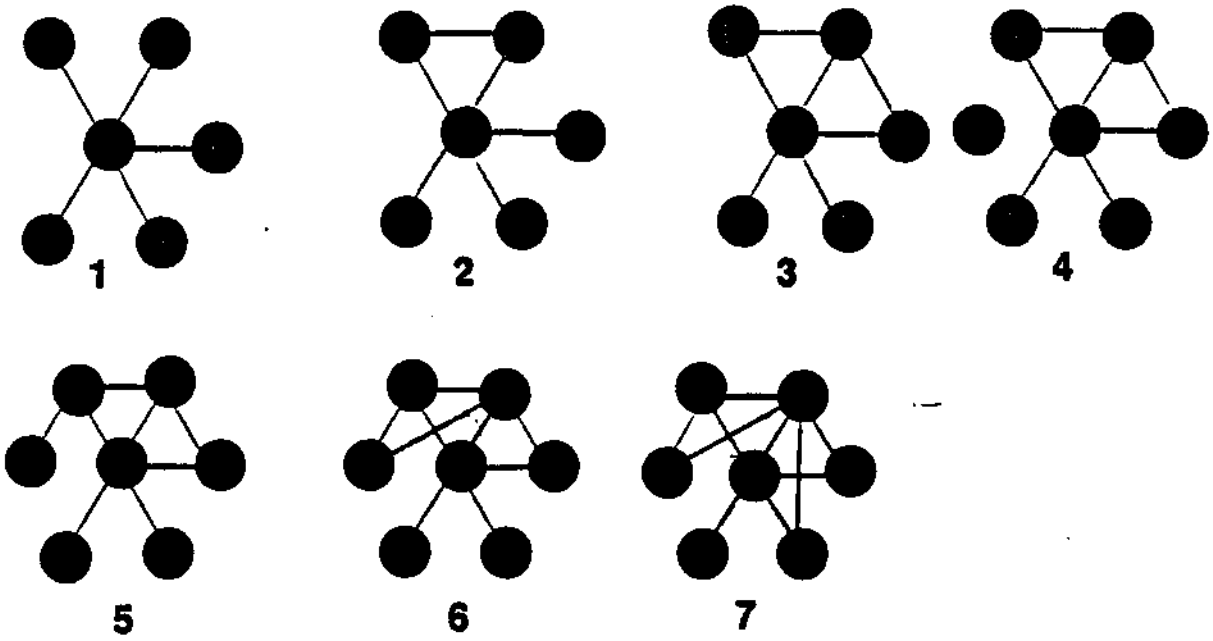



**Figure 3-60: MAX-5 in Operation**

**The inverse is computed by:**

$$f^{-1} = \langle_{y} + A_{z}^{y-1}$$
$$= A_{xy}^{-1} + A_{2}^{-1}$$
$$= D_{xy} + D_{2}$$

$$\sigma_{pre} = \text{distinct } w,x,y \in V,\ xy\ M\ E,\ d(x) < k,\ d(y) < k,\ 2 * V,\ d(w) = k$$

$$2 * V$$

$$\langle j\sim^{1} = \text{distinct } w,x,y \in V,\ xy \in E,\ d(x) \wedge k,\ d(y)\ £\ k > d(w) = k$$

$$2 \ll V,\ d(2) = 0$$

The floor shifts to $^{<P}2\wedge0\wedge3^{>4}$ Note that a re incorporates Properties of $K_{1k}$ preserved under f to deduces that some vertex of degree k will always be present Figure 3-61 shows MAX-$K^{*1}$ operating on a graph $G \ll G_p$ and a graph $G * G_p$ for k = 4.

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

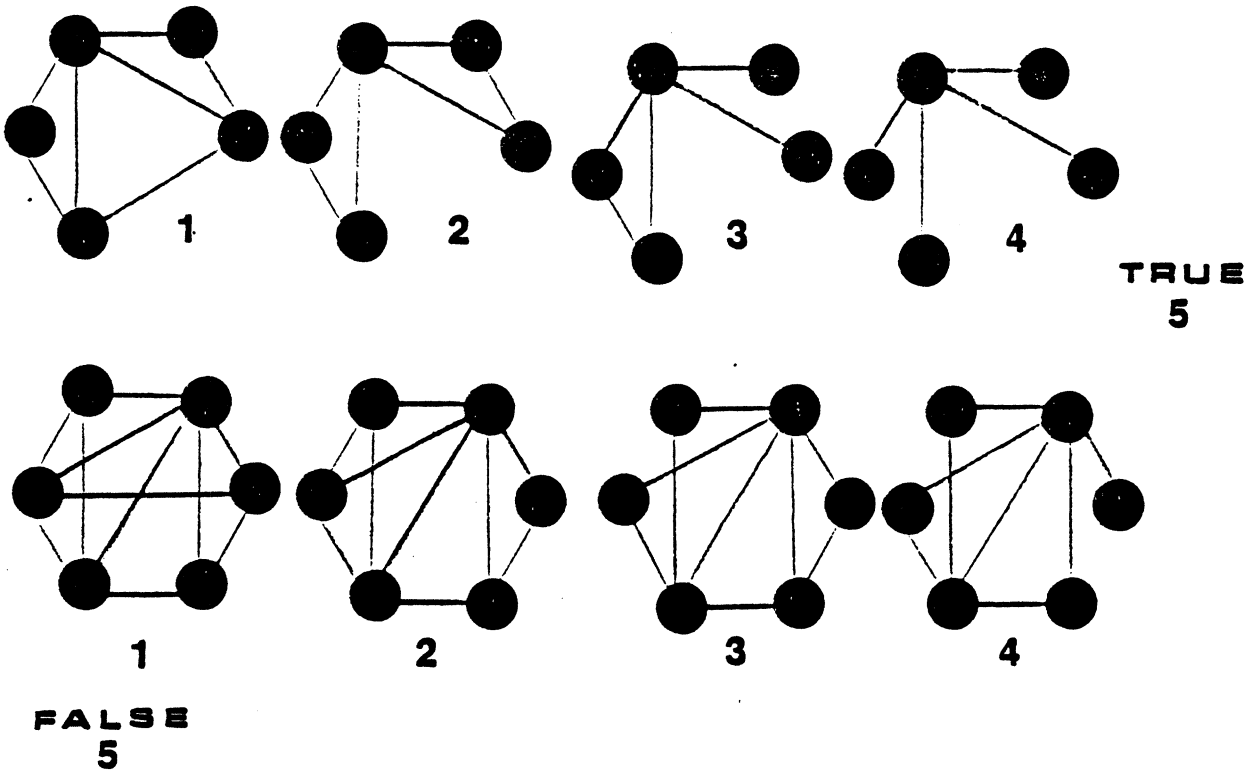



**Figure 3-61:** MAX-$4^{-1}$ in Operation

degree k vertex until only $K_{1,k}$ remains. If a graph has a vertex z of degree greater than k, 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 MAX-$K^{-1}$ and isomorphism to $K_{1,k}$ will never occur. Thus 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 $r \geq 3$

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

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

that is, a pinwheel is composed of a *rim* ($C_{w_1 w_2 \ldots w_r}$) and a complete graph (the *hub*) on $v_1, v_2, \ldots, v_h$. 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

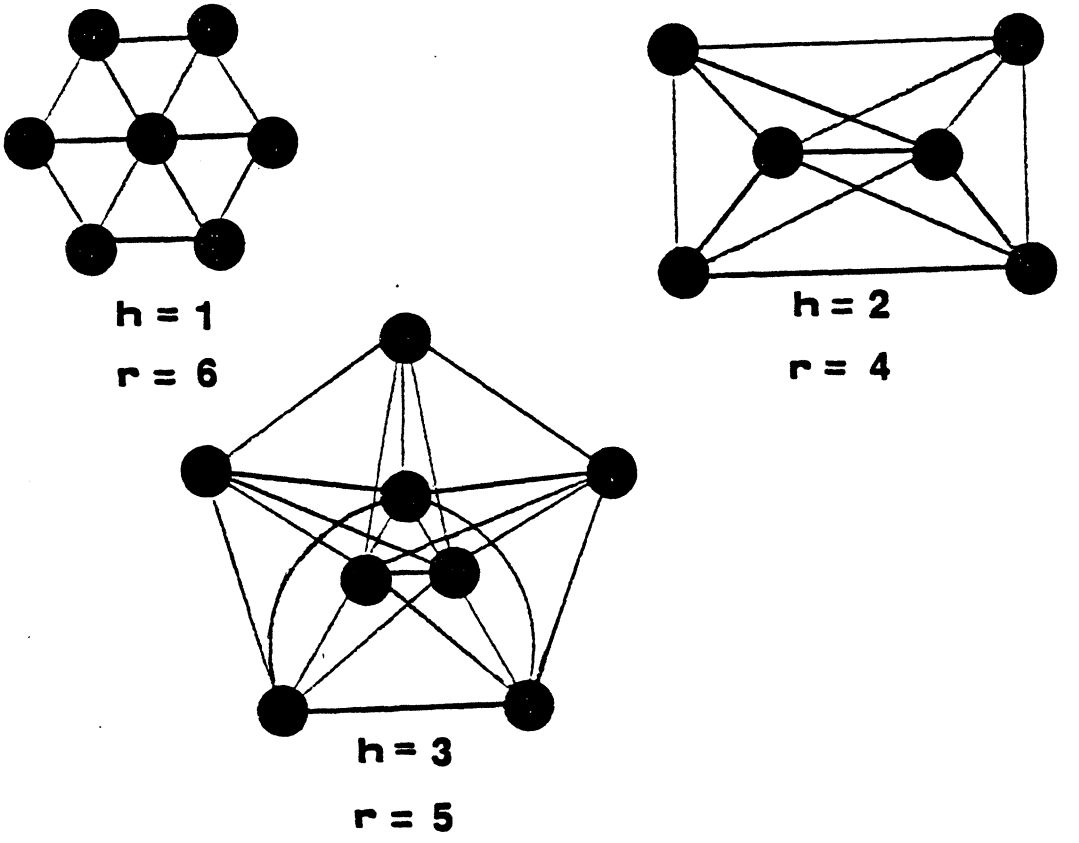



**Figure 3-62:** Some Pinwheels

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

$$(A_{vv_1}A_{vv_2}...A_{vv_k}S_{xvy})^*(W_{k,3}) \text{ where distinct } x,y,v_1,v_2,...,v_k \in V,\ v \notin V,\ xy \in E,$$
$$d(x) = k + 2,\ d(y) = k + 2,\ d(v_i) = \max,\ i = 1,2,...,k$$

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 $\langle P_2, L_\Omega, \Sigma_5 \rangle$, 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:

$$\begin{aligned} f^{-1} &= (A_{vv_1}A_{vv_2}...A_{vv_k}S_{xvy})^{-1} \\ &= S^{-1}_{xvy}A^{-1}_{vv_k}A^{-1}_{vv_{k-1}}...A^{-1}_{vv_1} \\ &= D_vD_{vy}D_{xv}A_{xy}D_{vv_k}D_{vv_{k-1}}...D_{vv_1} \end{aligned}$$

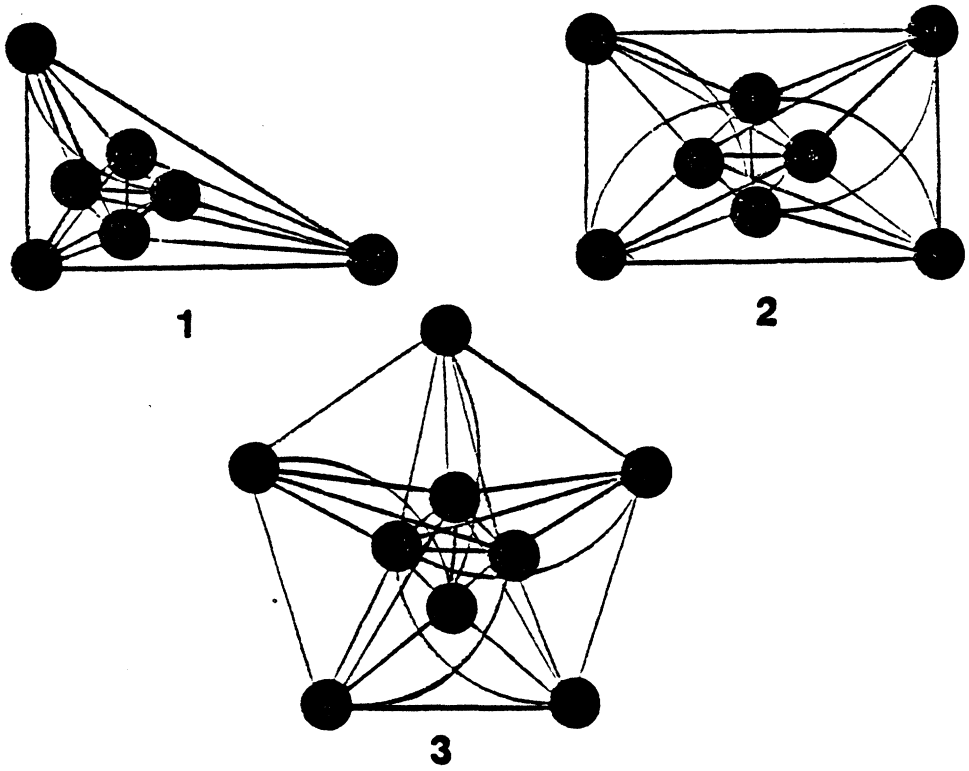



**Figure 3-63:** PINWHEEL in Operation

$\sigma_{pre} = \sigma$ = distinct $x,y,v_1,v_2,\ldots,v_k \in V$, $v \notin V$, $xy \in E$,
$d(x) = k + 2$, $d(y) = k + 2$, $d(v_i) = max$, $i = 1,2,\ldots,k$

$\sigma^{-1}$ = distinct $x,y,v,v_1,v_2,\ldots,v_k \in V$, $xy \notin E$, $xv,vy \in E$,
$d(x) = k + 2$, $d(y) = k + 2$, $d(v) = k + 2$,
$d(v_i) = max$, $i = 1,2,\ldots,k$

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

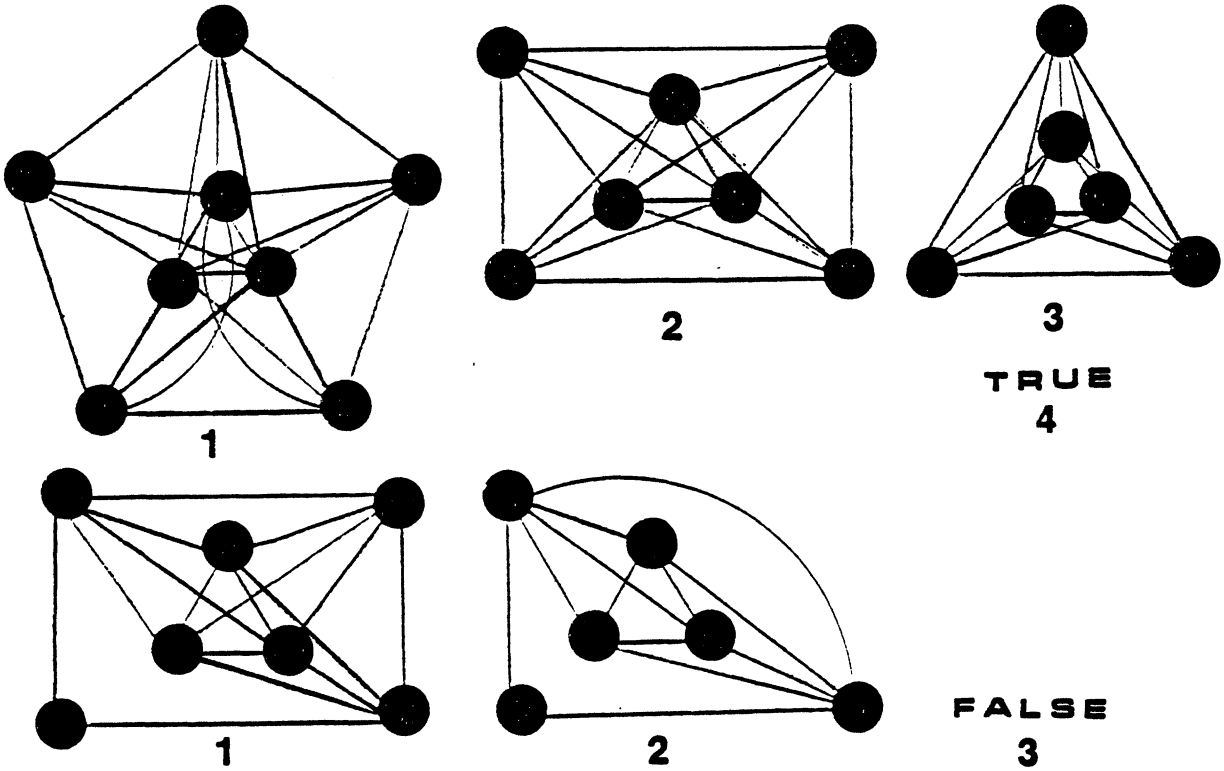



**Figure 3-64:** PINWHEEL$^{-1}$ in Operation

### 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 $\langle V_1,S_1 \rangle, \langle V_2,S_2 \rangle, \ldots, \langle V_k,S_k \rangle$ such that

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

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

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

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

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

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

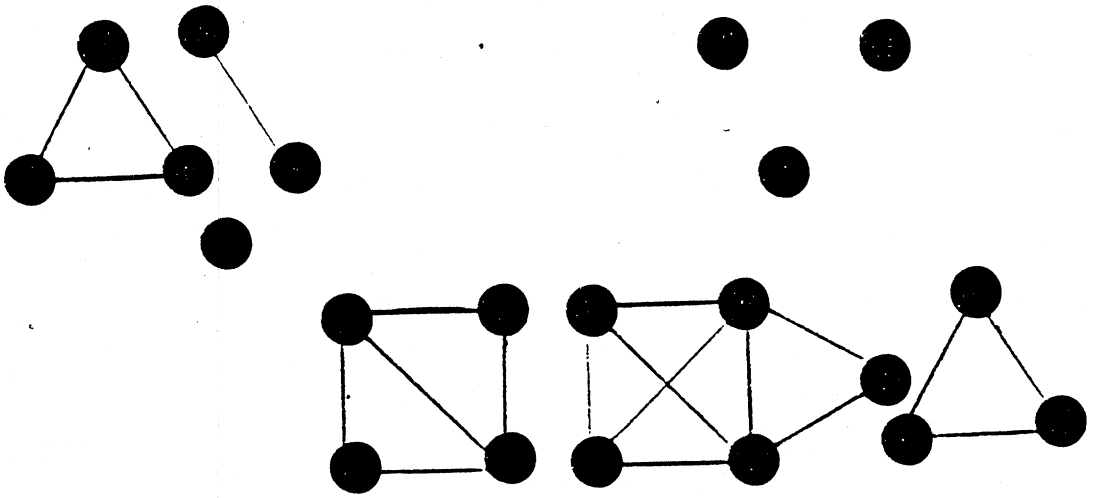


**Figure 3-65:** Some Graphs with Three Components

adjacency to $v_1, v_2, \ldots, v_t$ 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 $v_1, v_2, \ldots, v_t$, or both. Figure 3-66 shows the iterative steps in a sample run of K-COMPONENTS for k = 2.

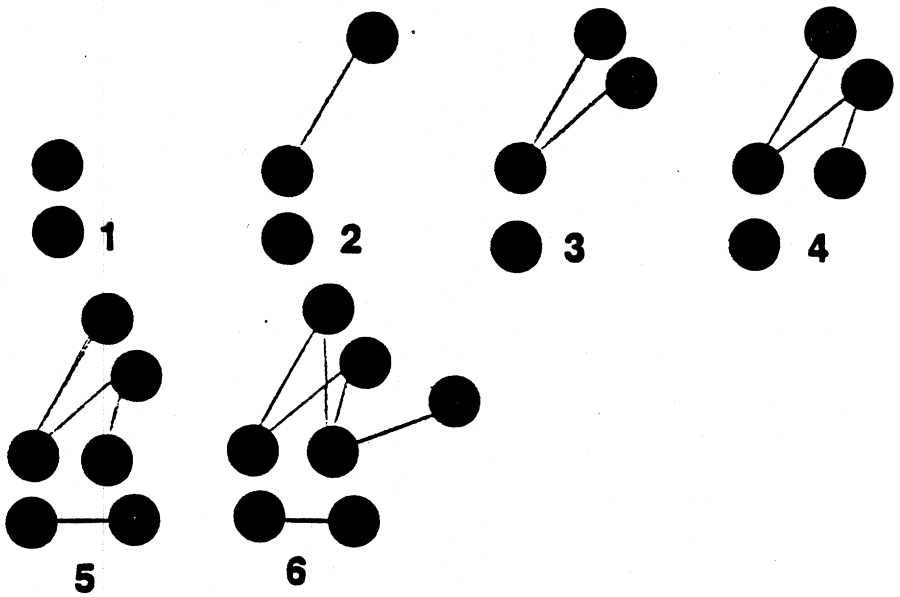



**Figure 3-66:** 2-COMPONENTS in Operation

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 $xv_iyw...v$ is a valid xv path. In either case, G is connected and K-COMPONENTS is correct

The inverse is computed by:

$$f\sim^{1} = (B_{wz} + (A_{xy} + A_{xv_1} A_{xv_2} ..A_{xv_t} + A_{xy} A_{xv_1} A_{xv_2} ...A_{xv_t}) F_{xyv_r\sim v_t} f^{1}$$

$$- B"^{1}_{wz} + F'^{1}_{xyv_1...v_t} (A_{xy} + A_{xv_1} A_{xv_2} ...A_{xv_t} + A_{xy} A_{xv_1} A_{xv_2} ... A_{xv_t} f^{1}$$

$$= B"^{1}_{wz} + F"^{1}_{xyv_1...v_t} (A"^{1}_{xy} + (A_{xv_1} A_{xv_2} ...A_{xv_t} f^{1} + (A_{xy} A_{xv_1} A_{xv_2} ... A_{xv_t})^{-1})$$

$$= B\text{-}^{1}_{wz} + F"^{1}_{xyv_1...v_t} (A\text{-}^{1}_{xy} - A"^{1} A" L A\text{-}^{1}_{xv_t} + A\text{-}_{xv_t} V L_{xv_{t-1}} A\text{-}^{1}_{xv_1} A"^{1}_{xy})$$

$$= D_z D_{wz} + I (D_{xy} + D_{xy} D_{xv_t} ...D_{xv_{t-1}} + D_{xv_1} D_{xv_t} D_{xv_{t-1}} ... D_{xv} D_{xy})$$

*a* sa = w e V, z * V
distinct $x,v_1,v_2,v_t$ s V, y 0 V, d(x) > 0,
$xv_i$ e E, i=1,2,~,t

a"$^{1}$ « distinct w,z e V, wz s E d(z) = 1
distinct x, y, $v_r v_r v_t$ e V, d(x) > 0, d(y) > 0,
xy e E or $xv_i, yv_i$ e E, i = l,2,...,t

The floor remains constant (Although "or" is not in *2y* an equivalent more complex notation in $Z_3$ can convey the same list of possible bindings.) Figure 3-67 shows K-COMPONENTS"$^{1}$ operating on a graph G e $G_p$ and a graph G * $G_p$ 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.

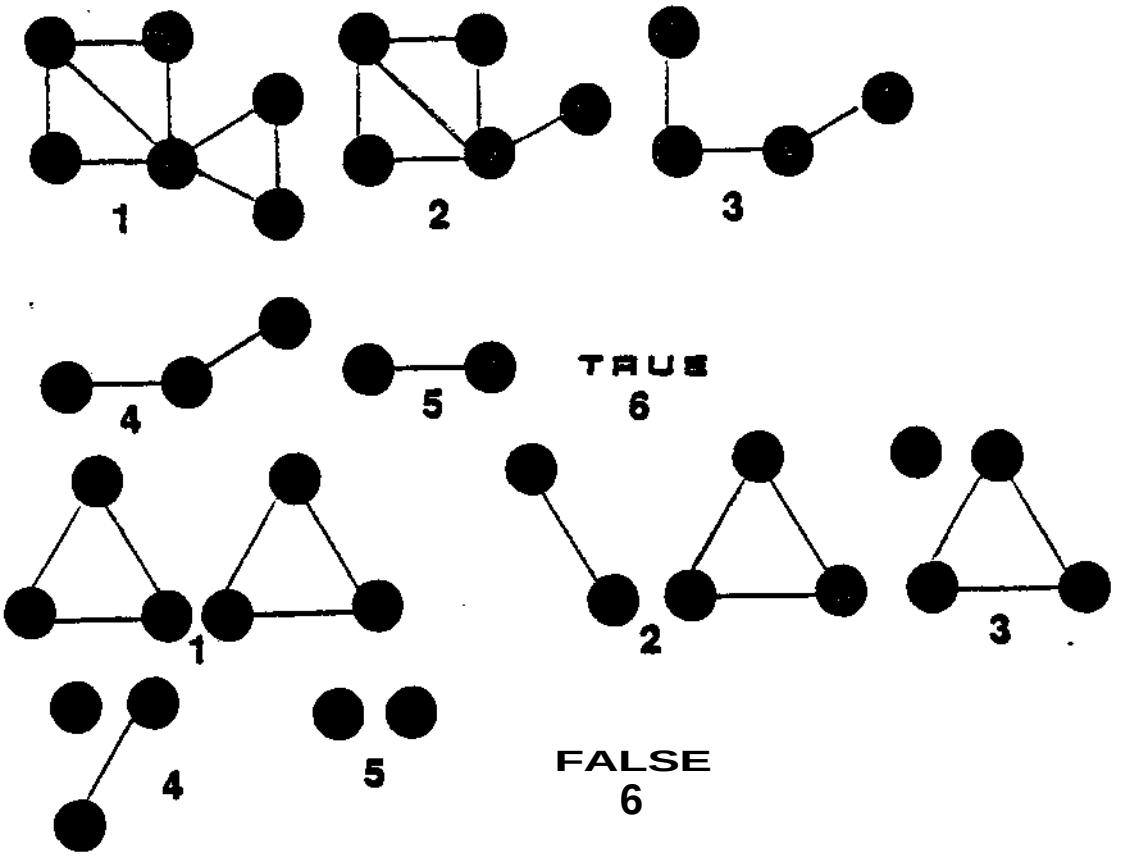



**Figure 3-67:** 1-COMPONENTS"[1] in Operation

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:

$$((A_{\_} + A_{\_}A_{1}A_{2}A_{t} + A_{\_}A_{1}A_{2}A_{t}\ldots A_{\_})F_{\_})^{*}(K,J$$

where distinct $x, v_r v_2, \_, v_t \in V$, $y \neq V$, $d(x) > 0$, $xv_i \in E$, $i = 1.2 \_\_ t$

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, $EM_{\_\_\_}$, $OM^{1}_{\_\_\_\_}$, $OM^{2}_{\_\_\_\_}$, $OM_{\_\_\_\_}$, $CM_{\_\_\_\_}$, and $PR_{\_\_\_\_}$.

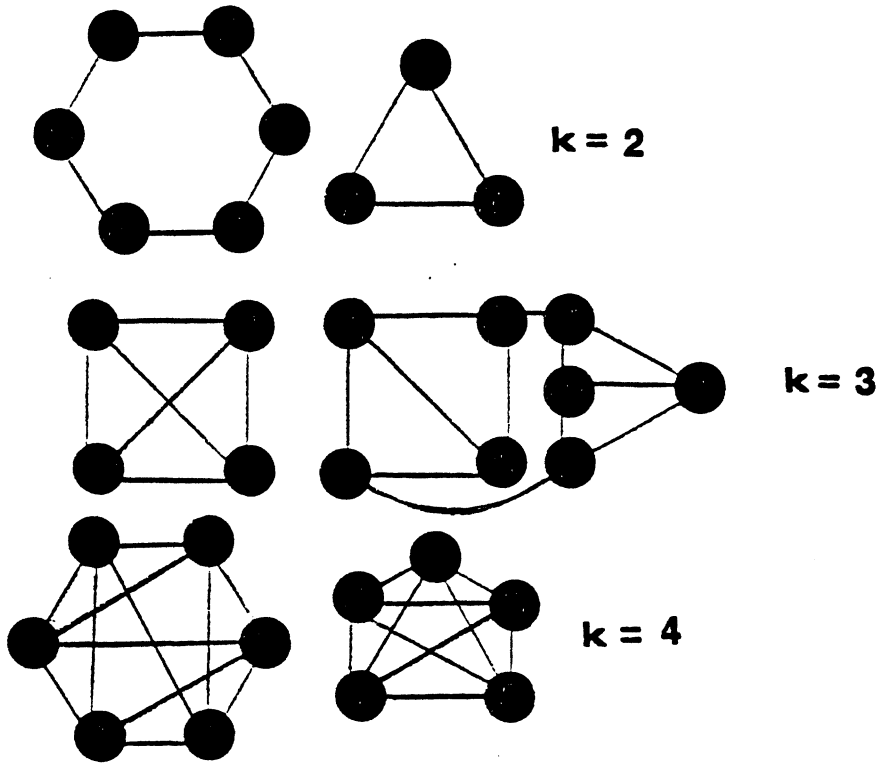



**Figure 3-68:** Some Regular Graphs

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

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

$OM_{vwv_1 \cdots v_{k+1} w_1 \cdots w_{k-1}}$ 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. $OM^1_{vwv_1 \cdots v_{k-1} w_1 \cdots w_{k-1}}$ 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. $OM^2_{vwv_1 \cdots v_{k+1} w_1 \cdots w_{k-1}}$ 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:

$$OM^1_{vwv_1 \cdots v_{k-1} w_1 \cdots w_{k-1}} = A_{vv_1} \cdots A_{vv_{k-1}} D_{v_1 v_2} \cdots D_{v_{k-2} v_{k-1}} A_{ww_1} \cdots A_{ww_{k-1}} D_{w_1 w_2} \cdots D_{w_{k-2} w_{k-1}} A_{vw} A_v A_w$$

$$OM^2_{vwv_1\cdots v_{k+1}w_1\cdots w_{k-1}} = A_{vv_1}\cdots A_{vv_k} D_{v_1v_2}\cdots D_{v_kv_{k+1}} A_{ww_1}\cdots A_{ww_{k-1}} D_{w_1w_2}\cdots$$
$$D_{w_{k-2}w_{k-1}} A_{wv_{k+1}} A_v A_w$$

$$OM_{vwv_1\cdots v_{k+1}w_1\cdots w_{k-1}} = OM^1_{vwv_1\cdots v_{k-1}w_1\cdots w_{k-1}} + OM^2_{vwv_1\cdots v_{k+1}w_1\cdots w_{k-1}}$$

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

$$CM_{v_1\cdots v_{k+1}} = A_{v_1v_2} A_{v_1v_3}\cdots A_{v_1v_k} A_{v_2v_3}\cdots A_{v_iv_j}\cdots A_{v_kv_{k+1}} A_{v_1}\cdots A_{v_{k+1}}$$

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

$FR_{v_1\cdots v_k}$ "fractures" the degree k vertex $v_1$ into a set of k vertices, $v_1,\ldots,v_k$, 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 $x_1,\ldots,x_k$, then

$$FR_{v_1\cdots v_k} = A_{v_2x_2} D_{v_1x_2} A_{v_3x_3} D_{v_1x_3}\cdots A_{v_kx_k} D_{v_1x_k} CM_{v_1\cdots v_k}$$

Also for notational convenience we define the inverses:

$$EM^{-1}_{xv_1\cdots v_k} = D_x A_{v_{k-1}v_k}\cdots A_{v_1v_2} D_{xv_k}\cdots D_{xv_1}$$

$$(OM^1_{vwv_1\cdots v_{k-1}w_1\cdots w_{k-1}})^{-1} = D_w D_v D_{vw} A_{w_{k-2}w_{k-1}}\cdots A_{w_1w_2} D_{ww_{k-1}}\cdots$$
$$D_{ww_1} A_{v_{k-2}v_{k-1}}\cdots A_{v_1v_2} D_{vv_{k-1}}\cdots D_{vv_1}$$

$$(OM^2_{vwv_1\cdots v_{k+1}w_1\cdots w_{k-1}})^{-1} = D_w D_v D_{wv_{k+1}} A_{w_{k-2}w_{k-1}}\cdots A_{w_1w_2} D_{ww_{k-1}}\cdots$$
$$D_{ww_1} A_{v_kv_{k+1}}\cdots A_{v_1v_2} D_{vv_k}\cdots D_{vv_1}$$

$$OM^{-1}_{vwv_1\cdots v_{k+1}w_1\cdots w_{k-1}} = (OM^1_{vwv_1\cdots v_{k-1}w_1\cdots w_{k-1}})^{-1}) +$$
$$(OM^2_{vwv_1\cdots v_{k+1}w_1\cdots w_{k-1}})^{-1}$$

$$CM^{-1}_{v_1\cdots v_{k+1}} = D_{v_{k+1}}\cdots D_{v_1} D_{v_kv_{k+1}}\cdots D_{v_1v_2}$$

$$FR^{-1}_{v_1\cdots v_k} = CM^{-1}_{v_1\cdots v_k} A_{v_1x_k} D_{v_kx_k} A_{v_1x_{k-1}} D_{v_{k-1}x_{k-1}}\cdots A_{v_1x_2} D_{v_2x_2}$$

Now the R-property for EVEN-REGULAR is

$(EM_{xv_1\cdots v_k} + A_{y_2q} A_{y_1p} D_{y_1y_2} CM_{y_1\cdots y_{k+1}} D_{pq} + FR_{z_1\cdots z_k})^*(Q_{k+1})$ where

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

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

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

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

$$Q_{k+1} = \{ \cup_{i=1}^{t} K^i_{k+1} \mid t=1,2,\ldots \}$$

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.

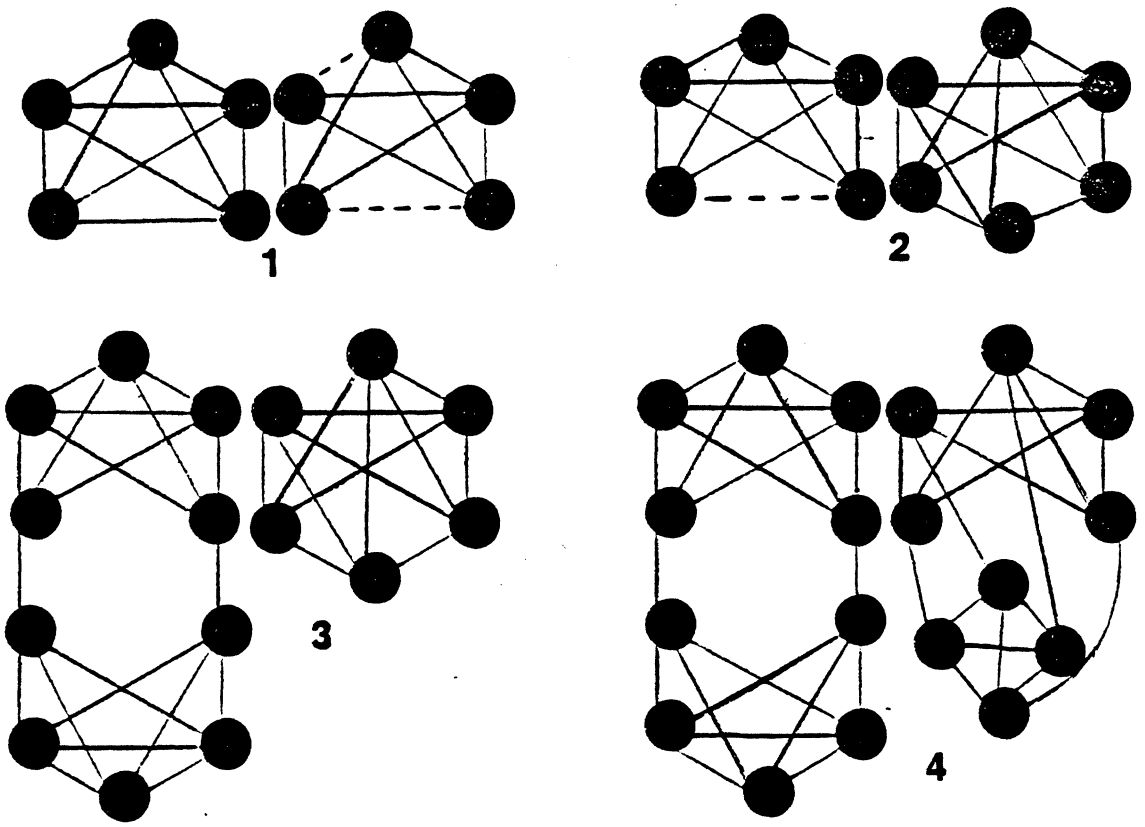



**Figure 3-69:** EVEN-REGULAR in Operation for k = 4

The edges to be used for the next iteration appear dotted in the figure. The floor for k-regular graphs with k even is $\langle P_2, L_\Omega, \Sigma_2 \rangle$. 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 $(y_1y_2)$ and connects the copy to the original graph with two edges ($y_1p$ and $y_2q$), 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:

$$f^{-1} = (EM_{xv_1\cdots v_k} + A_{y_2q}A_{y_1p}D_{y_1y_2}CM_{y_1\cdots y_{k+1}}D_{pq} + FR_{z_1\cdots z_k})^{-1}$$

$$= EM^{-1}_{xv_1\cdots v_k} + (A_{y_2q}A_{y_1p}D_{y_1y_2}CM_{y_1\cdots y_{k+1}}D_{pq})^{-1} + FR^{-1}_{z_1\cdots z_k}$$

$$= EM^{-1}_{xv_1\cdots v_k} + A_{pq}CM_{y_1\cdots y_{k+1}}A_{y_1y_2}D_{y_1p}D_{y_2q} + FR^{-1}_{z_1\cdots z_k}$$

$\sigma_{pre}$ = distinct $v_1, v_2, \ldots, v_k \in V$, $x \notin V$, $v_{2i-1}v_{2i} \in E$, $i=1,2,\ldots,k/2$, $d(v_j) = k$, $j = 1,2,\ldots,k$

distinct $p,q \in V$, distinct $y_1, y_2, \ldots, y_{k+1} \notin V$, $pq \in E$, $d(p) = k$, $d(q) = k$

distinct $x_1, x_2, \ldots, x_k, z_1 \in V$, distinct $z_2, z_3, \ldots, z_k \notin V$, $z_1x_i \in E$, $i = 1,2,\ldots,k$, $d(z_1) = k$

$\sigma^{-1}$ = distinct $x, v_1, v_2, \ldots, v_k \in V$, $xv_i \in E$, $i=1,2,\ldots,k$, $v_{2i-1}v_{2i} \notin E$, $i=1,2,\ldots,k/2$, $d(x) = k$, $d(v_i) = k$, $i=1,2,\ldots,k$

distinct $p,q,y_1, y_2, \ldots, y_{k+1} \in V$, $pq \notin V$, $y_iy_j \in E$, $d(y_i) = k$, $i = 1,2,\ldots,k+1$, $j = 1,2,\ldots,k+1$ except $y_1y_2$; $y_1p$, $y_2q \in E$

distinct $x_i, z_i \in V$, $x_iz_i \in E$, $d(x_i) = k$, $d(z_i) = k$, $i = 1,2,\ldots,k$; $x_1z_j \notin E$, $j = 2,3,\ldots,k$; $z_{rs} \in E$, $r,s = 1,2,\ldots,k$

The floor shifts to $\langle P_2, L_\Omega, \Sigma_3 \rangle$. 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 ($z_1$) 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.

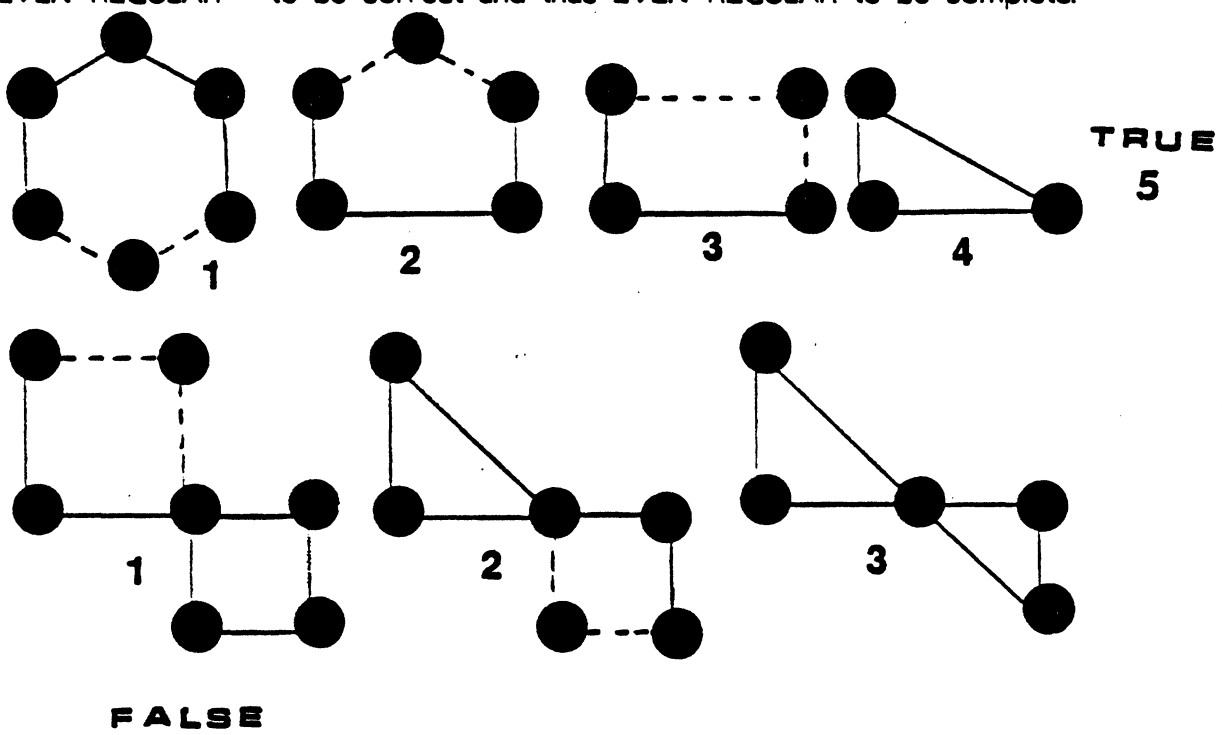



**Figure 3-70:** EVEN-REGULAR$^{-1}$ in Operation for k = 2

The R-property for ODD-REGULAR is

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

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

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

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

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

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

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.

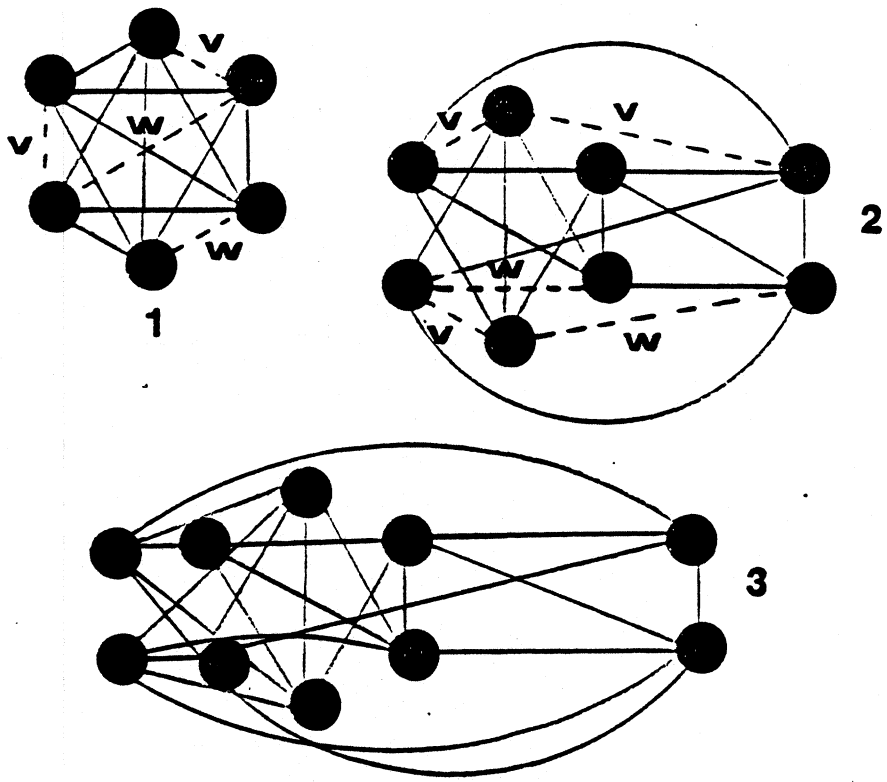



**Figure 3-71:** ODD-REGULAR in Operation for k = 5

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 $\langle P_2, L_\Omega, \Sigma_2 \rangle$. 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:

$f^{-1}$ = (OM$_{vwv_1 \cdots v_{k+1} w_1 \cdots w_{k-1}}$ +

A A D CM D + FR $f^{-1}$
$y_2 q$ $y_P$ $y_y y_2$ $y_1 y_{k+1}$ PQ $1 \cdots k$

= OM$^{-1}_{vwv_1 \cdots v_{k+1} w_1 \cdots w_{k-1}}$ +

(A A D CM D )$^{-1}$ + FR$^{-1}$
V $y_1 P$ $y_1 y_2$ $y_1 v_{k+1}$ Pq * $1 \cdots z_k$

= OM$^{-1}_{v wv_1 \cdots v_{k+1} w_1 \cdots w_{k-1}}$ +

A CM A D D + FR$^{-1}$
pq $y_r .. y_{k+1}$ $y_1 y_2$ $y_1 P$ $y_2 q$ v * k

*a* = distinct v,w « V, distinct $v_1, v_2, \ldots, v_{k+1}$ s V,
pre

distinct $x_r x_r$»,$x_{k-1}$ s V, d($v_i$) * d($x_i$) =k,

i = 1,2.-..k; distinct $V_j$.,^^, $w_{2j-1} w_{2j}$ s E,

i=1,2,...,(k+1)/2; j = 1.2,...,(k-1)/2

distinct p,q e v, distinct $y_r y_2$»-'$y_{k+1}$ * V, pq s E,

d(p) = k, d(q) = k

distinct $x_r x_{r-}, x_{k<2_1}$ s V, distinct $z_2, z_2, .^z_k$ * V,

$2^_i$ e E, i = 1,2^k, dJz^ = k

$a^{-1}$ = distinct v.v^$v_2$^...^ s V, distinct w,$w_r w_2$,«,$w_{k-1}$ s v

vw,$w_i$,ww$_j$ « E, i=1.2,~,k+1; j ~ 1,2,™,k-1

$^V 2i-1 ^V 2i' ^W 2i-1 ^W 2j$ * E' i=1.2_(k+1)/2;

j = 1,2_,(k-1)/2. dM = k, d(w) =k, d($v_i$) = k,

i= 1,2__k-t-1. d<$w_i$) » k, i= T,2,__k-1

distinct p,q,$y_r y_2$,~,$y_{k+1}$ e V, pq S V, y.$y_j$ 6 E. d($y_i$) = k,

i = 1,2._,k+1, j = 1,2.._,k+1 except y ^;

$y_i$p, $y_2$q s E

distinct $x_i, z_i$ « V, $x_i z_i$ •• E, d($x_i$) = k, d($z_i$) = k, i = 1,2,...,k;

x , !$_j$ i E j = 2.3,...,k; $z_{rs}$ s E, r,s = 1,2,...,k

The floor shifts to $^{<P_2}$'$^L$Q'E3$^>$- Pigure 3-72 shows ODD-REGULAR$^{-1}$ operating on a graph G s $G_p$ and a graph G « $G_p$ for k = 3. The edges to be used in the next

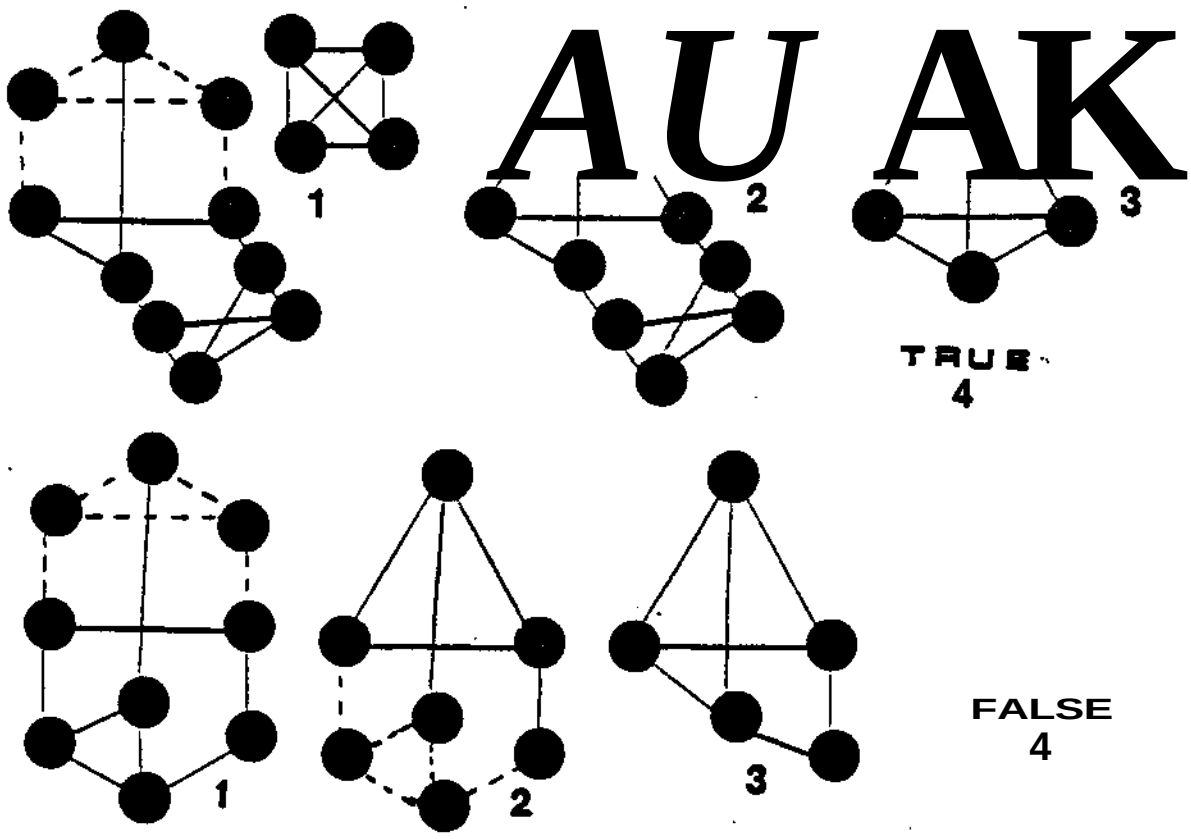



Figure 3-72: ODD-REGULAR"[1] in Operation for k = 3

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 * $G_p$ 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 « $G_p$ 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 = <V,E> is *connected* if for every pair of vertices in V there is a path constructabie between them using only edges in E Several examples of connected graphs appear in Figure 3-73.

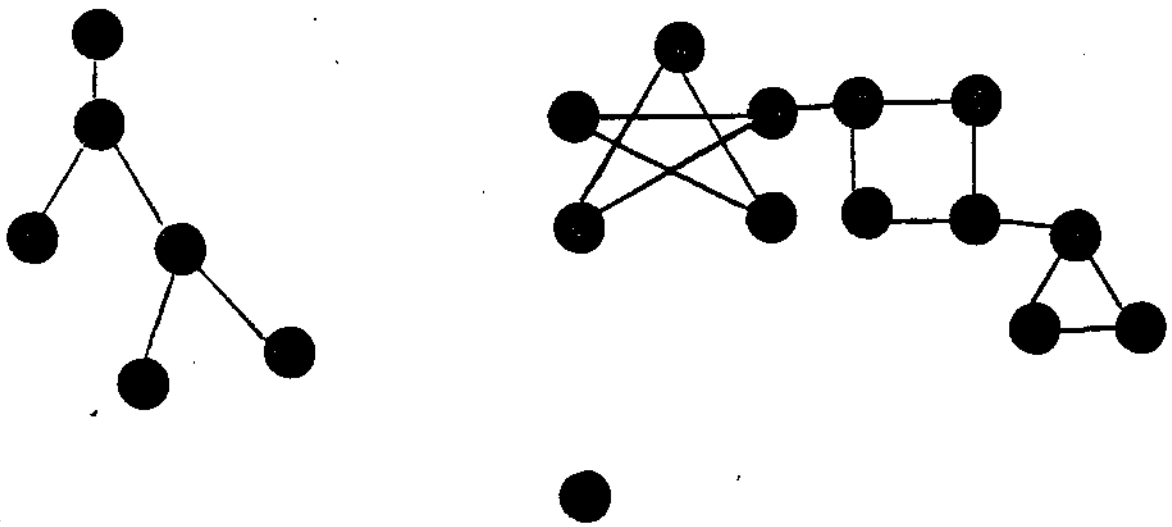


Figure 3*73: Some Connected Graphs

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

B*..(KJ where x e V
Xy I

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

«N + A 1UN + A r)A F 1 r)*(KJ
xv xv xy xyv ...v · 1
where distinct x,v.! e V, y 0 V, xv.! e E, 0 £ r £ d(x)

Figure 3-74 shows the iterative steps in a sample run of CONNECTED. The floor is <$P_4$,$L_r2_5$>.

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 (2 0) of x's adjacencies to the v.'s and permits x to retain any or all of those adjacencies as

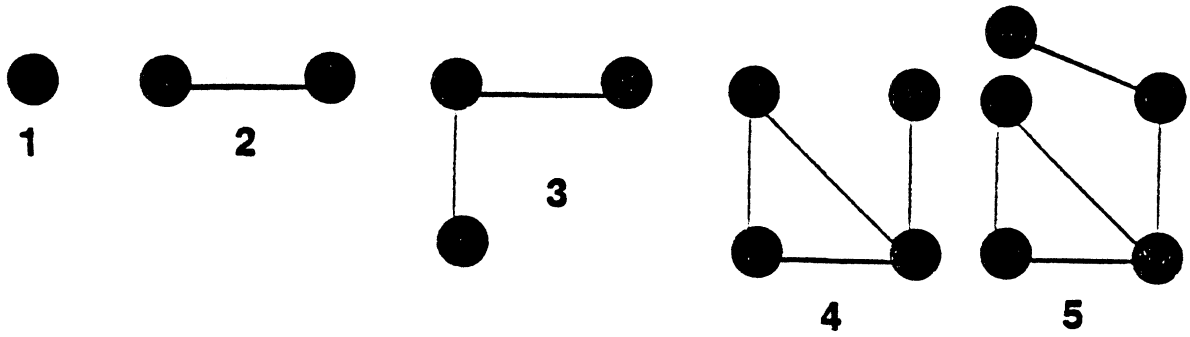



**Figure 3-74:** CONNECTED in Operation

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:

$$f^{-1} = ((N + A_{xv_1})\ldots(N + A_{xv_r})A_{xy}F_{xyv_1\ldots v_r})^{-1}$$

$$= F^{-1}_{xyv_1\ldots v_r}A^{-1}_{xy}(N + A_{xv_r})^{-1}\ldots(N + A_{xv_1})^{-1}$$

$$= I_{xy}D_{xy}(N + D_{xv_r})\ldots(N + D_{xv_1})$$

$$\sigma_{pre} = \sigma = \text{distinct } x,v_i \in V,\ y \notin V,\ xv_i \in E,\ 0 \le r \le d(x)$$

$$\sigma^{-1} = x,y,v_i \in V,\ xy,yv_1,\ldots,yv_r \in E,\ 0 \le r \le d(x)$$

The floor remains constant. Figure 3-75 shows CONNECTED$^{-1}$ operating on a graph $G \in G_p$ and a graph $G \notin G_p$. 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 = <V,E> 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

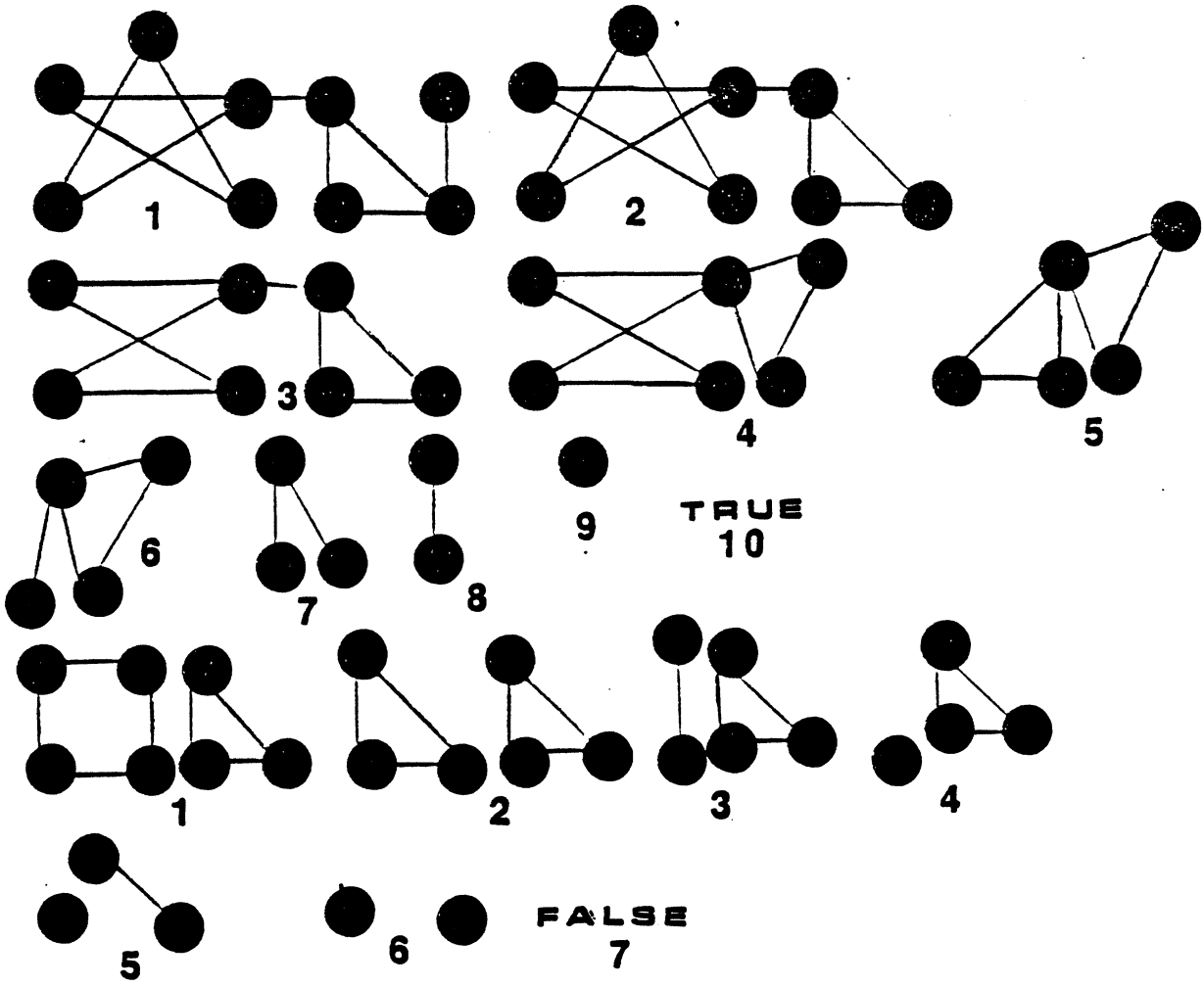



**Figure 3-75:** CONNECTED$^{-1}$ in Operation

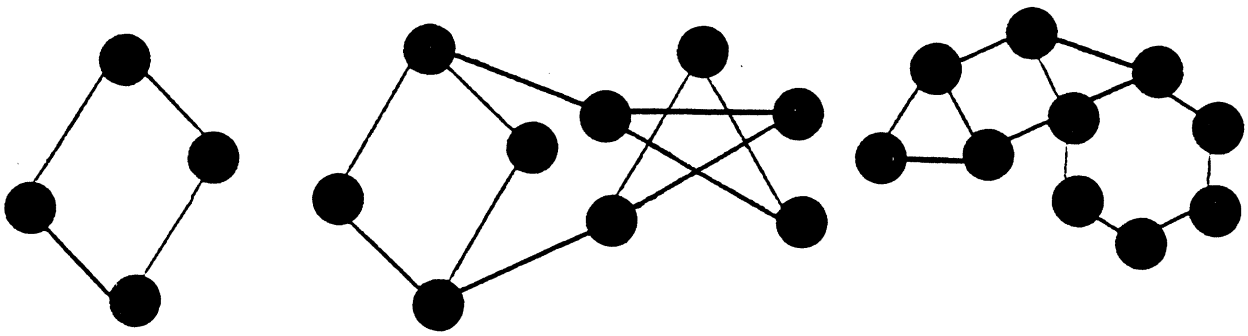


**Figure 3-76:** Some Biconnected Graphs

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

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

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

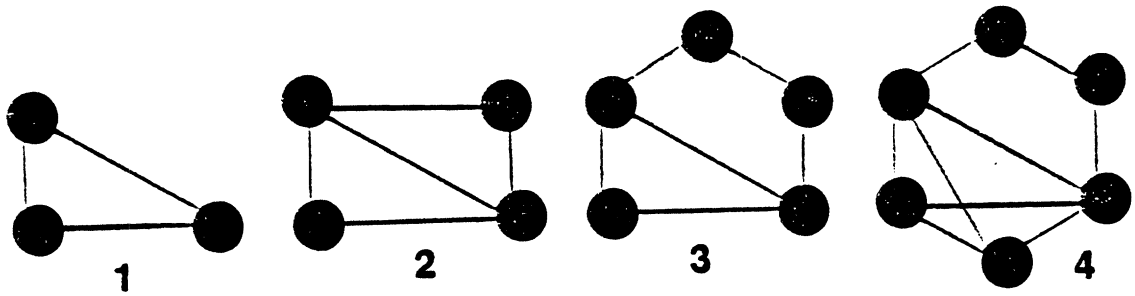



**Figure 3-77:** BICONNECTED in Operation

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 $xv_i$ is not present, $xyv_i$ 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:

$$f^{-1} = ((N + A_{xv_1})\ldots(N + A_{xv_r})A_{xy}F_{xyv_1\ldots v_r} + (N + A_{wt_1})\ldots (N + A_{wt_s})A_{wt_{s+1}}A_{wz}F_{wzt_1\ldots t_{s+1}})^{-1}$$

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

$$= I_{xy}D_{xy}(N + D_{xv_r})\ldots(N + D_{xv_1}) + I_{wz}D_{wz}D_{wt_{s+1}}(N + D_{wt_s})\ldots (N + D_{wt_1})$$

$\sigma_{pre}$ = distinct $x, v_i \in V$, $y \notin V$, $xv_i \in E$, $1 \le r \le d(x) - 1$, $d(x) > 1$, $d(v_i) > 1$, $i = 1,2,\ldots,r$

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

$d(t_i) > 1,\ i = 1,2,\ldots,s$

$a''''^1$ = distinct $x,y,v_i \in V$, $xy, yv_1 \ldots yv_r \in E$, $1 \le r$,

$d(x) > 1$, $d(y) > 1$, $d(v_i) > 1$,

$|\{pv_i \mid p \in V, pv_i \notin E, p \neq x, p \neq y\}| > 0$,

$i = 1,2 \ldots r$

distinct $w,z,t_1 \ldots t_{s+1} \in V$, $wz, zt_1, \ldots, zt_{s+1}, wt_1 \ldots t_{s+1} \in E$, $d(w) > 1$,

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

$|\{pt_i \mid p \in V, pt_i \in E, p \neq x, p \neq y\}| > 0$, $i = 1,2 \ldots s+1$

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 $G \notin G_p$. 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 $K_2$. Thus BICONNECTED$''''^1$ is correct and BICONNECTED is completa

### 3.7.23. k-Connected Graphs

A graph G = <V,E> 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:

$((N + A_{xv_1} \ldots N + A_{xv_r})A_{xy\,xyv_1 \ldots v_r} + (N + A_{wt_1} \ldots N + A_{wt_{s-k+2}})A_{wt_{s+1}} A_{wz} F_{wzt_1 \ldots t_{s+1}})^*(K_{k+1})$

where distinct $x,v_i \in V$, $y \notin V$, $xv_i \in E$, $k-1 \le r \le d(x) - k+1$

distinct $w,t_i \in V$, $2 \neq V$, $wt_i \in E$, $s = d(w) - k+1$

Figure 3-80 shows the iterative steps in a sample run of K-CONNECTED for k = 4. The floor for K-CONNECTED is $<P_4 J_{-1n} Z_5>$.

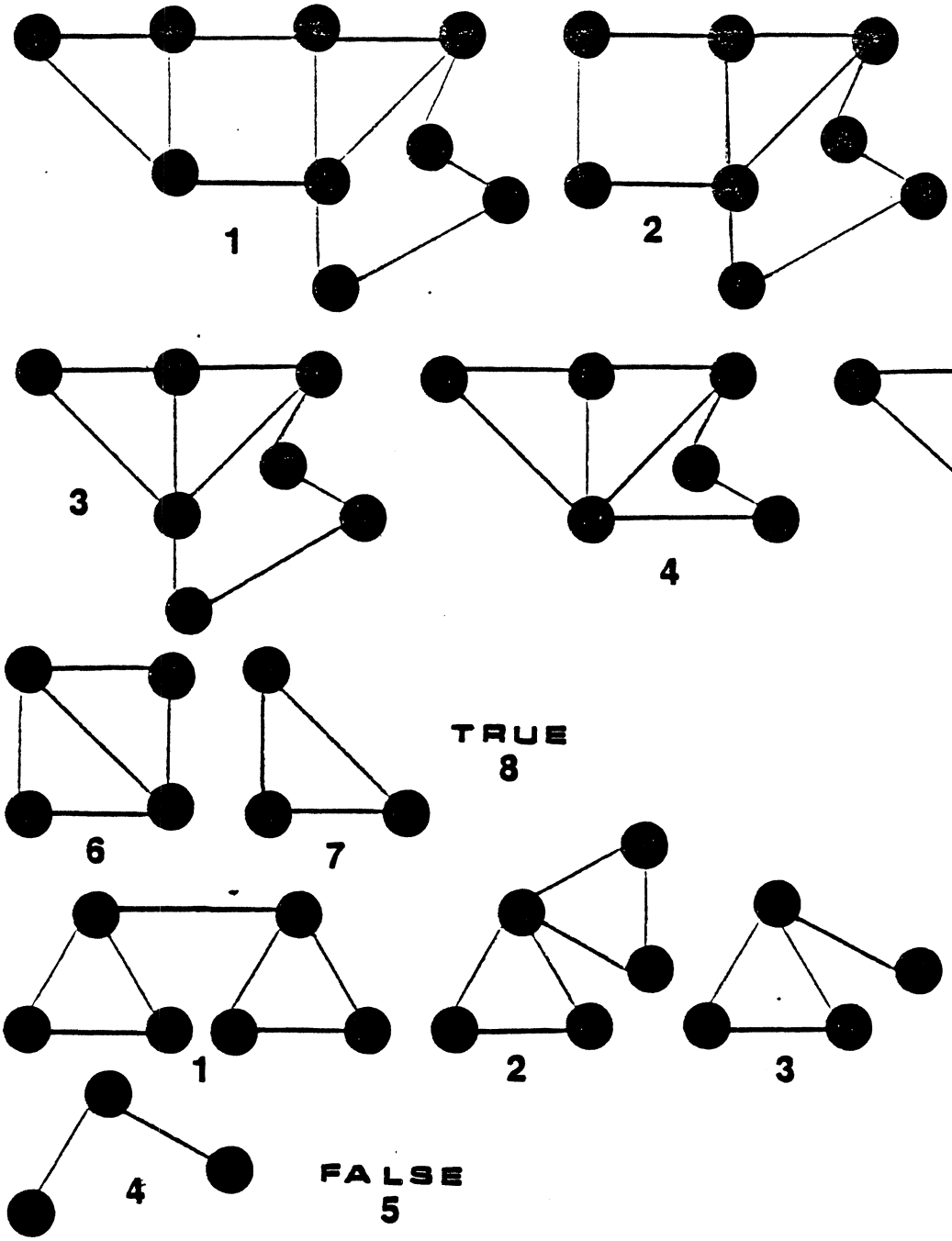



**Figure 3-78:** BICONNECTED$^{-1}$ in Operation

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

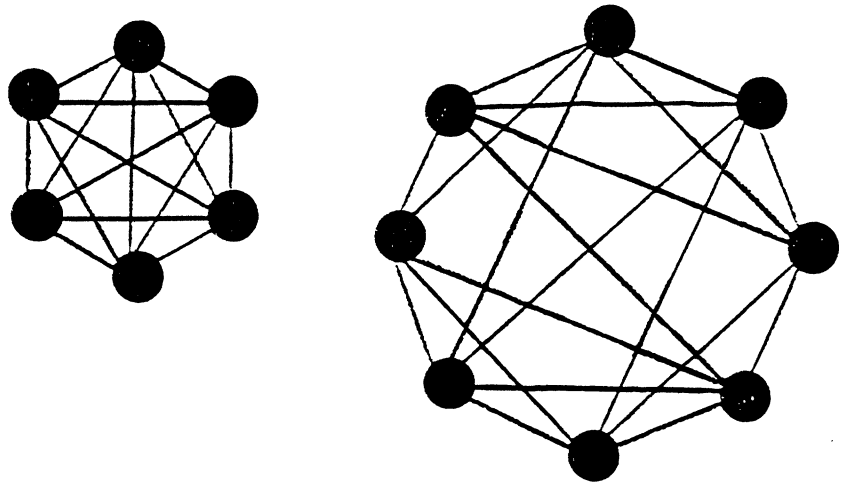


**Figure 3-79:** Some 5-Connected Graphs

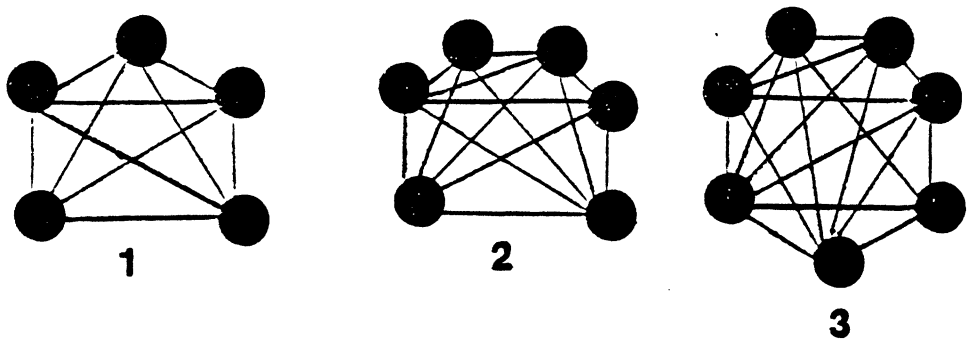



**Figure 3-80:** 4-CONNECTED in Operation

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 $w_1,\ldots,w_{k-1}$, 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-CONNECTED$^{-1}$ is computed similarly to that for BICONNECTED:

$$f^{-1} = ((N + A_{xv_1})\ldots(N + A_{xv_r})A_{xy}F_{xyv_1\ldots v_r} + (N + A_{wt_1})\ldots(N + A_{wt_{s-k+2}})A_{wt_{s-k+3}}\ldots A_{wt_{s+1}} A_{wz}F_{wzt_1\ldots t_{s-k+1}})^{-1}$$

$$= ((N + A_{xv_1})\ldots(N + A_{xv_r})A_{xy}F_{xyv_1\ldots v_r})^{-1}$$
$$+ ((N + A_{wt_1})\ldots(N + A_{wt_{s-k+2}})A_{wt_{s-k+2}}\ldots A_{wt_{s+1}}$$
$$A_{wz}F_{wzt_1\ldots t_{s+1}})^{-1}$$
$$= I_{xy}D_{xy}(N + D_{xv_r})\ldots(N + D_{xv_1}) +$$
$$I_{wz}D_{wz}D_{wt_{s+1}}\ldots D_{wt_{s-k+3}}(N + D_{wt_{s-k+2}})\ldots$$
$$(N + D_{wt_1})$$

$\sigma_{pre}$ $= \{x, v_i \in V,\ y \notin V,\ xv_i \in E,$
$k-1 \le r \le d(x) - k+1,\ d(x) > k-1,\ d(v_i) > k-1$
$\{w, t_i \in V,\ z \notin V,\ wt_i \in E,\ s = d(w) - k+1,$
$d(w) > k-1,\ d(t_i) > k-1$

$\sigma^{-1}$ $= \{x, y, v_i \in V,\ xy,\ yv_1, \ldots, yv_r \in E,$
$k-1 \le r < d(x) - k + 1,\ d(x) > k-1,\ d(y) > k-1,$
$d(v_i) > k-1,$
$|\{pv_i \mid p \in E,\ pv_i \in E,\ p \ne x,\ p \ne y\}| > k-2,$
$i=1,2,\ldots,r$
$\{w, z, t_i \in V,\ wz,\ zt_1, \ldots, zt_s, wt_{s-k+3}, \ldots, wt_{s+1} \in E,$
$s = d(w) - k + 1,\ d(w) > k-1,\ d(z) > k-1,$
$|\{pt_i \mid p \in E,\ pt_i \in E,\ p \ne w,\ p \ne z\}| > k-2,$
$d(t_i) > k-1,\ i=1,2,\ldots,s$

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-CONNECTED$^{-1}$ operating on a graph $G \in G_p$ and a graph $G \notin G_p$ 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-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-CONNECTED$^{-1}$ performs properly on $G \in G_p$ and will eventually reduce it to $K_{k+1}$. If $G \notin G_p$, there are k−1 points which disconnect it. This ability to

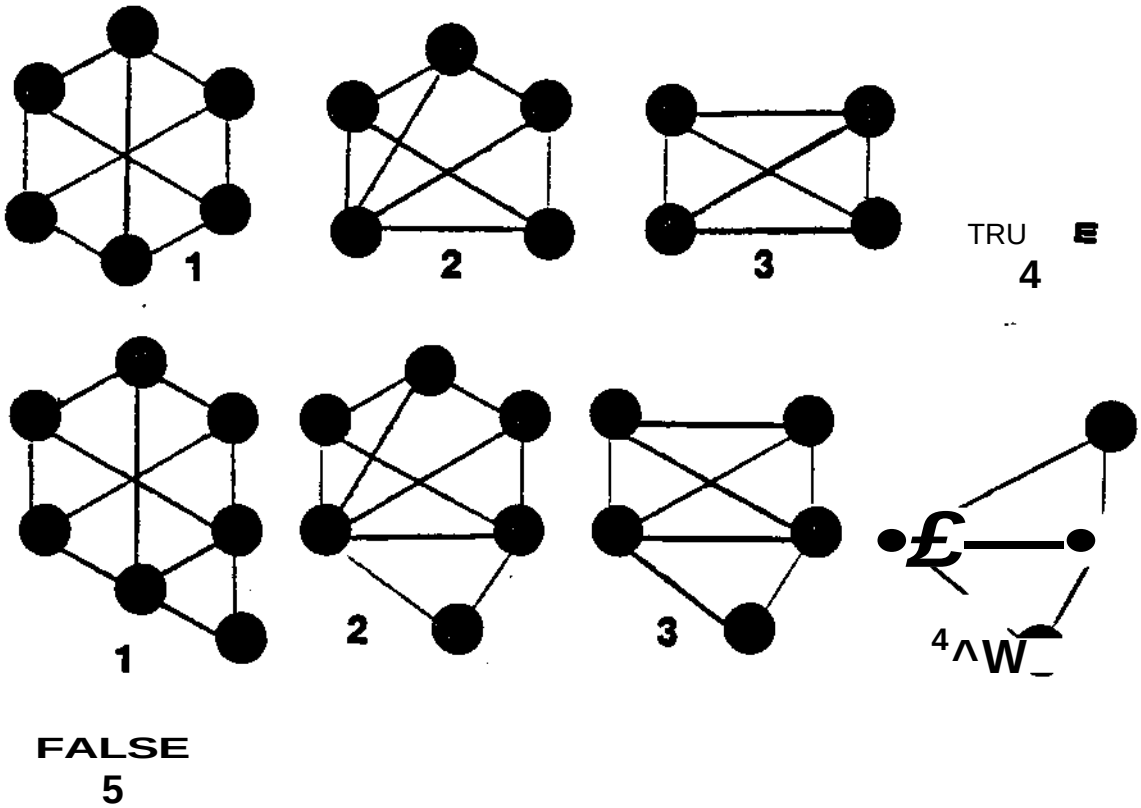



**Figure 3-81:** 3-CONNECTED""[1] in Operation

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-CONNECTED""[1] is correct and K-CONNECTED is complete.

# 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 $Z_2$ with $I_3$- T^s second section explores loop marking and contrasts $£_3$ with $Z_A$- 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 $Z_2$ and £3 is givea

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 = <f,S,a> as a recursive algorithm, called on (G,0) for any graph G described by S:

p(G,k) = (G,k) if enough

s p(G\k+1) where G' = f(g) using elements from G selected

by $\sigma$ 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:

$(A^*_{xy}A_z)^*(<\phi,\phi>,0)$ where distinct $x,y \in V$, $z \notin V$

Figure 4-1 shows the iterative steps in a sample run of VERTICES.

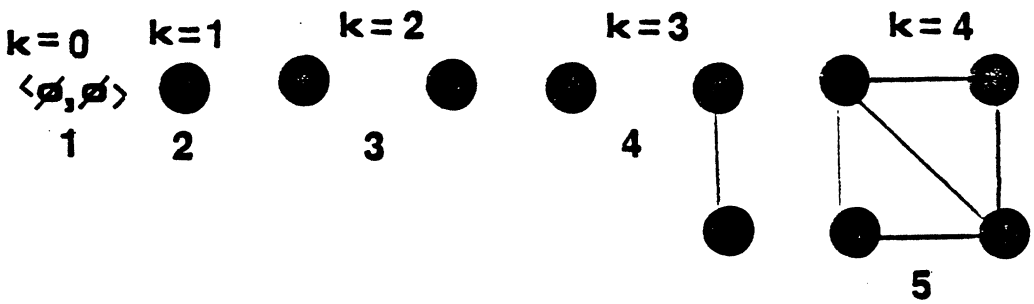



**Figure 4-1:** VERTICES in Operation

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 $<P_1,L_1,\Sigma_1>$.

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:

$(A_{yz}A^*_x)^*(K_1,0)$ where distinct $y,z \in V$, $yz \notin E$

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 $<P_1,L_1,\Sigma_1>$ also.

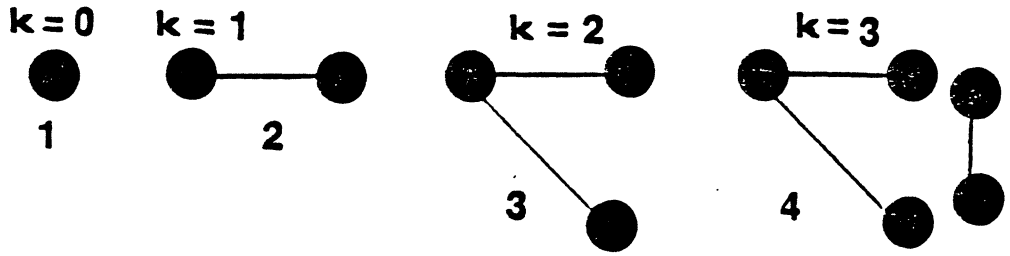



**Figure 4-2:** EDGES in Operation

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 $\sigma^{-1}$ in order to apply $f^{-1}$

= (FALSE,k) if G is not described by S and $\sigma^{-1}$ 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 $\Sigma^{-1}$. 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 $\sigma^{-1}$.

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 $\sigma^{-1}$ 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:

$$
\begin{aligned}
f^{-1} &= (A^*_{xy} A_z)^{-1} \\
&= A_z^{-1} (A^*_{xy})^{-1} \\
&= A_z^{-1} (A_{xy}^{-1})^e \\
&= D_z D^e_{xy} \\
\sigma_{pre} &= \text{distinct } x,y \in V,\ z \notin V,\ xy \notin E \\
\sigma^{-1} &= \text{distinct } x,y,z \in V,\ xy \in E,\ d(x) = 0
\end{aligned}
$$

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:

$$
\begin{aligned}
f^{-1} &= (A_{yz} A^*_x)^{-1} \\
&= (A^*_x)^{-1} A_{yz}^{-1} \\
&= (A_x^{-1})^e A_{yz}^{-1} \\
&= D^e_x D_{yz} \\
\sigma_{pre} &= \text{distinct } y,z \in V,\ x \notin V,\ yz \notin E \\
\sigma^{-1} &= \text{distinct } x,y,z \in V,\ yz \in E,\ d(x) = 0
\end{aligned}
$$

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.

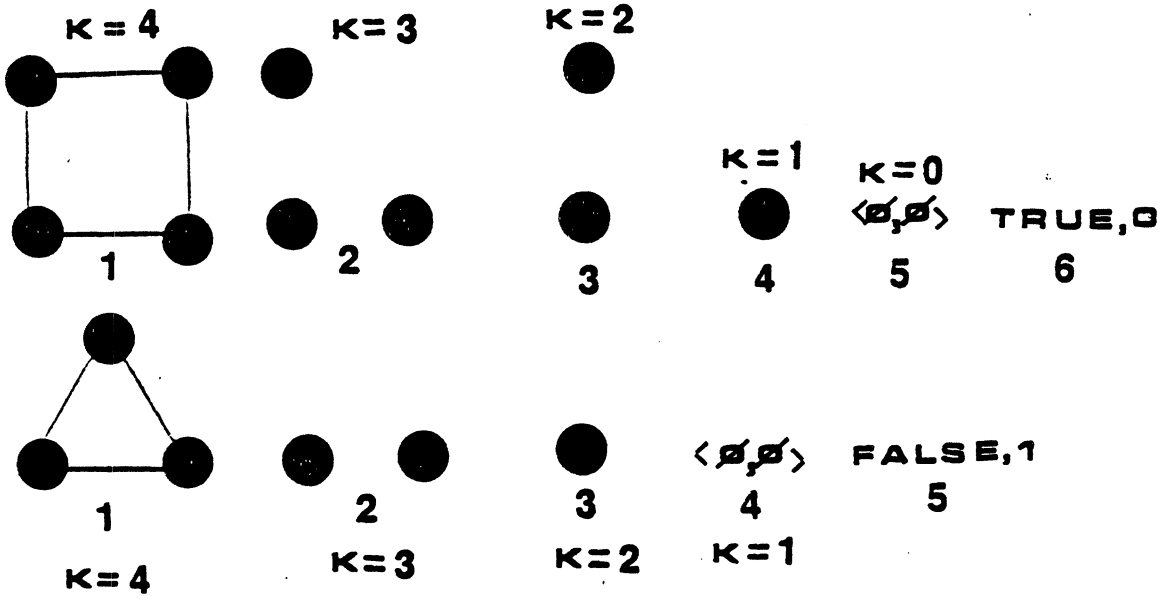



Figure 4-3: VERTICES$^{-1}$ in Operation

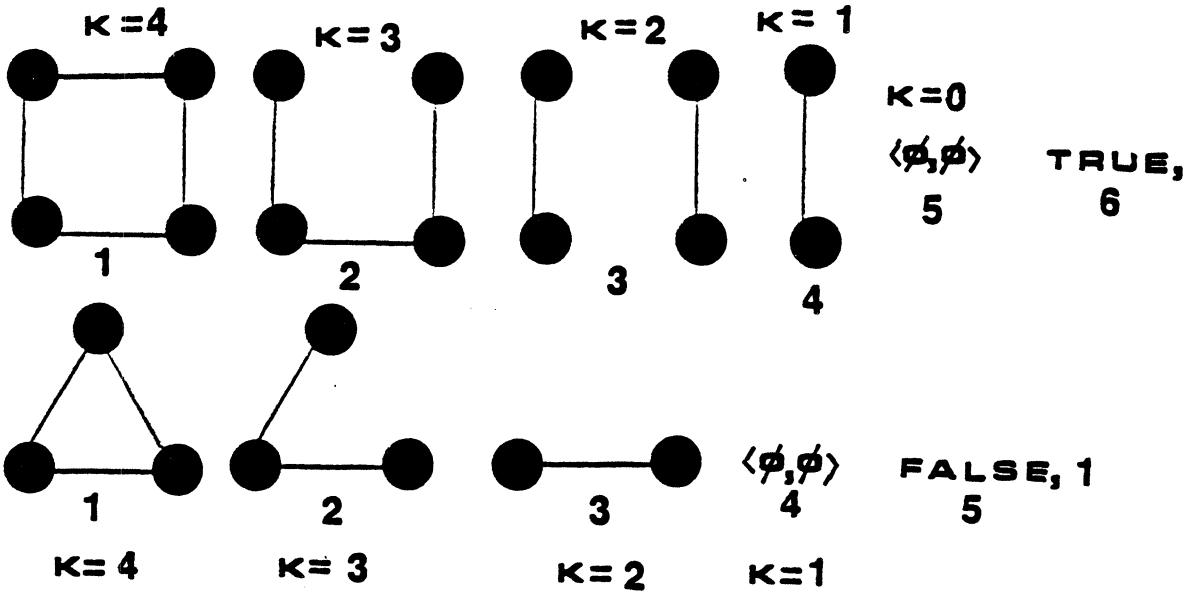



Figure 4-4: EDGES$^{-1}$ in Operation

### 4.1.2. Calculating the Degree of a Vertex

The $R^+$-property "has vertex v of degree k" is the concept used to extend $\Sigma_2$ to $\Sigma_3$. This property, DEGREE, may be stated as:

$(A_{vw}(A_x + A_{yz})^*)^*(<\{v\},\phi>,0)$ where distinct v,w,y,z ∈ V, x ∉ V, vw ∉ 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

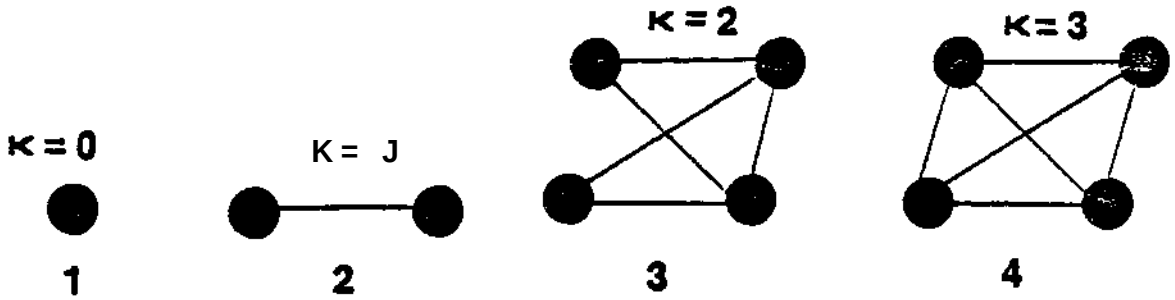



**Figure 4-5: DEGREE in Operation**

edge must involve vertex v. The floor for DEGREE is $<P_r L_r Z_2^>$- Th® inverse. DEGREE"[1], is computed by:

$f\sim^{1} = (A_{vw} (A_x + A_{yz})*f^1$

$_8 ((A_x + A_{yz})*f^1 A\sim^1_{vw}$

$s ((A_{X-} + A_{yz} f V A^{"1}_{vw}$

$= (A_x^{-1} + A_{yz}^{-1})^{e} A_{vw}^{-1}$

$» (D_x + D_{yz})^{e} D_{vw}$

$0_{pre} = 0$ as distinct v,w,y,z € V, x * V, yz,vw *2* E

$a\sim^{1}$ as distinct x,v,w,y,z 6 V, yz,vw e E, d(x) = 0

The floor shifts to $<P_{<5}, L_1, E^>$. 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 *o* e $V_3$ - $V_2$ 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 (FALSER) to mean that v had degree n-k. Of course such calls could be inefficient; it might be more economical to use 0(n) (one register for each vertex) storage and calculate the degree of all vertices in 0(n + m) iterations. As an R-property in $<PX, Z_3^>$ iterates, it could update the degrees of the vertices at 0(i) cost where i is "the number of iterations. In the worst case, 0(m) = $0(n^2)$ and 0(m + n) as $0(n^2)$; in the best case, m = 0 and 0(m + n) = 0(n). Thus an R-property whose floor requires $£_3$ rather than $£_2$ represents an additional complexity of between O(ni) and $O(n^2 i)$ for a graph on n vertices requiring i iterations for

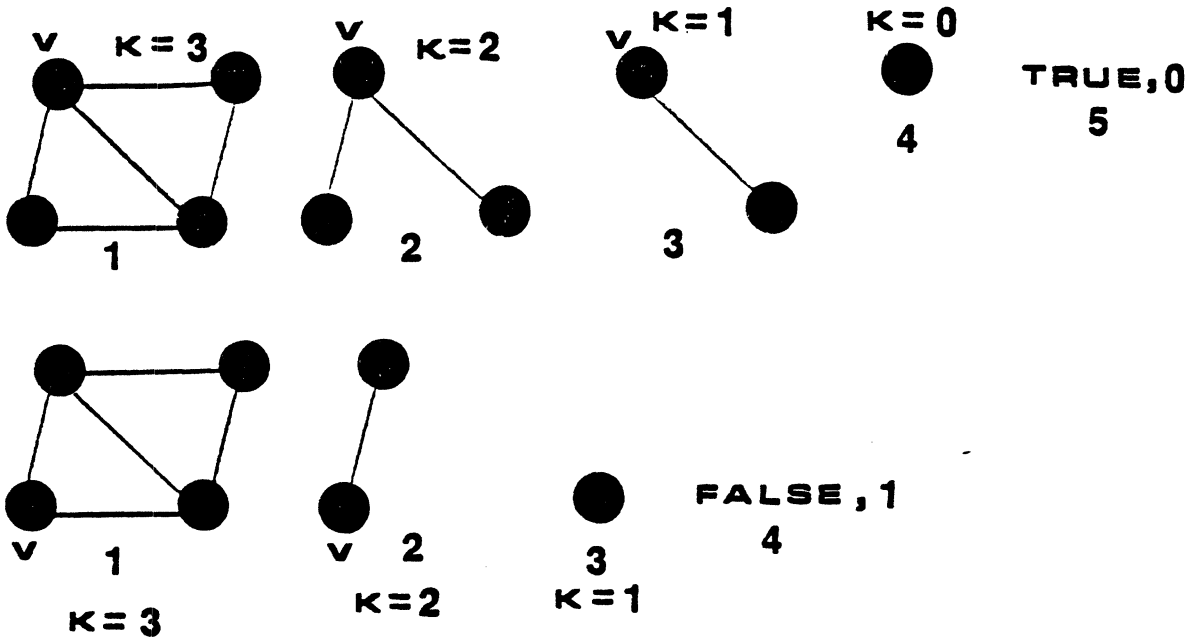



Figure 4-6: DEGREE$^{-1}$ in Operation

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 $\Sigma_3$ and $\Sigma_4$ 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:

$(A_x^*(D_{yy}A_{yz})^{\cap}L)^*(E_p,0)$ where distinct y,z $\in$ V, yy $\in$ E, yz $\notin$ 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

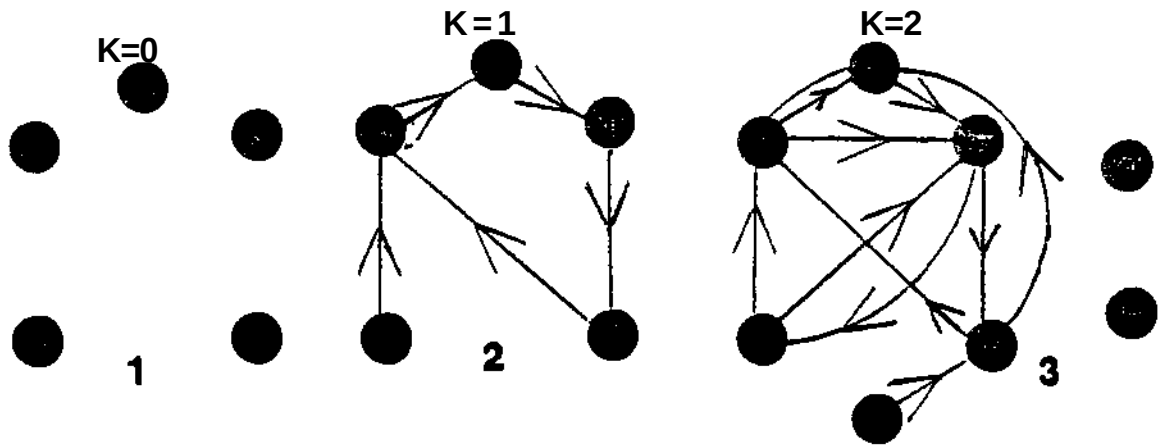



Figure 4-7: MAX in Operation

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 $\langle P2'\hat{}\rangle'\hat{}2\rangle'$ "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:

$$
\begin{aligned}
f^{-1} &= (A_x^*(D_{yy}A_{yz})^n L)^{-1} \\
&= L^{-1}((D_{yy}A_{yz})^n)^{-1}(A_x^*)^{-1} \\
&= L^{-1}((D_{yy}A_{yz})^{-1})^n(A_x^{-1})^e \\
&= L^{-1}(A_{yz}^{-1}D_{yy}^{-1})^n(A_x^{-1})^e \\
&= \underline{U}D_{yz}A_{yy})^n D_x^a
\end{aligned}
$$

$a_{pre}$ = distinct y,z s V, x *ft* V, yy s E, yz 8 E

a"[1] = distinct x.y.z s V, yy *ft* E. yz s E. d(x) = 0

By d(x) = 0, we mean both the in degree and the out-degree of x are zero. The

floor shifts to $<P_3,L_1,\Sigma_3>$. Figure 4-8 shows $MAX^{-1}$ operating on two graphs, one with a correct k value and one with an incorrect k value.

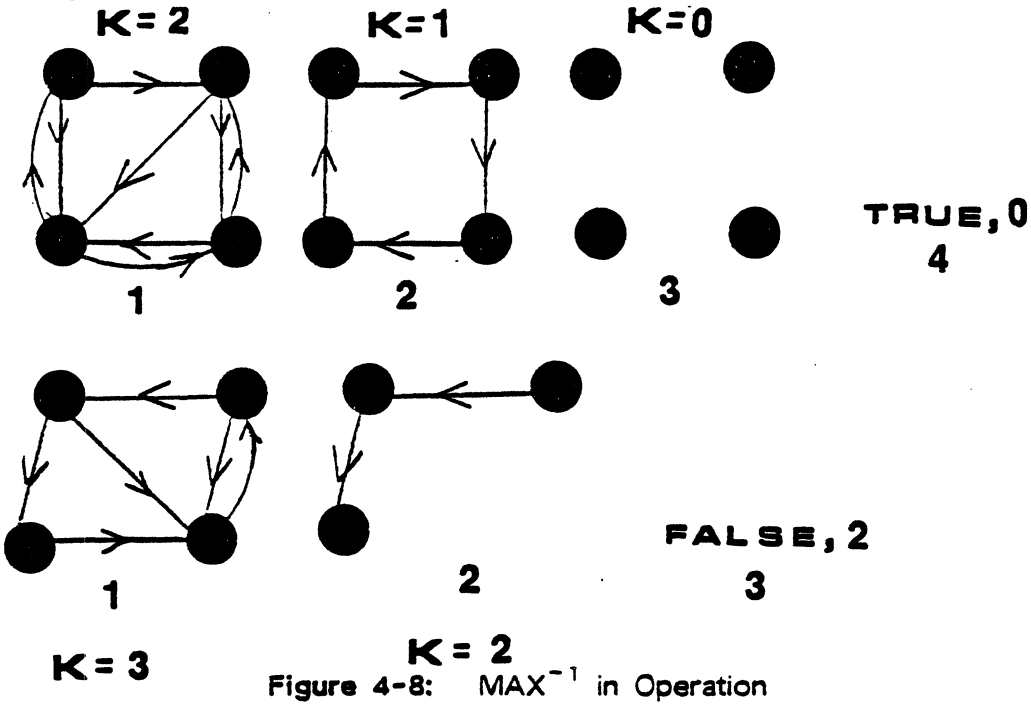



**Figure 4-8:** $MAX^{-1}$ in Operation

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 $\sigma \in \Sigma_4 - \Sigma_3$ 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. $\Sigma_4$ therefore represents an additional complexity

of $O(m + n)$ over $\Sigma_3$.

### 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.

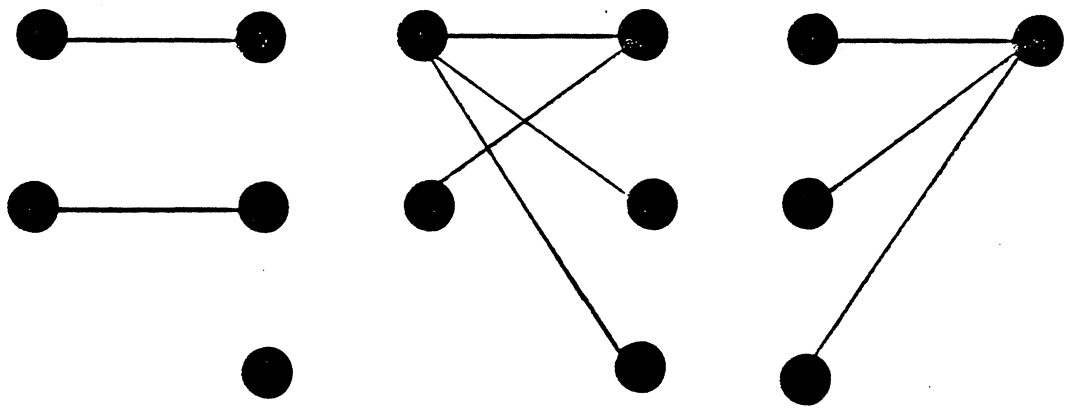


**Figure 4-9:** Some Bipartite Graphs

The R-property BIPARTITE is:

$$(A_x + A_{xx}A_x + A_{yz})^*(<\{1,2\},\{11\}>) \text{ where } y,z \in V,\ x \notin V,$$
$$|\{yy,zz\} \cap E| = 1$$

Figure 4-10 shows the iterative steps in a sample run of BIPARTITE. In $L_1$, $<\{1,2\},\{11\}>$ is characterized by $E \cap \underline{1} = 0$ and $E \cap 1 \neq 0$, but so is any edgeless graph with some loops. In $L_2$ the seed is uniquely characterized as:

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

In $L_{1n}$ the seed is uniquely characterized as $E \cap \underline{1} = 0$, $E \cap 1 \neq 0$ and $n = 2$. Thus the floors for bipartite graphs are $<P_1,L_{1n},\Sigma_5>$ and $<P_1,L_2,\Sigma_5>$. The seed graph makes the partition of the vertices explicit: those vertices with loops are in one

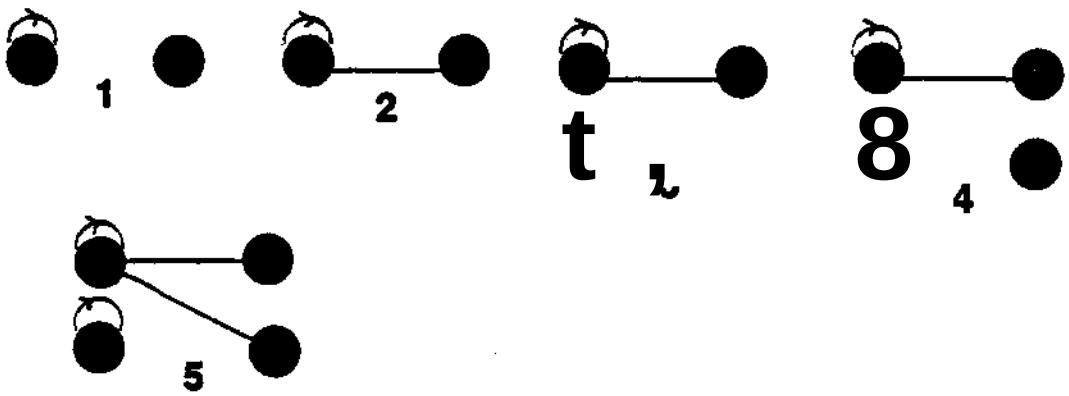



**Figure 4-10: BIPARTITE in Operation**

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:

$$f\sim^{1} \quad a\ (A_x + A_{xx}A_x + A_{yz}\ f^{1}$$
$$a\ A_x\sim^{1} + (A_{xx}A_x f^{1} + A_{yz}''''^{1}$$
$$a\ A_x\sim^{1} + A_x\sim^{1}A_{xx}\sim^{1} + A_{yz}''^{1}$$
$$= D_x + D_xD_{xx} + D_{yz}$$

$$*_{pro}\ a''^{1} = y,z \in V,\ x \in V,\ yz \in E,\ |\{yy,zz\} \cap E| = 1$$
$$\bullet\ x.y,z\ s\ V,\ yz\ a\ E,\ |\{yy,zz\} \cap E| * 1,\ d(x) < 1,$$
$$|\{p|p \bullet V,\ d(p)=0\}| > 1,$$
$$|\{p|p\ a\ V,\ pp\ s\ E,\ d(p) < 1\}| > 1$$

The floors shift to $<P_{22}, L_{11}, E_5>$ and $<P_2, L_2, E_5>$. Figure 4-11 shows BIPARTITE$''^{1}$ operating on a graph $G\ s\ G_p$ and a graph $G < G_p$. 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

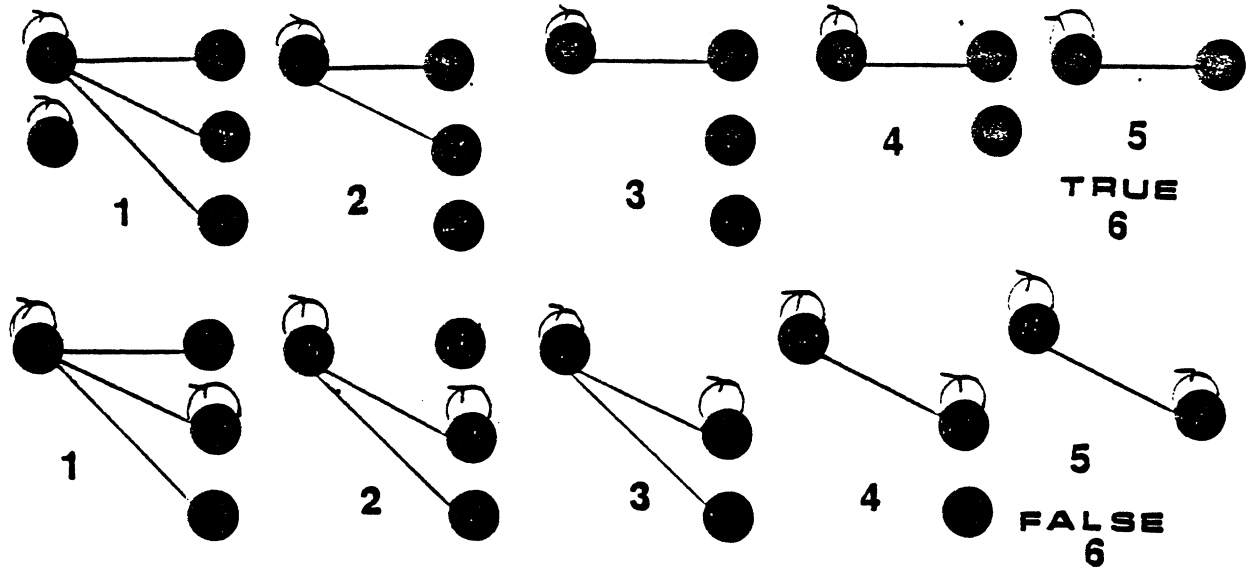



**Figure 4-11:** BIPARTITE$^{-1}$ in Operation

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:

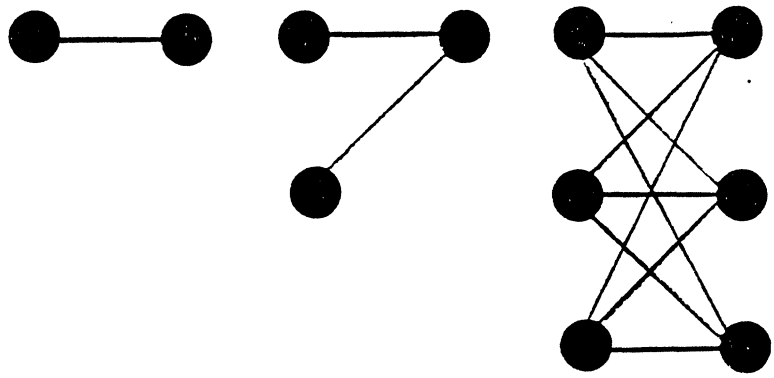


**Figure 4-12:** Some Complete Bipartite Graphs

$(A_{vx_1}...A_{vx_p}A_v + A_{wy_1}...A_{wy_q}A_{ww}A_w)^*(<\{1,2\},\{11,12\}>)$ where

distinct $x_i \in V$, $v \notin V$, $x_ix_i \in E$, $i=1,2,...,p$; $|\{z \mid x \in V, xx \in E\}| = p$

distinct $y_i \in V$, $w \notin V$, $y_iy_i \notin E$, $i=1,2,...,q$; $|\{z \mid y \in V, yy \notin E\}| = q$

Figure 4-13 shows the iterative steps in a sample run of COMPLETE-BIPARTITE.

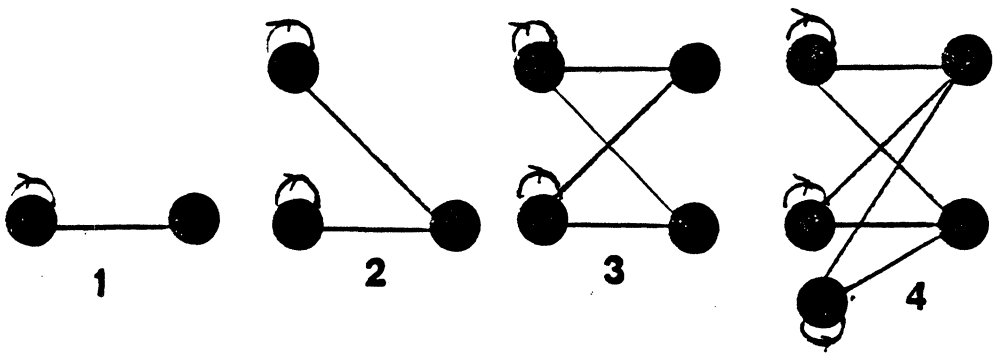



**Figure 4-13:** COMPLETE-BIPARTITE in Operation

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 $<P_1,L_{1n},\Sigma_5>$ and $<P_1,L_2,\Sigma_5>$.

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:

$$f^{-1} = (A_{vx_1}...A_{vx_p}A_v + A_{wy_1}...A_{wy_q}A_{ww}A_w)^{-1}$$

$$= (A_{vx_1}...A_{vx_p}A_v)^{-1} + (A_{wy_1}...A_{wy_q}A_{ww}A_w)^{-1}$$

$$= D_vD_{vx_p}...D_{vx_1} + D_wD_{ww}D_{wy_q}...D_{wy_1}$$

$\sigma_{pre} = \sigma$ = distinct $x_i \in V$, $v \notin V$, $x_ix_i \in E$,

$$|\{z \mid x \in V,\ xx \in E\}| = p$$

distinct $y_i \in V$, $w \notin V$, $y_iy_i \notin E$,

$$|\{z \mid y \in V,\ yy \notin E\}| = q$$

$\sigma^{-1}$ = distinct $v,x_i \in V$, $xx_i, vx_i$

$\in E$, $i=1,2,....,p$;

$$|\{z \mid z \in V,\ zz \in E\}| = p,$$

$$|\{z \mid z \in V,\ zz \notin E\}| > 1,\ d(v) = p$$

distinct $w,y_i \in V$, $wy_i \in E$,

$y_iy_i \notin E$, $i=1,2,....,q$,

$$|\{z \mid z \in V,\ zz \notin E\}| = q,$$

$$|\{z \mid z \in V,\ zz \in E\}| > 1,\ d(w) = q$$

Again the floors shift to $\langle P_2, L_{1n}, \Sigma_5 \rangle$ and $\langle P_2, L_2, \Sigma_5 \rangle$. Figure 4–14 shows COMPLETE–BIPARTITE$^{-1}$ operating on a graph $G \in G_p$ and a graph $G \notin G_p$. 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.

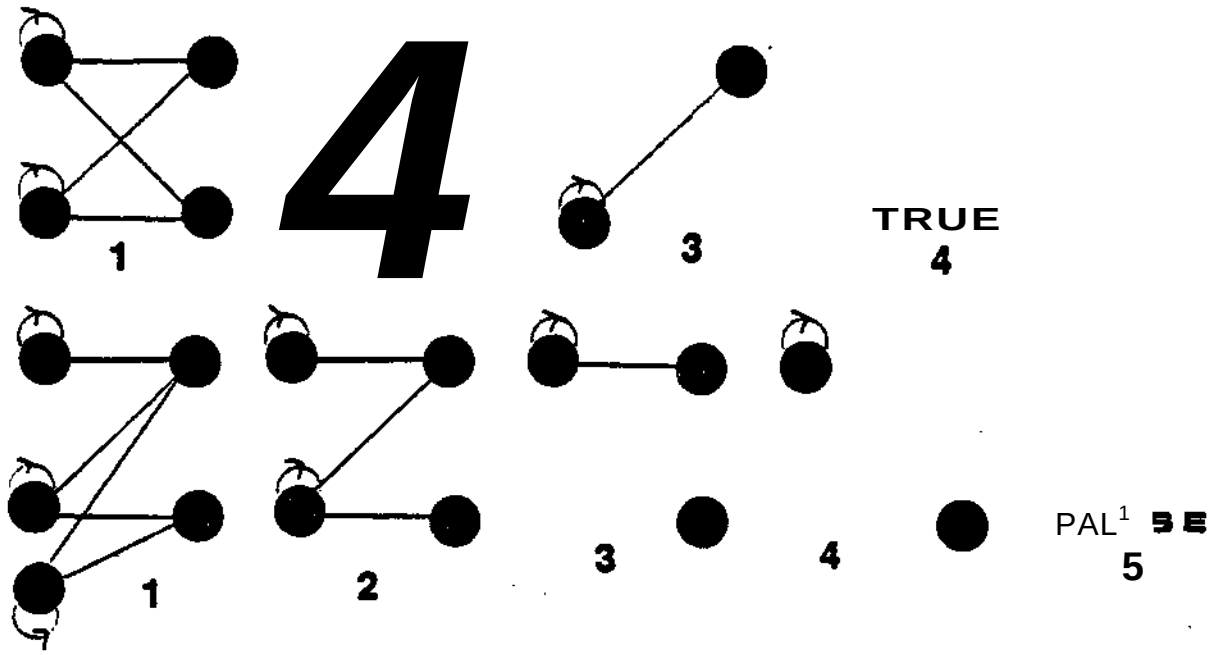



Figure 4-14: COMPLETE-BIPARTITE$^{-1}$ in Operation

4.3.3. K-Vertex-Covered Graphs

A vertex x *covers an edge* yz if x is y or i Given a graph G = <V,E>, a set of vertices A £ 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-cover able.* If a graph G is k-vertex-coverabie, 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.

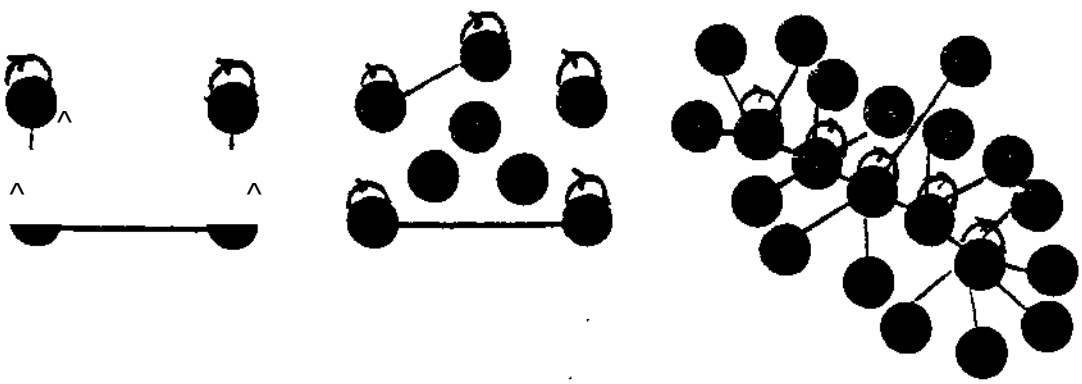


Figure 4-15: Some 5-Vertex-Covered Graphs

The R-property K-VERTEX-COVERED is

$(A_x + A_{yz})^*(LE_k)$ 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.

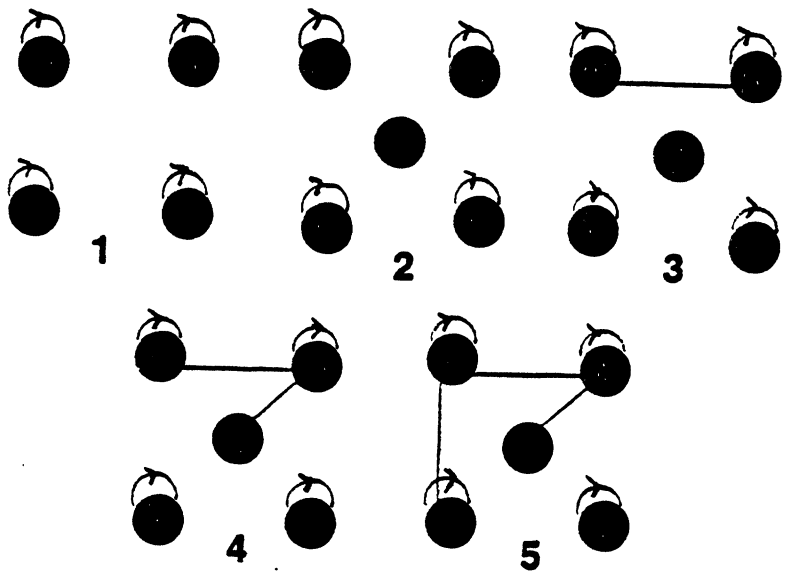



**Figure 4-16:** 4-VERTEX-COVERED in Operation

The seed is a graph on k vertices with no edges and all possible loops, which we have abbreviated as $LE_k$:

$LE_k = <\{1,2,\ldots,k\},\{11,22,\ldots,kk\}>$

The set of all $LE_k$'s is described in $L_1$ as

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

and

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

but in order to distinguish a particular $LE_k$ we require $L_{1n}$. Thus the floor for K-VERTEX-COVERED is $<P_1,L_{1n},\Sigma_5>$. 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:

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

$= A_x^{-1} + A_{yz}^{-1}$

$= D_x + D_{yz}$

$o_{pre} = x \ll V$

$y,z \in V.\ y \neq z,\ yz \ll E \mid \{yy.zz\} \cap E \mid \neq 0$

$a^{-1} = x \in V,\ xx \ll E.\ d(x) = 0$

$y,z \in V,\ y \neq z.\ yz \bullet E,\ \mid \{yy,zz\} \cap E \mid \neq 0$

The floor shifts to $\langle p_2, L_{1n}, 2_5 \rangle$. Figure 4-17 shows K-VERTEX-COVERED$^{-1}$ operating on a graph $G \in G_p$ and a graph $G \ll G_p$ for k = 3.

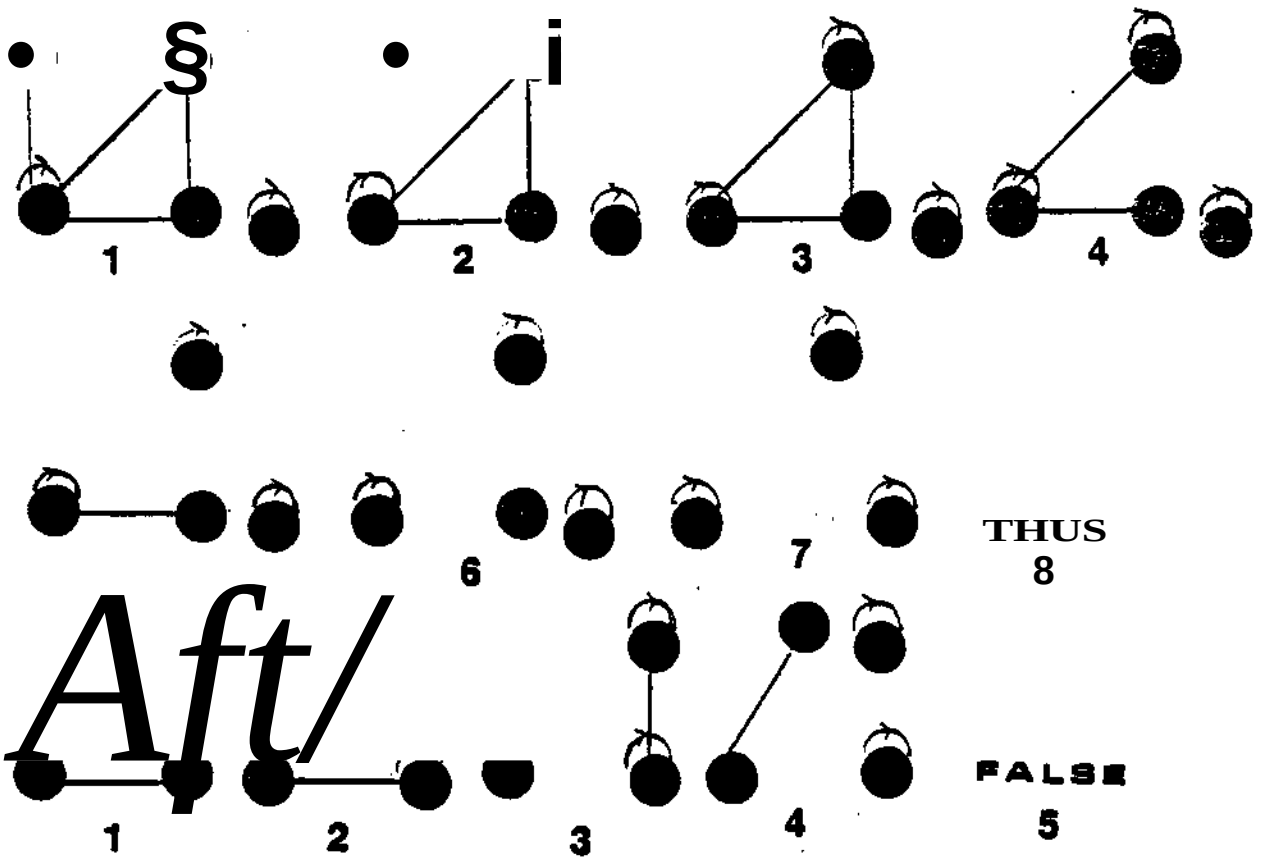



**Figure 4-17:** 3-VERTEX-C0VERED~$^{1}$ in Operation

Just as for BIPARTITE$^{-1}$, K-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-VERTEX-COVERED$^{-1}$ is correct and K-VERTEX-COVERED is complete.

### 4.3.4. Graphs with K Independent Vertices

Given a graph G = <V,E>, 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.

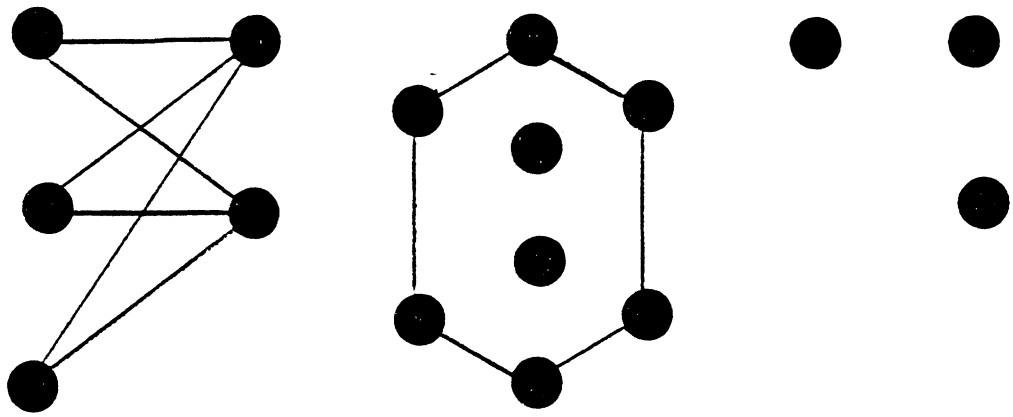


**Figure 4-18:** Some Graphs with 3 Independent Vertices

The R-property K-INDEPENDENT is

$(A_x + A_{yz})^*(LE_k)$ 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.

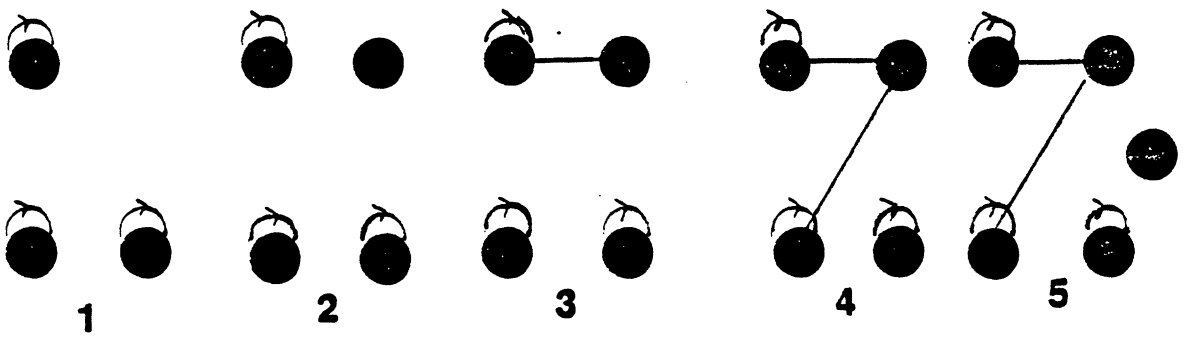



**Figure 4-19:** 3-INDEPENDENT in Operation

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:

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

$$\sigma_{pre} = x \notin V$$
distinct $y,z \in V$, $yz \notin E$, $|\{yy,zz\} \cap E| \leq 1$

$$\sigma^{-1} = x \in V,\ d(x) = 0$$
distinct $y,z \in V$, $yz \in E$, $|\{yy,zz\} \cap E| \leq 1$

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 $G \notin G_p$. On a graph from $G_p$ all edges and unlooped vertices will be deleted. On a graph $G \notin G_p$, 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 = <V,E> be a graph and let c be a function defined on V, i.e., c(v) is defined and unique for each v ∈ V. Then we say that c is a *labelling* of G, that

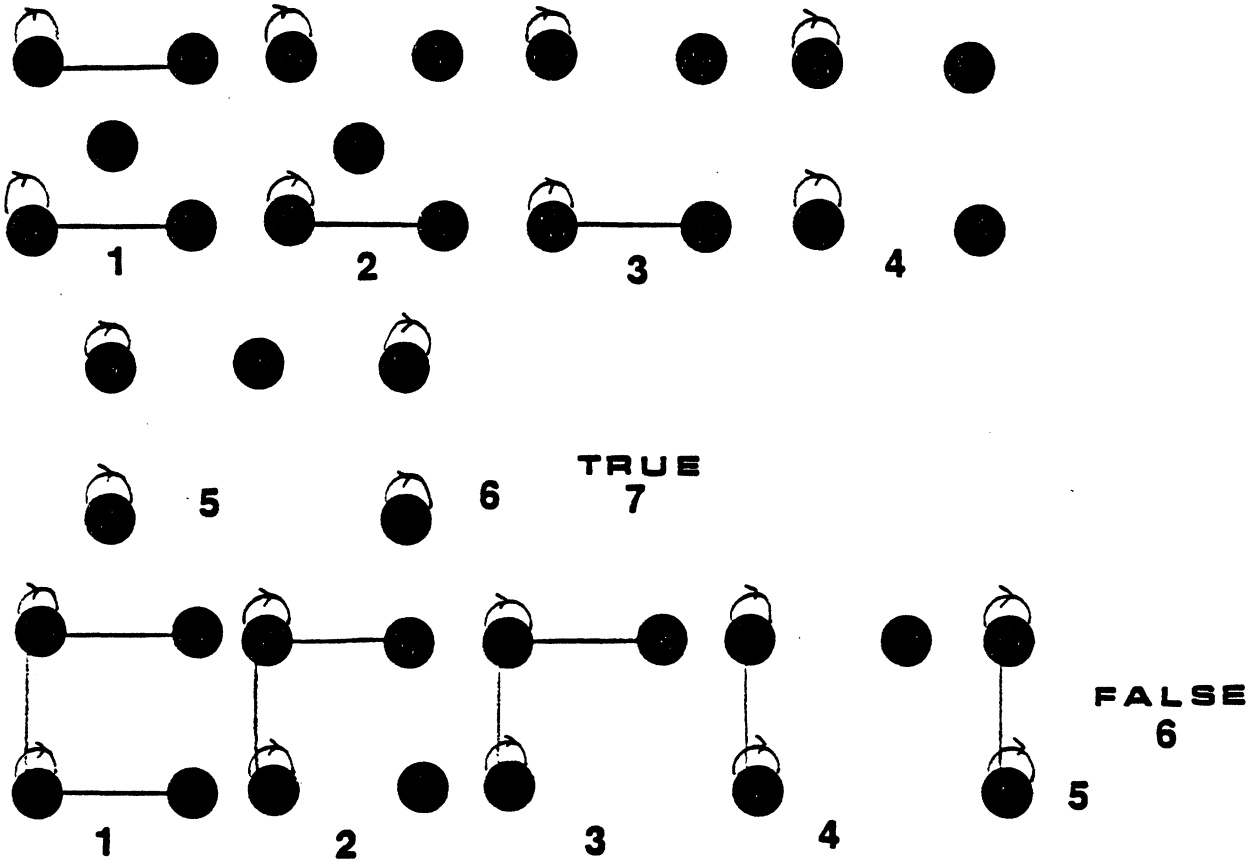



**Figure 4-20:** 2-INDEPENDENT$^{-1}$ in Operation

the range of c, $\Lambda = \{c(v) \mid v \in V\}$, is the set of *labels* for G, and that G = <V,E,c,Λ> 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->Λ 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 = <f,S,a> had its origin in the ordered triple <P.LE>. 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 a. $Z_{xa}(G)$ will set c(x) to a, leaving the remainder of the graph unchanged More formally, we define the primitive operator sets:

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 $L_Q$ 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:

- I -> labels are unique | labels are not unique
- | -> labels range from 1 to k

The first, appended to the languages $L_1$ and $L_{jn}$ will yield the $L_c$-languages $L_{iuc}$ and hnuc *or i s 1,2,~'6- Ths second appended to the languages $L_1$ and $L_n$, will yield the $L_c$-languages $L_{ic}$ and $L_{inc}$ for i « 1,2,~.,6.

Finally, *2* must be augmented to test colors, as well as vertices and edges. We augment the original *2* grammars with the following:

| | |
|---|---|
| I | -> c(vertex) * c(vertex) \| c(vertex) # ctvertex) \| |
| | -> 1 £ color £ k \| color even j color odd |
| color | -> a \| p \| 7 \| _ |

The expression # will be interpreted semantically as "is different from" and the expression = as "is identical to." By appending these forms to the languages $E_1$ through $Z_6$ we produce the $E_c$-*languages* $E_{1c}$ through $E_{6c}$, respectively. We now can formally define an *$R^c$-property* as the semantic interpretation of the triple

<f,S,σ>, where f is a terminal $P_c$-expression, S is a terminal $L_c$-expression and σ is a terminal $\Sigma_c$-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.

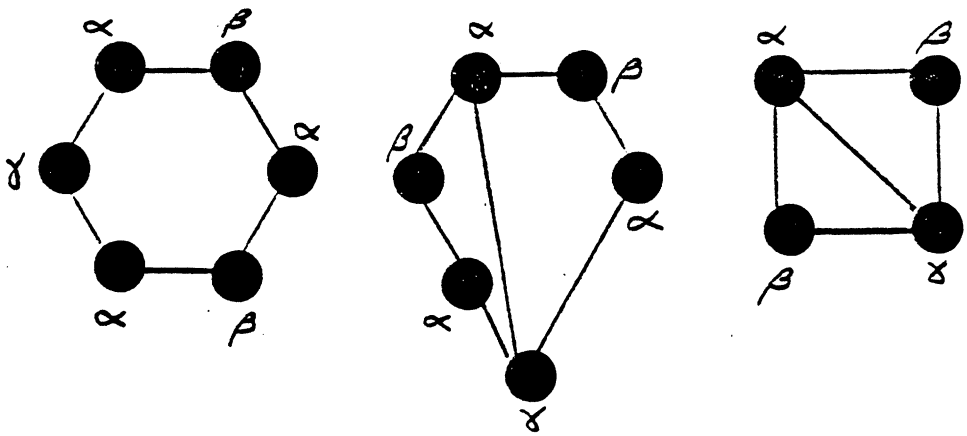



**Figure 4-21:** Some 3-Colored Graphs

The R-property K-COLORED is

$$(A_{xy} + Z_{z\alpha}A_z)^*(U_k) \text{ where } x,y \in V,\ c(x) \neq c(y)$$
$$z \notin V,\ 1 \leq \alpha \leq k$$

The seed is $U_k$, the uniquely colored edgeless graph on k different-colored vertices:

$$U_k = Z_{v_1 1}Z_{v_2 2}\cdots Z_{v_k k}E_k$$

Note that our "colors" are really integers between 1 and k, inclusive. $U_k$ is in $L_{1nu}$. Figure 4-22 shows the iterative steps in a sample run of K-COLORED for k = 4. The floor for K-COLORED is $<P_{1c},L_{1nuc},\Sigma_{1c}>$.

Clearly K-COLORED is correct: the only edges it adds are between

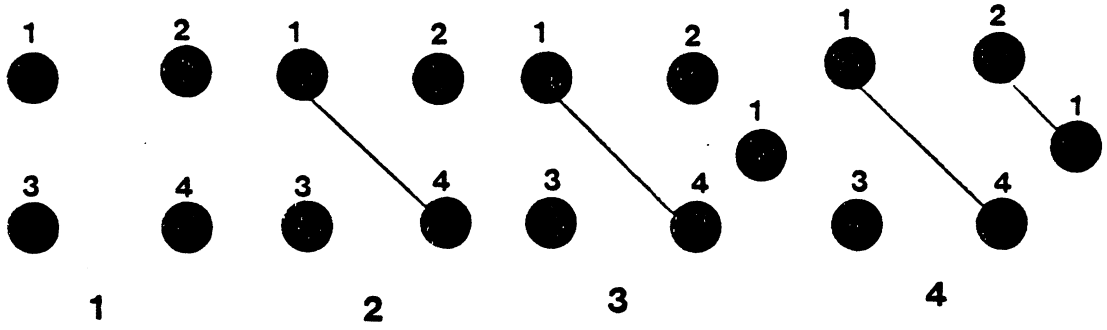



**Figure 4-22:** 4-COLORED in Operation

different-colored vertices, and each vertex is assigned a color when it is added to the graph.

The automatic inversion of an $R^c$-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 $\lambda$, 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-COLORED$^{-1}$ is computed from:

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

$$\begin{aligned} \sigma_{pre} &= x,y \in V,\ xy \notin E,\ c(x) \neq c(y) \\ &\quad z \notin V,\ 1 \leq \alpha \leq k \end{aligned}$$

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

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-COLORED$^{-1}$ operating on a graph $G \in G_p$ and a graph G

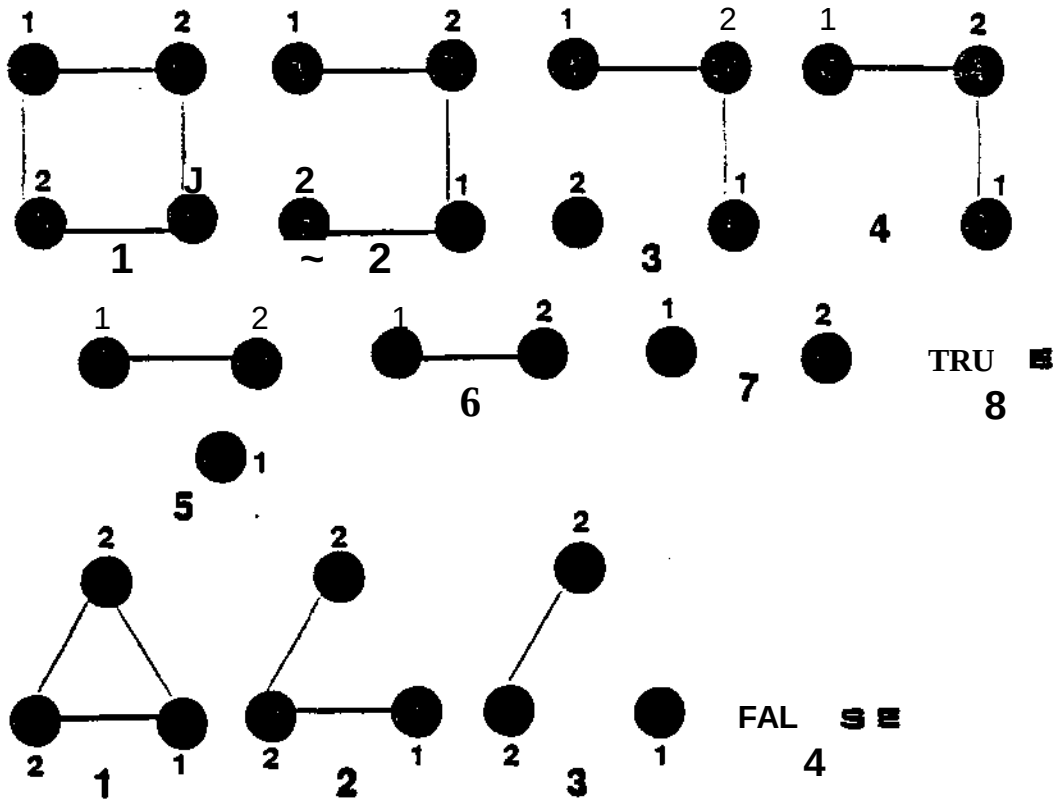



**Figure 4-23:** 2-COLORED"[1] in Operation

* $G_p$ for k = 2. On a correctly-labelled k-colored graph, the edges will be removed one at at 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 * $G_p$ 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 ≥ 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-colorabie 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

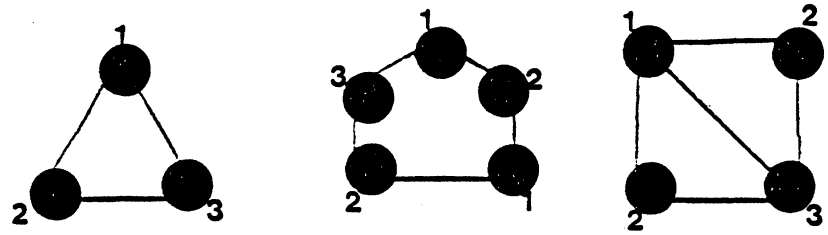



**Figure 4-24:** Some 3-Chromatic Graphs

operators, $S_{xvwy}$ and $X_{v_1s_1\ldots v_rs_r}$. $S_{xvwy}$ 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. $X_{v_1s_1\ldots v_rs_r}$ is an exchange operator which introduces similarly-labelled surrogate vertices $s_1,s_2,\ldots,s_r$ for the vertices $v_1,v_2,\ldots,v_r$, respectively. $X_{v_1s_1\ldots v_rs_r}$ 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 $s_j$. $X_{v_1s_1\ldots v_rs_r}$ also appends correctly-labelled vertices $s_1,s_2,\ldots,s_r$ to G:

$$X_{v_1s_1\ldots v_rs_r} = D_{v_iv_j}A_{s_iv_j}A_{v_is_j}A_{s_is_j}Z_{s_1c(v_1)}Z_{s_2c(v_2)}\ldots Z_{s_rc(v_r)}A_{s_1}A_{s_2}\ldots A_{s_r}$$

where $D_{v_iv_j}A_{s_iv_j}A_{v_is_j}A_{s_is_j}$ occurs for each $v_iv_j \in E$.

The $R^c$-property K-CHROMATIC] is

$$(Z_{x\alpha}A_x + A_{yz} + A_{tw_1}A_{tw_2}\ldots A_{tw_{k-3}}A_{uw_1}A_{uw_2}\ldots A_{uw_{k-3}}Z_{pc(w)}Z_{qc(t)}S_{tpqu} + X_{v_1s_1\ldots v_rs_r})^*(T_k)$$

where $x \notin V$, $1 \le \alpha \le k$,

$y,z \in V$, $c(y) \neq c(z)$

distinct $t,u,w_i \in V$, $p,q \notin V$, $tu \in E$, distinct $c(t),c(u),c(w_i)$

distinct $v_i \in V$, distinct $s_i \notin V$, $r \ge k-1$

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,

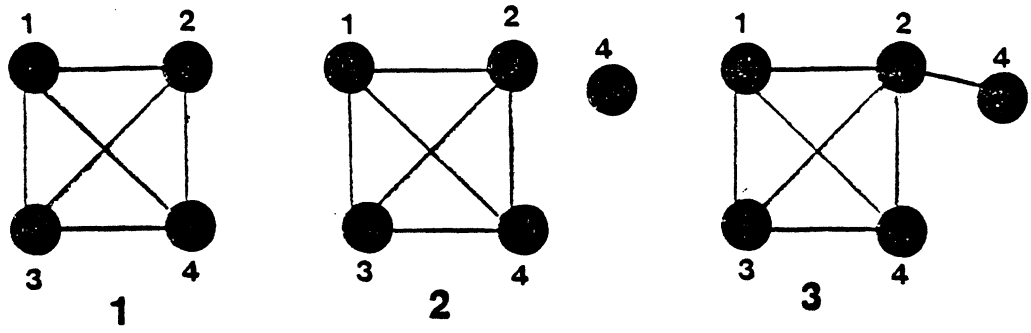



**Figure 4-25:** 4-CHROMATIC in Operation

$Z_{x\alpha}A_x$ 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.

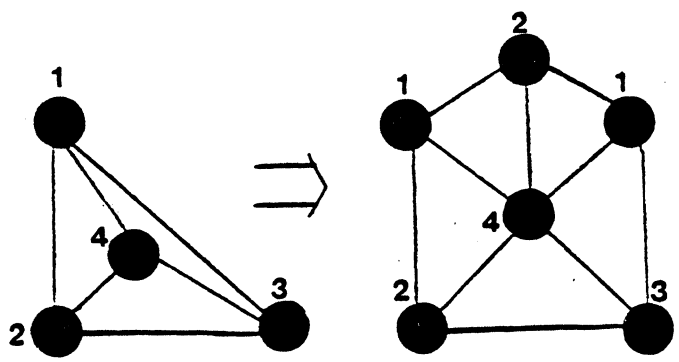



**Figure 4-26:** The Generation of $W_{1,5}$

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 $\{v_i\}$

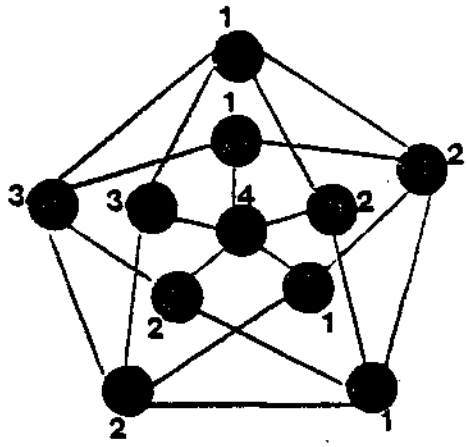



**Figure 4-27:** The Grotzsch Graph

is not necessarily all of V, it is possible to obliterate many, or even all, triangles. In particular, the Grotzsch graph is constructabie from $W_{1,5}$ by this technique, taking the $\{v_i\}$ to be the rim vertices. (See Figure 4-28.)

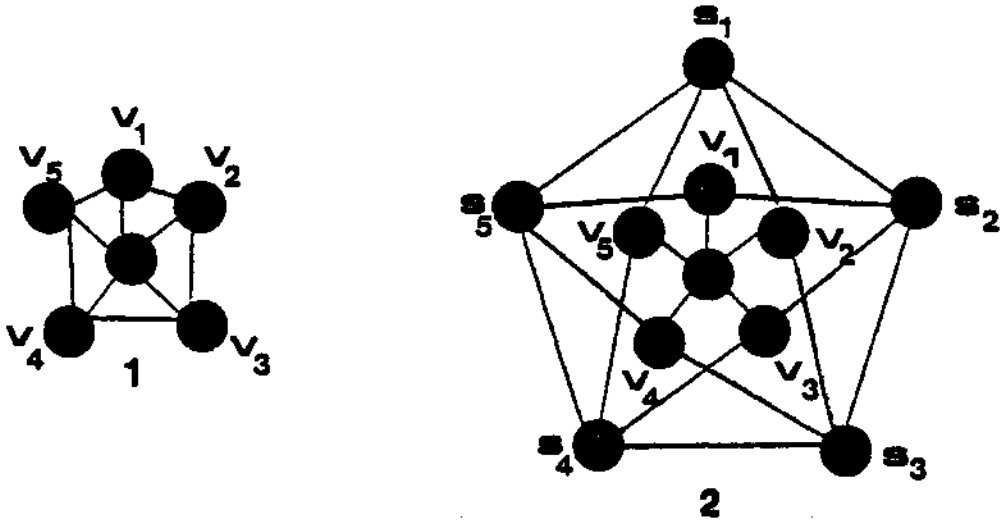



**Figure 4-28:** Generating the Grotzsch Graph

Having explained the motivation for K-CHROMATIC, we now demonstrate that it is correct If any k-1 coloring were possible for $Z_{xflC}A_xG$ 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-colorabla 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 $c(s_i) = c(v_i)$ 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 $v_1v_2 \in E$. Note, however, that $c(s_1)$ and $c(s_2)$ must still be distinct because $s_1s_2$ 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:

$$f^{-1} = (Z_{x\alpha}A_x + A_{yz} + A_{tw_1}A_{tw_2}\ldots A_{tw_{k-3}}A_{uw_1}A_{uw_2}\ldots A_{uw_{k-3}}Z_{pc(w)}Z_{qc(t)}S_{tpqu} + X_{v_1s_1\ldots v_rs_r})^{-1}$$

$$= (Z_{x\alpha}A_x)^{-1} + A_{yz}^{-1} + (A_{tw_1}A_{tw_2}\ldots A_{tw_{k-3}}A_{uw_1}A_{uw_2}\ldots A_{uw_{k-3}} Z_{pc(w)}Z_{qc(t)}S_{tpqu})^{-1} + X^{-1}_{v_1s_1\ldots v_rs_r}$$

$$= D_xZ_{x\lambda} + D_{yz} + D_{qu}D_{pq}D_{tp}A_{tu}D_qD_pZ_{q\lambda}Z_{p\lambda} D_{uw_{k-3}}D_{uw_{k-4}}\ldots D_{uw_1}D_{tw_{k-3}}D_{tw_{k-4}}\ldots D_{tw_1} + D_{s_r}D_{s_{r-1}}\ldots D_{s_1}Z_{s_r\lambda}Z_{s_{r-1}\lambda}\ldots Z_{s_1\lambda}D_{s_is_j}D_{v_is_j}D_{s_iv_j}A_{v_iv_j}$$

$\sigma_{pre}$ = $x \notin V$, $1 \leq \alpha \leq k$

$y,z \in V$, $yz \notin E$, $c(y) \neq c(z)$

distinct $t,u,w_i \in V$, $p,q \notin V$, $tu \in E$, distinct $c(t),c(u),c(w_i)$

distinct $v_i \in V$, distinct $s_i \notin V$, $r \geq k-1$

$\sigma^{-1}$ = distinct $x,x' \in V$, $d(x) = 0$, $1 \leq \alpha \leq k$, $c(x') = c(x)$

$y,z,y',z' \in V$, $yz,y'z' \in E$, $c(y) \neq c(z)$, $c(y') = c(y)$, $c(z') = c(z)$

distinct $p,q,t,u,w_i \in V$, distinct $c(t),c(u),c(w_i)$; $c(w) = c(p)$, $c(q) = c(t)$, $d(p) = k-1$, $d(q) = k-1$; $tp,pq,qu \in E$, $tu \notin E$

distinct $v_i,s_i \in V$, $c(v_i) = c(s_i)$, $s_is_j,s_iv_j,v_is_j \in E$, $v_iv_j \notin E$,

$c(v_i) \neq c(v_j)$

The floor shifts to $\langle P_{2c}, L_{1nuc}, \Sigma_{3c} \rangle$. Figure 4-29 shows K-CHROMATIC$^{-1}$ operating on a graph $G \in G_p$ and a graph $G \notin G_p$ for k = 2.

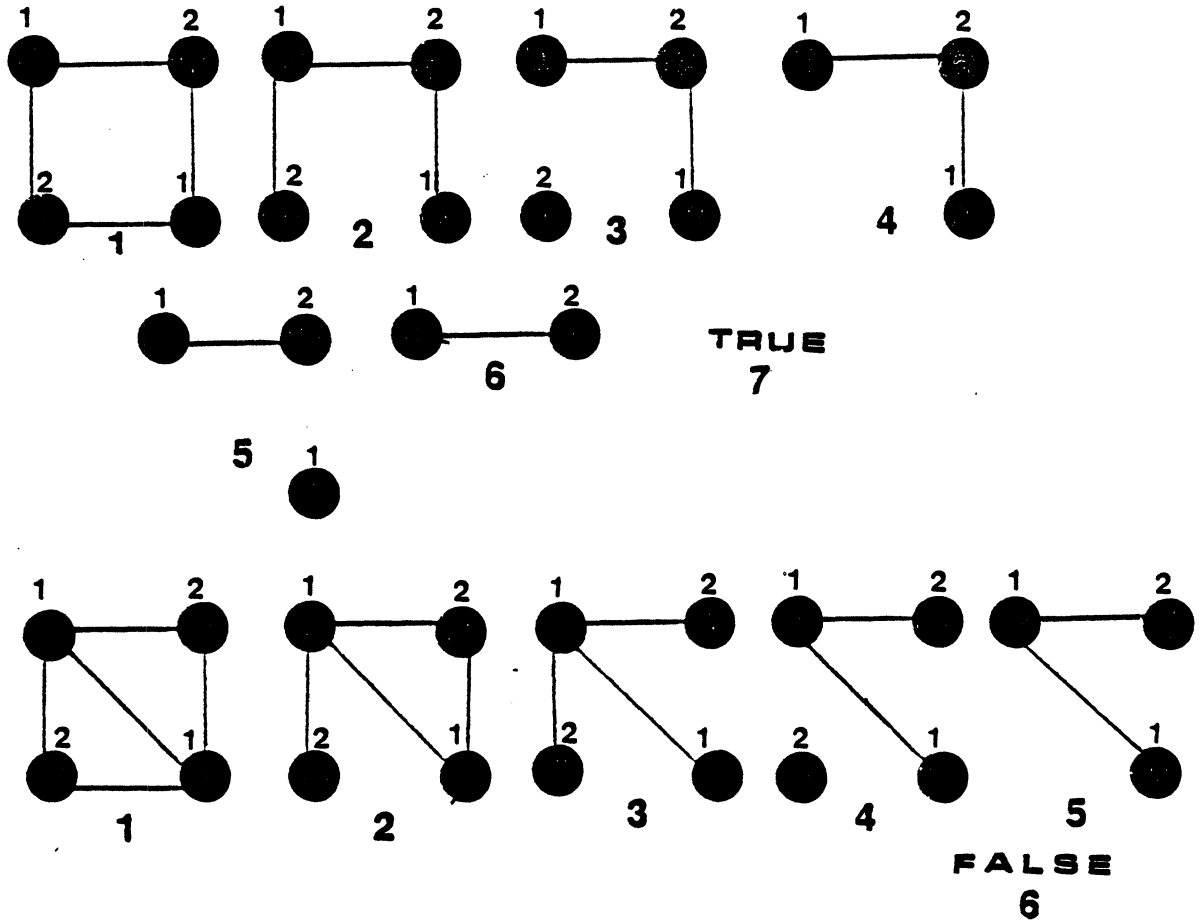



**Figure 4-29:** 2-CHROMATIC$^{-1}$ in Operation

If $G \notin G_p$ because some edge has endpoints of the same color or because G has the wrong number of colors, K-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-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 ($I_{tp}D_{tp}$ and $I_{uq}D_{uq}$) and since the inverse of the surrogate exchange process $X_{v_1s_1\ldots v_rs_r}$ is a sequence of r elementary edge contractions ($I_{v_is_i}D_{v_is_i}$), 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 = <V,E> 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 = <V,E>, G' = <V',E'> 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 $c_r c_2,\ldots,c_r$, the vertex covering number of G is $E[_{as1}\ c_{..}$

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 (hg):

$$^{(CZ}vc(x)^{S}xvy \ + \ ^{Z}zc(w)^{B}wz \ + A_{pq}]^{*}[Z_{t_1c(r_1)}Z_{s_1c(r_1)}(A_{s_1s_1} + A_{t_1t_1})B_{s_1t_1}B_{r_1s_1}$$
$$+ Z_{t_2c(r_2)}Z_{s_2c(r_2)}A_{s_2s_2}B_{s_2t_2}B_{r_2s_2} + Z_{bc(a)}Z_{dc(a)}A_{bb}S_{abde}$$

$$+ \ ^{2}gc<f>^{Z}hc<f,^{A}{}_{g3}{}^{A}hf^{B}{}_{g}h^{B}fg \ + \ ^{Z}jc(i,^{A}iJ^{B}ii \ + \ \backslash \ ^{\wedge} \ \backslash \ ^{\wedge} \ V^{\ o\ )}$$

where x,y • V, v *2 V,* xy,xx,yy s E, c<x) = c(y)

w 6 V, 2 * V, ww e E

p,q s V, |{pp,qq} n E | ≥ 1, c(p) = c(q>

$r_1$ s V, distinct $s_r t_1$ * V, r^ s E

$r_2$ e V, distinct $s_2,t_2$ * V, $r_2 r_2$ * E

distinct a,e e V, distinct b,d £ V, ae,aa,ee e E

f 6 V, distinct g,h * V, ff • E

i e V, j * V, |V| = 1, |E| = 0

distinct $u_r ii_2,u_3$ • V, u ^ u ^ u ^ 6 *E,* |V| =3, |E| =2

The seed is the uniquely colored complete graph $T_1$. The floor for VERTEX-COVER is <P2c/Licu/^5c>* Fi9ure 4-31 shows the iterative steps in a sample run of VERTEX-COVER (All of the labels are identical and omitted)

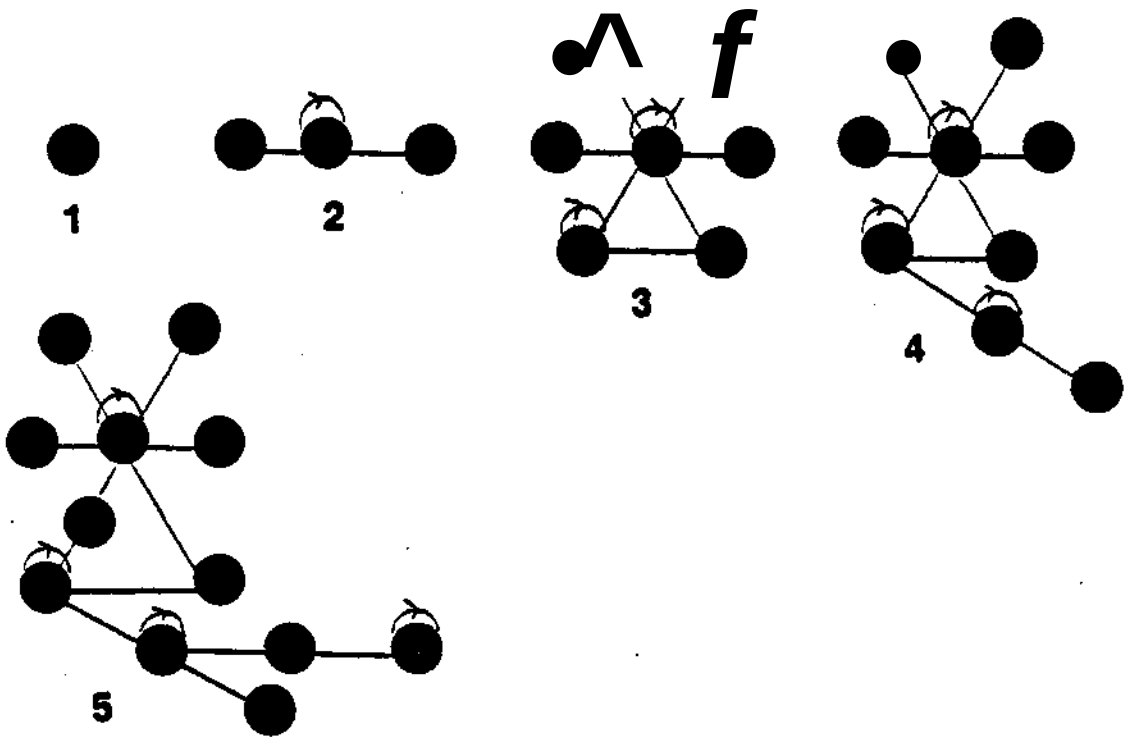



**Figure 4-31:** VERTEX-COVER in Operation

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 e). 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:

$$
\begin{aligned}
f^{-1} &= ([Z_{vc(x)}S_{xvy} + Z_{zc(w)}B_{wz} + A_{pq}]^*[Z_{t_1c(r_1)}Z_{s_1c(r_1)}(A_{s_1s_1} + \\
&\quad A_{t_1t_1})B_{s_1t_1}B_{r_1s_1} + Z_{t_2c(r_2)}Z_{s_2c(r_2)}A_{s_2s_2}B_{s_2t_2}B_{r_2s_2} + \\
&\quad Z_{bc(a)}Z_{dc(a)}A_{bb}S_{abde} + A_{gc(f)}A_{hc(f)}A_{gg}A_{hf}B_{gh}B_{fg} + \\
&\quad Z_{jc(i)}A_{jj}B_{ij} + A_{u_1u_1}A_{u_1u_3}])^{-1} \\
&= [Z_{t_1c(r_1)}Z_{s_1c(r_1)}(A_{s_1s_1} + A_{t_1t_1})B_{s_1t_1}B_{r_1s_1} + \\
&\quad Z_{t_2c(r_2)}Z_{s_2c(r_2)}A_{s_2s_2}B_{s_2t_2}B_{r_2s_2} + \\
&\quad Z_{bc(a)}Z_{dc(a)}A_{bb}S_{abde} + A_{gc(f)}A_{hc(f)}A_{gg}A_{hf}B_{gh}B_{fg} + \\
&\quad Z_{jc(i)}A_{jj}B_{ij} + A_{u_1u_1}A_{u_1u_3}]^{-1}([Z_{vc(x)}S_{xvy} + Z_{zc(w)}B_{wz} + \\
&\quad A_{pq}]^*)^{-1}
\end{aligned}
$$

$$= [Z_{t_1c(r_1)}Z_{s_1c(r_1)}(A_{s_1s_1} + A_{t_1t_1})B_{s_1t_1}B_{r_1s_1} +$$
$$Z_{t_2c(r_2)}Z_{s_2c(r_2)}A_{s_2s_2}B_{s_2t_2}B_{r_2s_2} +$$
$$Z_{bc(a)}Z_{dc(a)}A_{bb}S_{abde} + A_{gc(f)}A_{hc(f)}A_{gg}A_{hf}B_{gh}B_{fg} +$$
$$Z_{jc(i)}A_{jj}B_{ij} + A_{u_1u_1}A_{u_1u_3}]^{-1}([Z_{vc(x)}S_{xvy} + Z_{zc(w)}B_{wz} +$$
$$A_{pq}]^{-1})^e$$

$$= [D_{s_1}D_{r_1s_1}D_{t_1}D_{s_1t_1}(D_{s_1s_1} + D_{t_1t_1})Z_{s_1\lambda}Z_{t_1\lambda} +$$
$$D_{s_2}D_{r_2s_2}D_{t_2}D_{s_2t_2}D_{s_2s_2}Z_{s_2\lambda}Z_{t_2\lambda} +$$
$$D_dD_bD_{de}D_{bd}D_{ab}A_{ae}D_{bb}Z_{d\lambda}Z_{b\lambda} +$$
$$D_gD_{fg}D_hD_{gh}D_{hf}D_{gg}Z_{h\lambda}Z_{g\lambda} + D_jD_{ij}D_{jj}Z_{j\lambda} +$$
$$D_{u_1u_3}D_{u_1u_1}][A_{xy}D_{vy}D_{xv}D_xZ_{v\lambda} + D_zD_{wz}Z_{z\lambda} + D_{pq}]^e$$

$\sigma_{pre}$ $= x,y \in V,\ v \notin V,\ xy,xx,yy \in E,\ c(x) = c(y)$

$w \in V,\ z \notin V,\ ww \in E$

distinct $p,q,r \in V,\ pq,rr \notin E,\ |\{pp,qq\} \cap E| \geq 1,$

$c(p) = c(q),\ \text{not}[pp,pr \notin E,\ qq,qr \notin E]$

$r_1 \in V,$ distinct $s_1,t_1 \notin V,\ r_1r_1 \in E$

$r_2 \in V,$ distinct $s_2,t_2 \notin V,\ r_2r_2 \notin E$

distinct $a,e \in V,$ distinct $b,d \notin V,\ ae,aa,ee \in E$

$f \in V,$ distinct $g,h \notin V,\ ff \in E$

$i \in V,\ j \notin V,\ |V| = 1,\ |E| = 0$

distinct $u_1,u_2,u_3 \in V,\ u_1u_2,u_2u_2,u_2u_3 \in E,\ |V| = 3,$

$|E| = 2$

$\sigma^{-1}$ $=$ distinct $t',u'x,v,y \in V,\ t'x,u'y,xx,yy,xv,vy \in E,$

$vv,t't',u'u',xy \notin E,\ d(v) = 2,$

$c(x) = c(y) = c(v) = c(t') = c(u')$

distinct $s',w,z \in V,\ s'w,ww,wz \in E,\ s's',zz \notin E,$

$d(z) = 1,\ c(w) = c(z) = c(s')$

distinct $a',b',p,q \in V,\ a'p,b'q,pq \in E,$

$a'a',b'b' \notin E,\ |\{pp,qq\} \cap E| \geq 1,$

$c(p) = c(q) = c(a') = c(b'),$

there exists a cycle containing $pq$

distinct $r_1,s_1,t_1 \in V,\ r_1r_1,r_1s_1,s_1s_1,s_1t_1 \in E,\ t_1t_1 \notin E,$

$d(s_{i>} = 2,\ d(V =: 1.\ dr_t) = c(s_1) = dt_t)$

distinct $r_i, s_1, t_1$ s V, $r_1 r\ r_i s_r s_-, t_r 1 t_1 t_1$ s E_ $s_i s_i$ E.

$d(s_{i»} = 2.\ d(t,)» 1.\ dr^ = c:(s_1) = 'dt,)$

distinct $r_{2'} s_7, t_{-z}$ <5 V, $t_2 t 2' r 2 r 2$ * E, $s_2 s_{2'} r_2 s_r s t_{23}$ E.

$d<s_2) = 2.\ d(t_2) = 1,\ c(r_2) = ds_2) = dX_2)$

distinct a,b,d.e • V, aa,bb,ee.ab,bd,de s E, dd.ae * E,

$d(b) = 2.\ d(d) = 2,\ da) = c(b) = c<d) = de>$

distinct f,g,h s v, ff,gg,fg,gh,fh s E, hh ss E. d(g) = 2,

$d<h) = 2,\ c(f) = dg) = dh)$

distinct i,j s V, ij.jj s E, ii * E. d(j) = 1, |V| = 2,

|E| = 1, di) = dj)

distinct $u_r u_2.u_3$ e V, $u_t r^u{}_2 2'^U i 2'^U 13'^U 23$ S E, $^U 33$ * E,

$|V| * 3.\ |E| = 3,\ du,) = du_2) = c(u_3)$

The floor shifts to <P_ X. JL >• Figure 4-32 shows VERTEX-COVER"[1] operating on a graph G « Gp and a graph G * Gp. 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 e Gp VERTEX-COVER"[1] will return G to $K_r$ 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^ and has interesting connotations, to be discussed in 4.7.

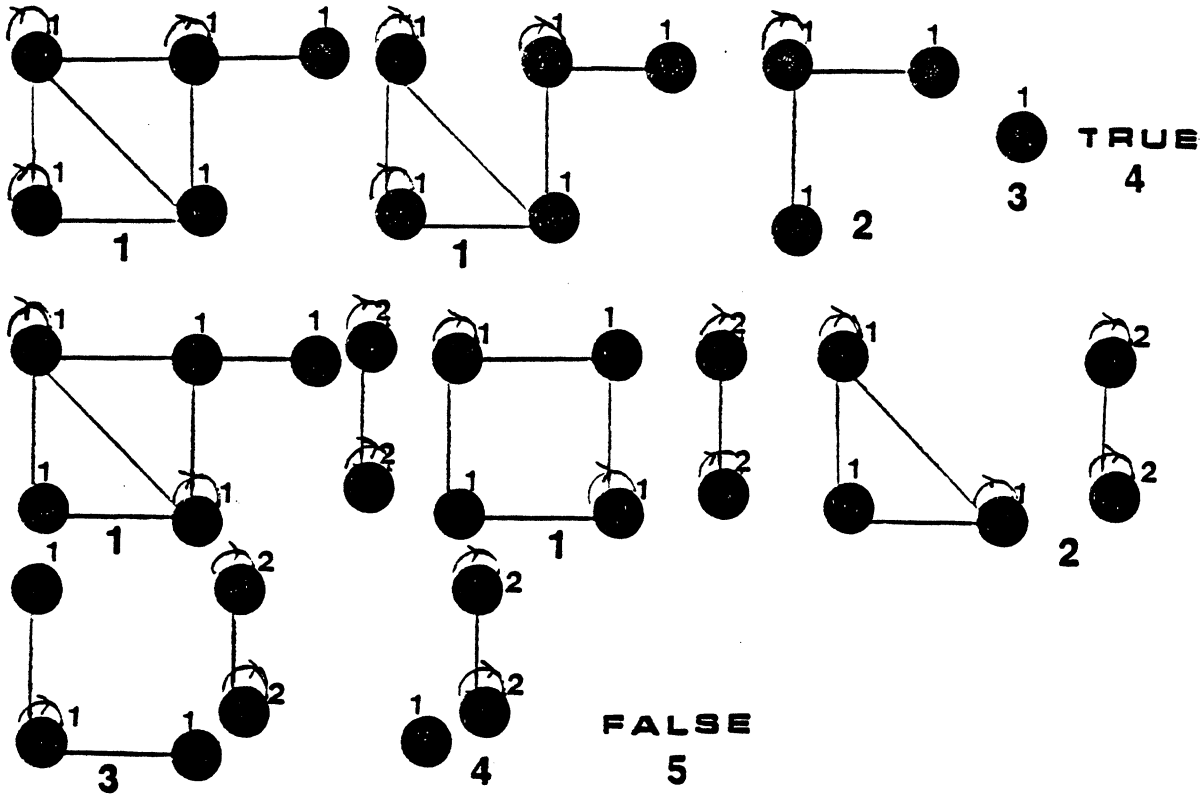



**Figure 4-32:** VERTEX-COVER$^{-1}$ in Operation

### 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.

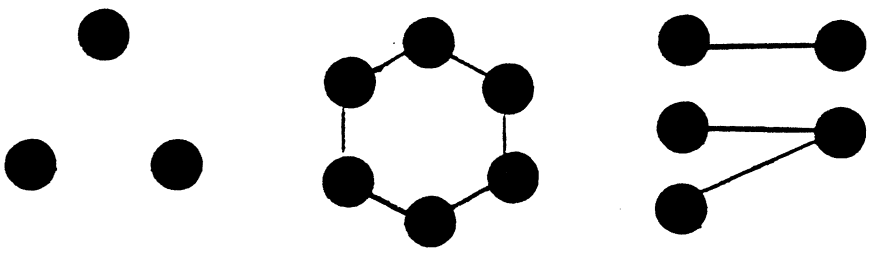


**Figure 4-33:** Some Graphs with Independence Number 3

The $R^c$-property INDEPENDENCE-K is:

$$(Z_{v_1\alpha_1}\ldots Z_{v_n\alpha_n} Z_{x\alpha} A^*_{xy_i} A_{xz_1}\ldots A_{xz_p} A_x)^*(LE_k)$$

where $x \notin V$, $y_i, z_i \in V$, $z_i$ is in the ith independent set,

p = number of label-indicated sets, $\alpha,\alpha_i$ are correctly constructed labels

Figure 4-34 shows the iterative steps in a sample run of INDEPENDENCE-K for k = 3.

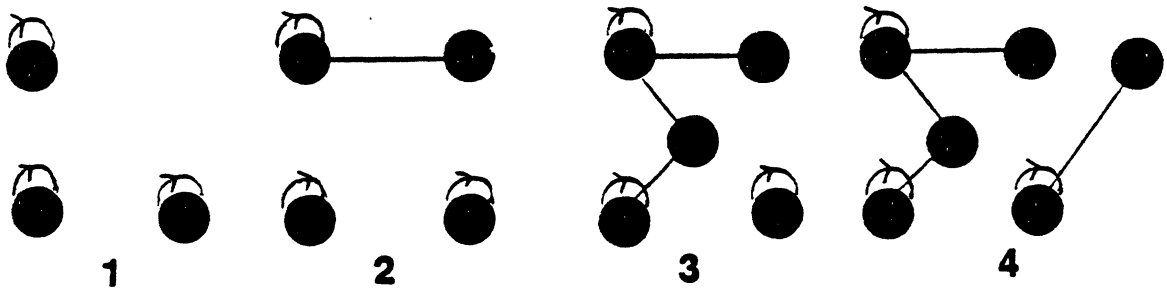



**Figure 4-34:** INDEPENDENCE-3 in Operation

The floor for graphs with independence number k is $\langle P_{1c}, L_{1nc}, \Sigma_{6c} \rangle$. 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:

$$f^{-1} = (Z_{v_1\alpha_1} \ldots Z_{v_n\alpha_n} Z_{x\alpha} A^*_{xy_i} A_{xz_1} \ldots A_{xz_p} A_x)^{-1}$$

$$= A_x^{-1}A_{xz_p}^{-1}\ldots A_{xz_1}^{-1}(A_{xy_i}A_{xy_i}^{*})^{-1}Z_{v\alpha}^{-1}Z_{v_n\alpha_n}^{-1}\ldots Z_{v_1\alpha_1}^{-1}$$

$$= D_x D_{xz_p}\ldots D_{xz_1} A_{xy_i}^{e} Z_v\lambda Z_{v_n}\lambda\ldots Z_{v_1}\lambda$$

$\sigma_{pre}$ = $x \notin V$, $y_i,z_i \in V$, $z_i$ is in the ith independent set, p = number of label-indicated sets, $\alpha,\alpha_i$ are correctly constructed labels

$\sigma^{-1}$ = $x,y_i,z_i \in V$, $xx \notin E$, $z_i$ is in the ith independent set, p = number of label-indicated sets, $\alpha,\alpha_i$ correctly constructed labels, c(x) indicates $y_i$ is not independent of x

The floor shifts to $<P_{2c},L_{1nc},\Sigma_{6c}>$. 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

$$f^{-1} = D_x D_{xz_p}\ldots D_{xz_1} A_{xy_i}^{e} Z_{v_n\beta_n}\ldots Z_{v_1\beta_1}$$

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 $G \notin G_p$ 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 $G \notin G_p$, there must be k loops, or G will fail. Given $G \notin G_p$ 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 $Z_{xy\alpha}$ to color edge xy with $\alpha$), 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 $\Sigma_{ie}$, corresponding to $P_{ic}$, $L_{ic}$, $L_{inc}$, $L_{iuc}$, $L_{inuc}$ and $\Sigma_{ic}$, respectively. The only difference between vertex

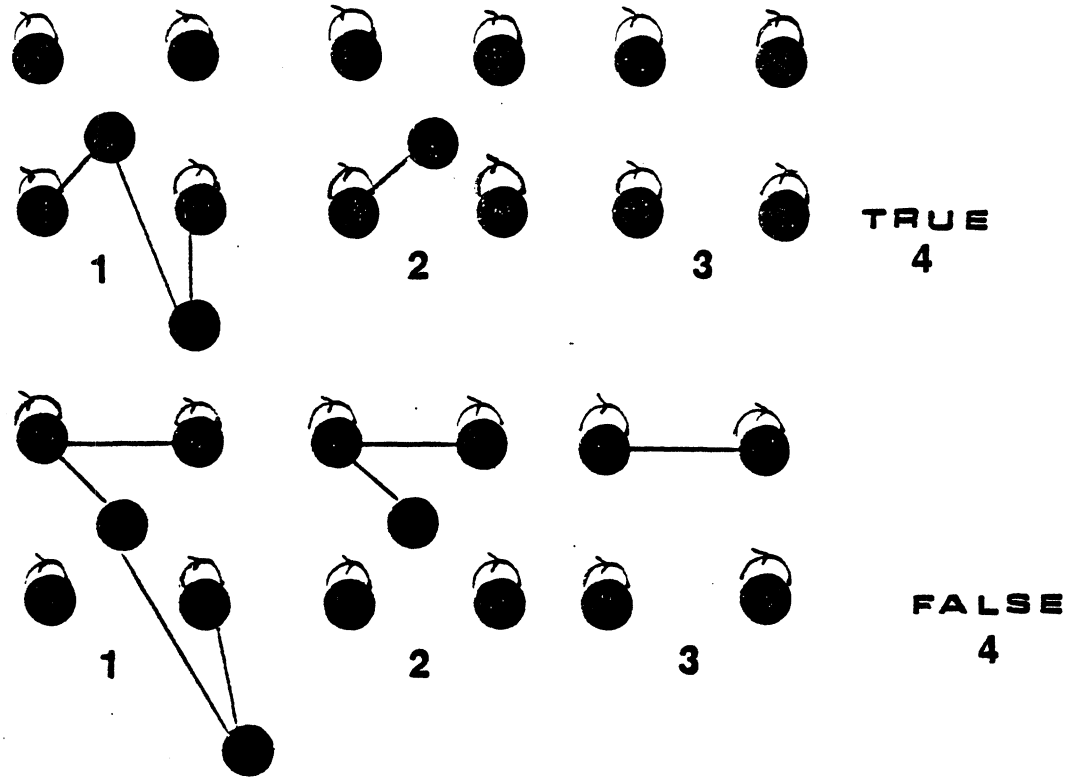



**Figure 4-35:** INDEPENDENCE-$4^{-1}$ in Operation

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 $R^e$-property CIRCUMFERENCE-K:

$$(Z_{xy\beta}A_{xy} + A_z + Z_{pq\alpha}B_{pq} + Z_{v_1v_2\alpha}\ldots Z_{v_{r-1}v_r\alpha}Z_{v_rv_1\alpha}Y_{v_1\ldots v_r})^*(C_k^1)$$

where $x,y,x',y' \in V$, $xx',yy' \in E$, $c(xx') = c(yy') = \beta$

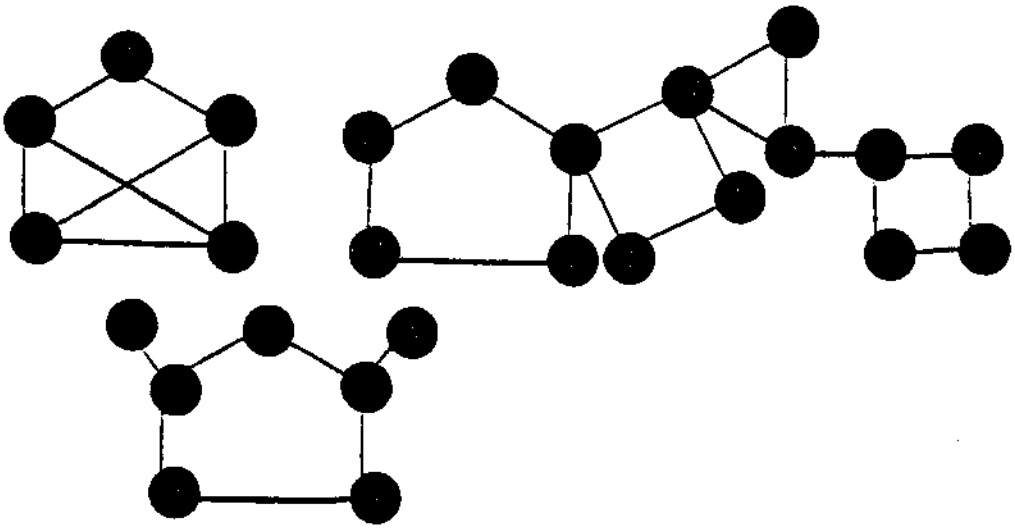


**Figure 4-36:** Some Graphs with Circumference 5

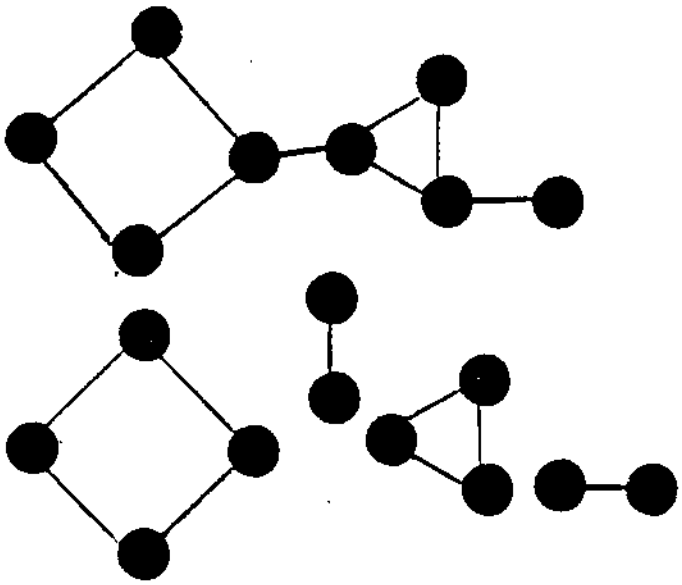


**Figure 4*37:** A Graph and its Blocks

p s V, q ^ V, a is a new color

$v_1$ s V, $v_2$,...,$v_r$ * V, <x is a new color, r £ k

The expression "a is a new color[1]' is an abbreviation for "ctt^ # a, $c(t_2)$ # a,...,$c(t_n)$ * <*>, $\{t_r t_2 ^_r t_n\}$ = V." The seed C^ 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 <$P_1$^LQ^ZJ^-

We will prove the correctness of CIRCUMFERENCE-K as we explain its

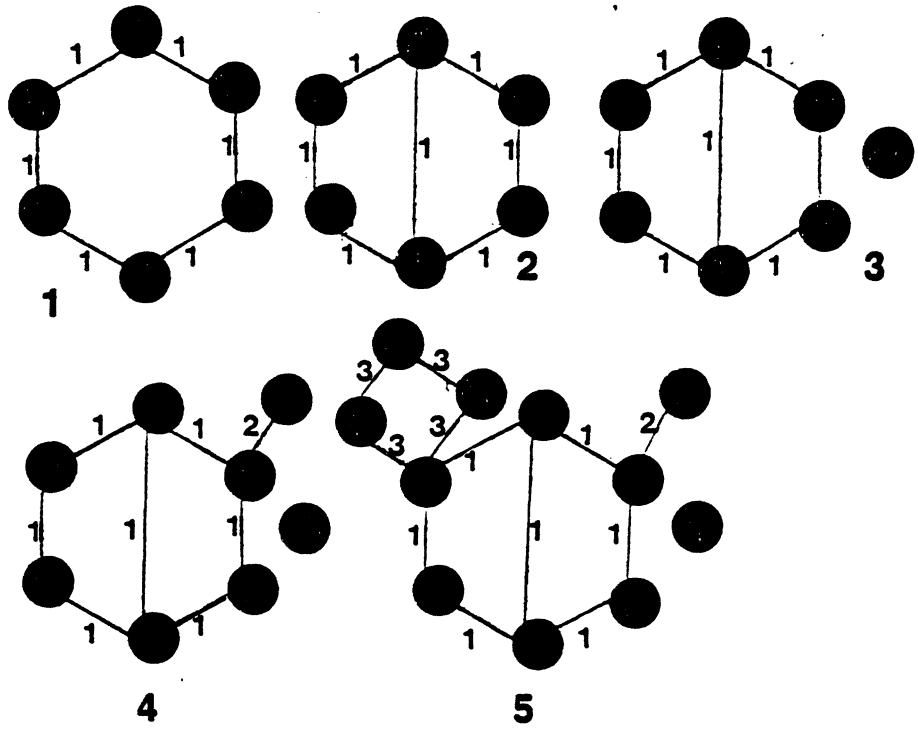



**Figure 4-38:** CIRCUMFERENCE-6 in Operation

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:

$$f^{-1} = (Z_{xy\beta}A_{xy} + A_z + Z_{pq\alpha}B_{pq} + Z_{v_1v_2\alpha}\cdots Z_{v_{r-1}v_r\alpha}Z_{v_rv_1\alpha}Y_{v_1\cdots v_r})^{-1}$$

$$= (Z_{xy\beta}A_{xy})^{-1} + A_z^{-1} + (Z_{pq\alpha}B_{pq})^{-1} +$$

$$(Z_{v_1v_2\alpha}\cdots Z_{v_{r-1}v_r\alpha} Z_{v_rv_2\alpha} Y_{v_1\ldots v_r})^{-1}$$

$$= D_{xy}Z_{xy\lambda} + D_z + D_qD_{pq}Z_{pq\lambda} + \underline{Y}_{v_1\ldots v_r} Z_{v_rv_1\lambda} Z_{v_{r-1}v_r\lambda}\cdots Z_{v_1v_2\lambda}$$

$\sigma_{pre}$ = x,y,x',y' ∈ V, xx',yy' ∈ E, xy ∉ E, c(xx') = c(yy') = β

z ∉ V

p ∈ V, q ∉ V, α is a new color

$v_1$ ∈ V, $v_2,\ldots,v_r$ ∉ V, α is a new color, r ≤ k

$\sigma^{-1}$ = x,y,x',y' ∈ V, xy,xx',yy' ∈ E, c(xx') = c(yy') = c(xy), x' ≠ y, y' ≠ x, there exist two edge-disjoint paths from x to y without edge xy,

z ∈ V, d(z) = 0

p,q ∈ V, pq ∈ E, d(q) = 1, c(pq) = α, α is a unique color

$v_i$ ∈ V, $v_1v_2,\ldots,v_{r-1}v_r,v_rv_1$ ∈ E,

$|\{rs|r,s \in V, rs \in E, c(rs) = c(v_1v_2)\}| = r$,

there exists another cycle of at least size r

The floor shifts to $\langle P_{2e}, L_{\Omega e}, \Sigma_{6e}\rangle$. Figure 4-39 shows CIRCUMFERENCE-$K^{-1}$ operating on a graph G ∈ $G_p$ and a graph G ∉ $G_p$ for k = 3.

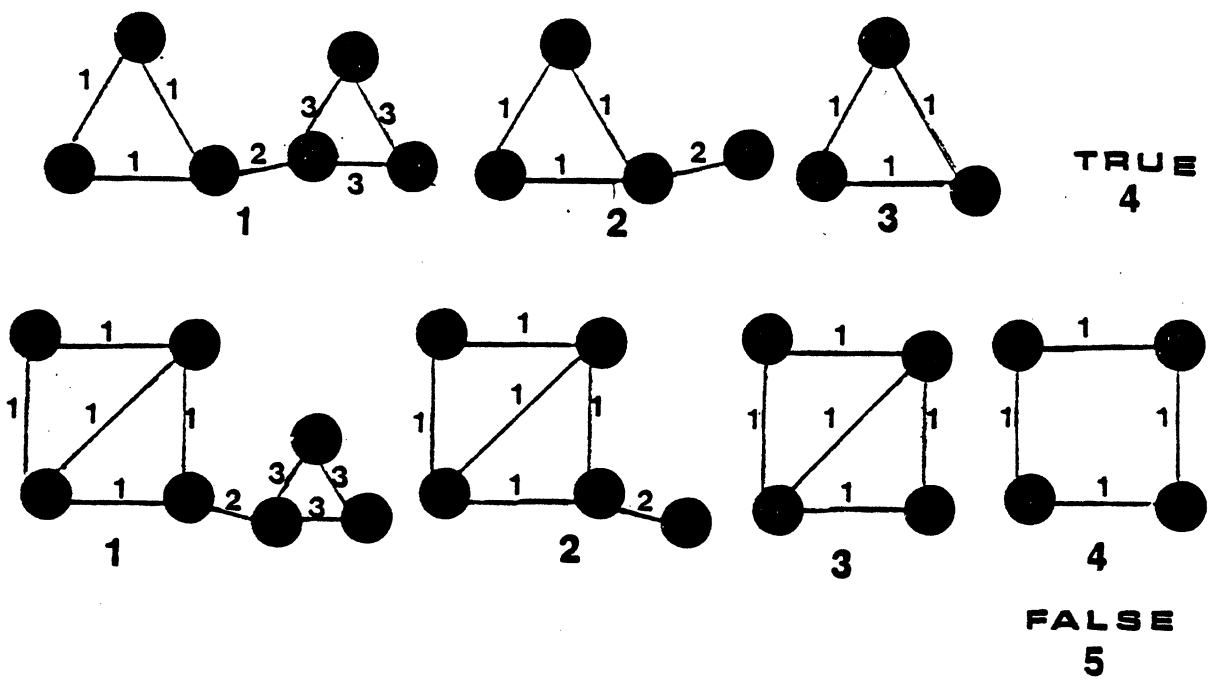



**Figure 4-39:** CIRCUMFERENCE-$3^{-1}$ in Operation

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 $C^*_k$ where the color <x is not relevant to the isomorphism testing. A graph G * $G_p$ with overly-small cycles will be reduced to <+4> and fail, while a graph with an overly-large cycle will retain it and fail. Thus C1RCUMFERENCEH<~$^1$ is correct and CIRCUMFERENCE-K is complete .

### 4.4.7. Graphs with Edge Covering Number K

For a graph G = <V,E> an edge subset E S 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.

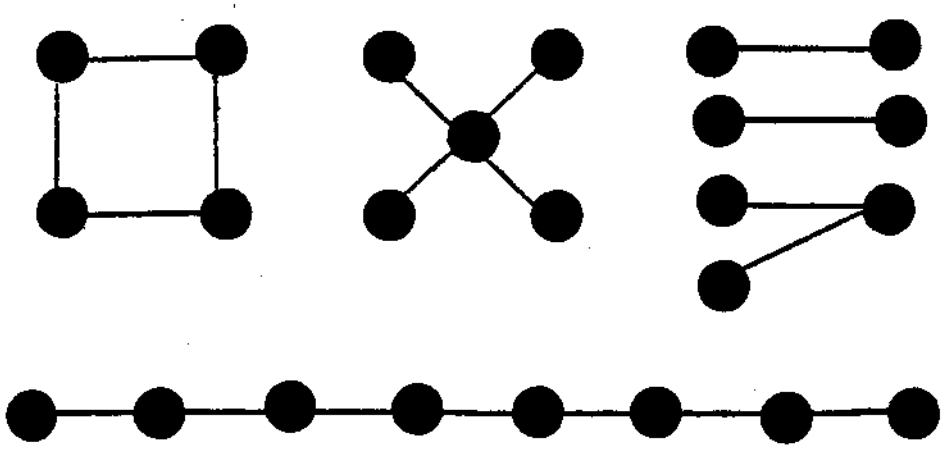


**Figure 4*40:** Some Graphs with Edge Covering Number 4

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 pth 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 $S_3$.

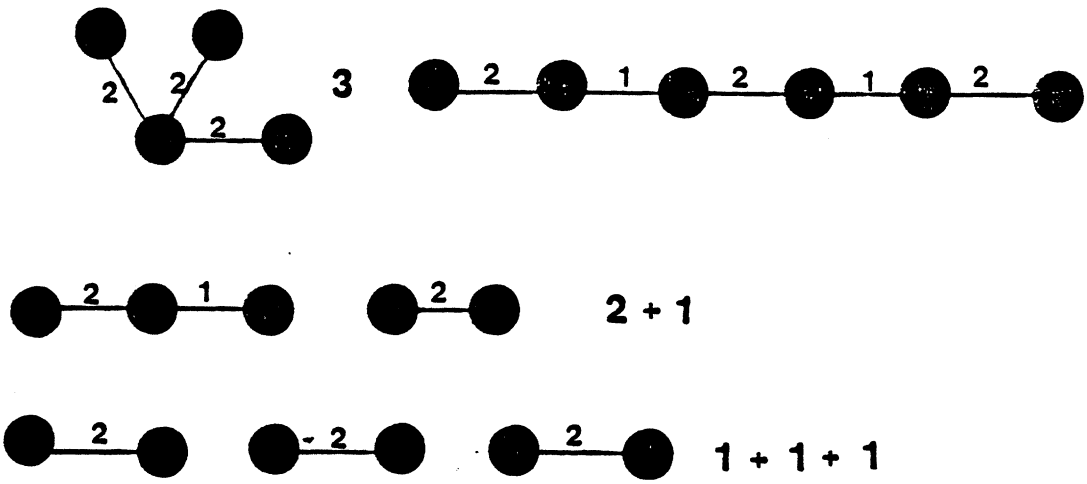



**Figure 4-41:** The Seed Graphs for Edge Cover 4

Now we can state the $R^e$-property K-EDGE-COVER as:

$(Z_{xyc(xt)-1}A_{xy}F_{xyu} + Z_{vwc(vt)-1}A_{vw})^*(S_k)$ where

distinct x,t,u ∈ V, y ∉ V, xt,xu ∈ E, c(xt) = c(xu) is even

v,w,t,u ∈ V, vt,wu ∈ E, c(vt) = c(wu) even,

not [r ∈ V, rv, rw, ∈ 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 $\langle P_{4e}, L_{\Omega e}, \Sigma_{5e} \rangle$. 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

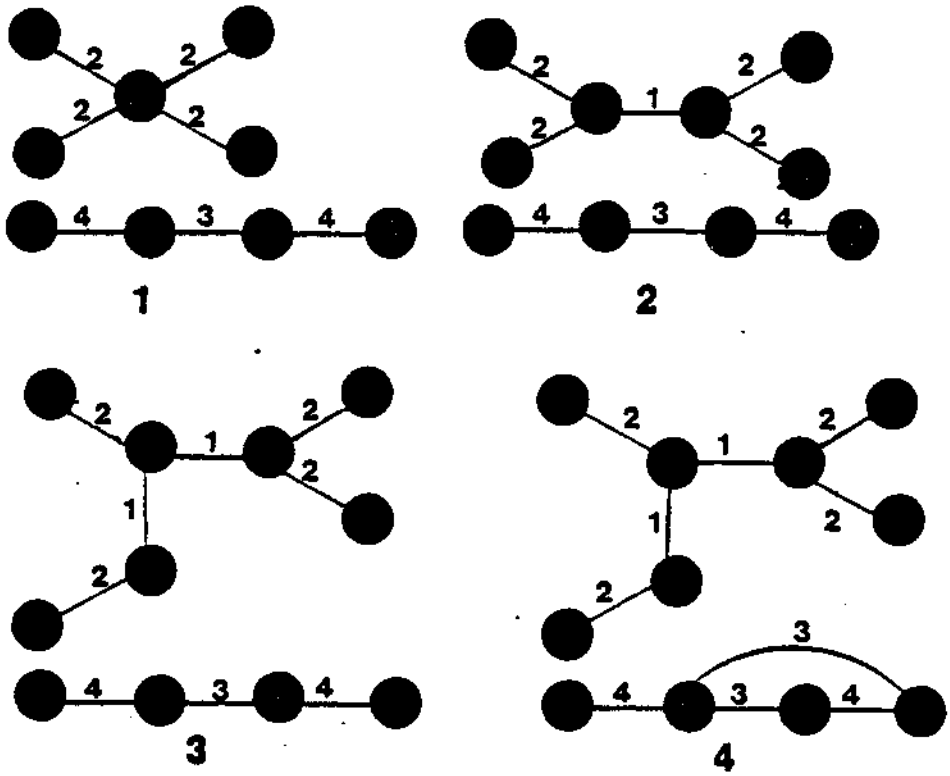



**Figure 4-42: 6-EDGE-COVER in Operation**

ultimately expands a star on k+1 vertices into a tree on 2k+1 vertices, with alternatingiy 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:

$f^{-1}$ $= (Z_{xyc(xt)\text{-}1} A_{xy} F_{xyu} + Z_{vwc(vt)-1} A_{vw})^{-1}$

$= (Z_{xyc(xt)\text{-}1} A_{xy} F_{xyu})^{-1} + (Z_{vwc(vt)-1} A_{vw})^{-1}$

$= I_{xy} D_{xy} Z_{xy} A + D_{vw} Z_{vw} A$

$\sigma_{pre}$ $^{s}$ distinct x,t,u • V, y *0* V, xt,xu e E, c(xt) = c(xu) is even

v,w,tu 6 V, vt,wu s £ vw 0 E, c(vt) = c(wu) even,

not [r e V, rv,rw e E, c(rv) = c(rw) even]

$\sigma^{-1}$ « distinct x,y,t,u e V, xy,xt,yu e E, c<xy) odd,

c(xt) = c(yu) even, c(xt) = c(xy) + 1, c(xt) = $_{C}$(yu),

$$|\{rs \mid rs \in E,\ c(sz) = c(xy)\}| < |\{rs \mid rs \in E,$$
$$c(rs) = c(xy) + 1\}|,\ |V| = |E| + 1$$
$$v,w \in V,\ vw \in E,\ c(vw) \text{ odd},$$
$$|\{rs \mid rs \in E,\ c(rs) = c(vw)\}| \geq |\{rs \mid rs \in E,$$
$$c(rs) = c(vw) + 1\}|,$$
$$\text{not}[r \in V,\ rv,rw \in E,\ c(rv) = c(rw) \text{ even}]$$

The floor remains constant. Figure 4-43 shows K-EDGE-COVER$^{-1}$ operating on a graph $G \in G_p$ and a graph $G \notin G_p$ for k = 3.

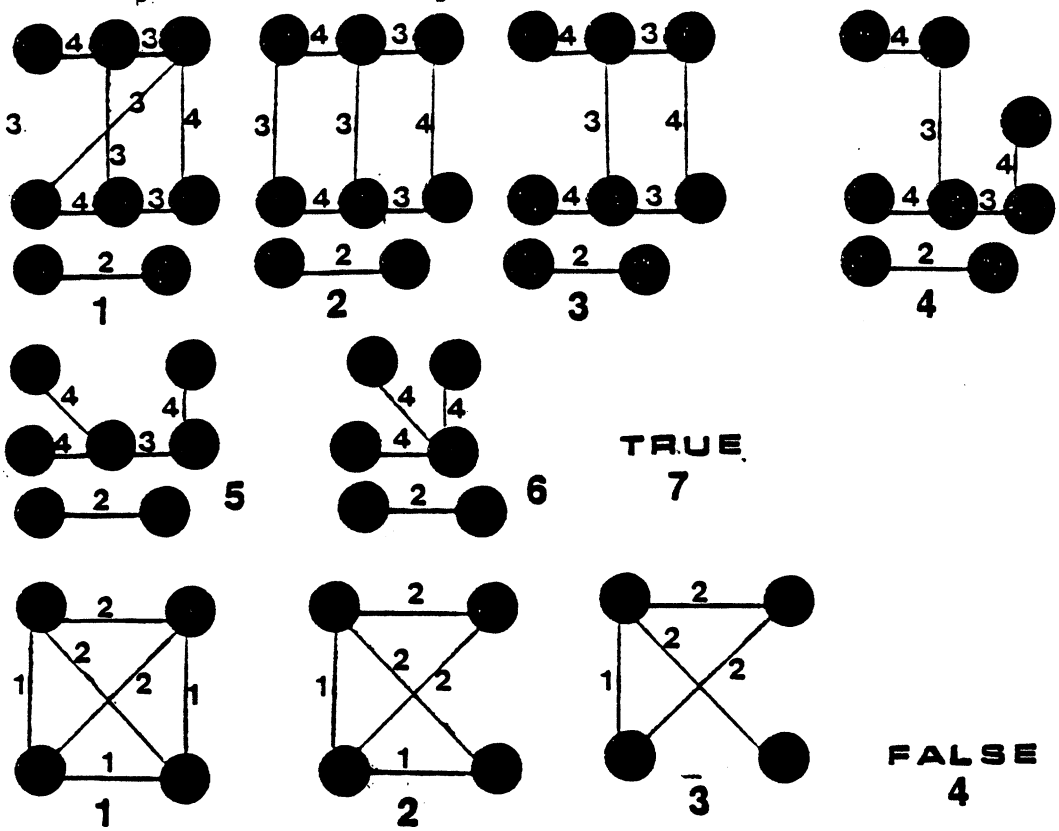



**Figure 4-43:** 3-EDGE-COVER$^{-1}$ in Operation

The generation process for K-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 $G \notin G_p$, 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-EDGE-COVER$^{-1}$ is correct and K-EDGE-COVER is complete.

### 4.4.8. Graphs with a k-Factor

A *k-factor* of a graph G = <V,E> 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.

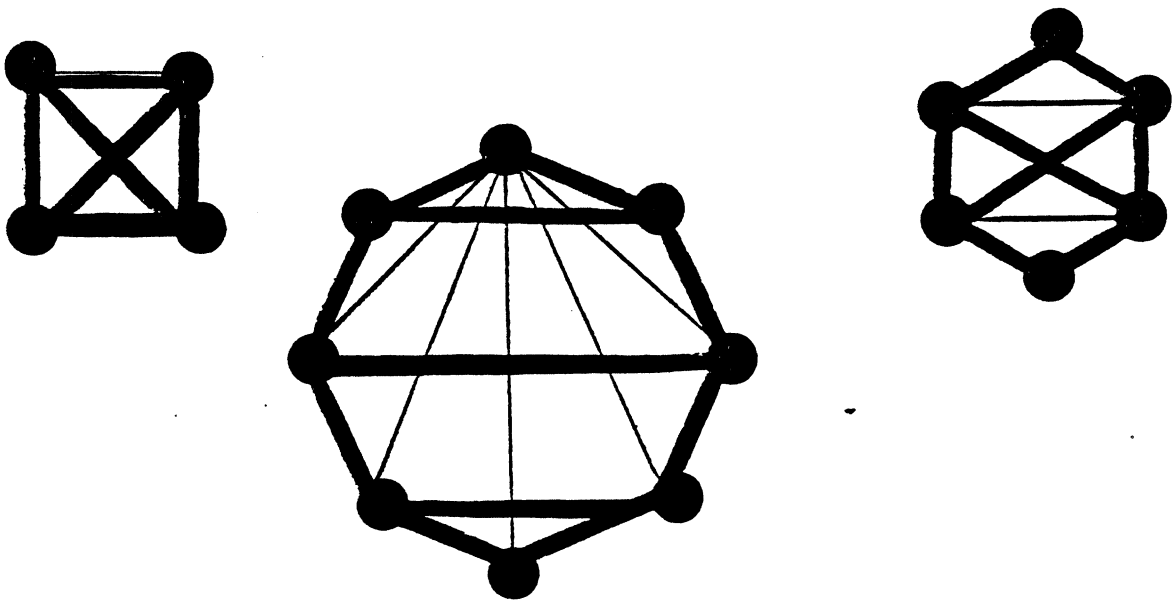


**Figure 4-44:** Some Graphs with 3-Factors

The factor edges appear darkened in the figure. The $R^e$-property K-FACTOR has separate options for k even and k odd, using the composite operators $EM_{xv_1\cdots v_k}$, $OM_{vwv_1\cdots v_{k+1}w_1\cdots w_{k-1}}$, $CM_{v_1\cdots v_k}$ and $FR_{v_1\cdots v_k}$ defined in 3.7.20 for appending even and odd degree vertices without changing the degree of any previously-existing vertex. The $R^e$-property K-FACTOR is

$$(Z_{xy\alpha}A_{xy} + EM_{ss_1\cdots s_k} + OM_{vwv_1\cdots v_{k+1}w_1\cdots w_{k-1}} + CM_{u_1\cdots u_k} + A_{y_2q}A_{y_1p}D_{y_1y_2}CM_{y_1\cdots y_{k+1}}D_{pq} + FR_{z_1\cdots z_k})^*(K_{k+1})$$

where $x,y \in V$, $xy \notin E$

distinct $s_r s_{2\_}, s_k$ s V . s ^ V , $^{s}2j\text{-}i^{s}2j$ $^{6\ E,\ c<s}2j\text{-}1^{S}2j\}$ # *'

j = 1,2,...,k/2; k even

distinct v^v.^v, . . s V, distinct w .,w _,...,w. . s V, distinct v,w £ V,

$^{v}2i\text{-}1^{v}2i'^{w}2j\text{-}1^{w}2i$ $^{e\ E.\ c(v}2i-1^{v}2i)$ ≠ α, $^{c(w}2_{j}\text{-}1^{w}2j)$ # *'

i = 1,2,...,(k+1)/2; j = 1,2....(k-1)/2; k odd

distinct $u_1, U_{2'}, -, u_k$ & V

distinct p,q e V, distinct $y_r y_2$'-'$V_{k+1}$ * V, pq e E. c(pq) # a

distinct x^x^x^Zj s V, distinct $z_2, z_3, \ldots, z_k$ * V, ZjX.$_{,}$ e E, i = 1,2,~,k;

$c(z_1 x_i)$ # a

Figure 4-45 shows the iterative steps in a sample run of K-FACTOR for k = 4.

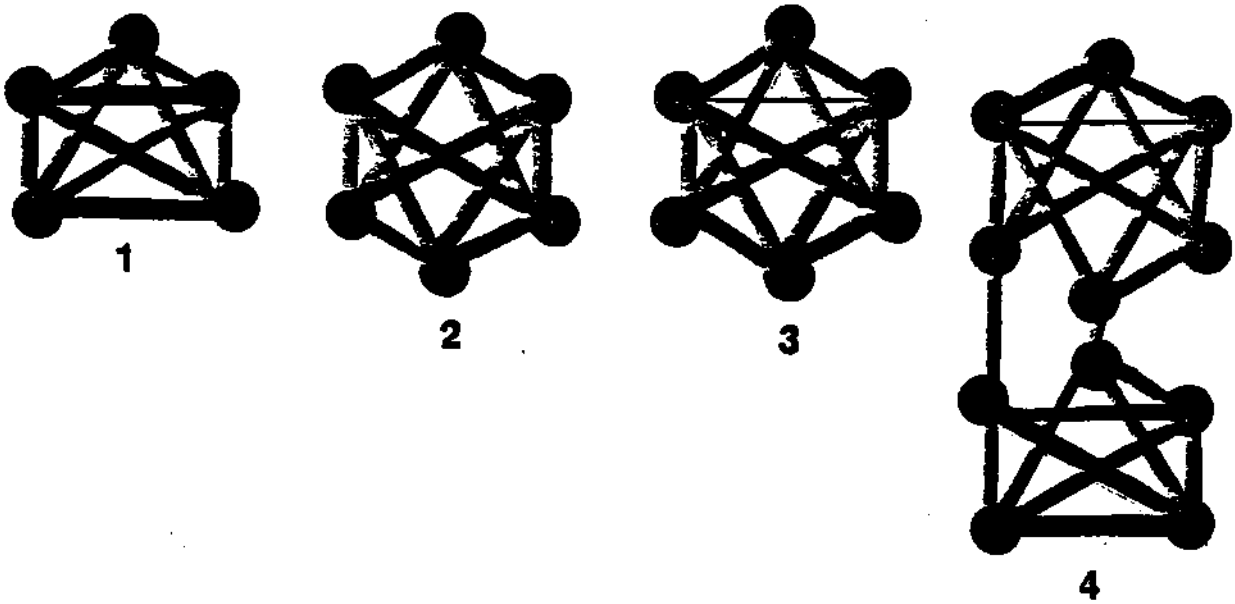



**Figure 4-45: 4-FACTOR in Operation**

Throughout its execution, K-FACTOR distinguishes the edges in the factor (unlabelled) from the edges not in the factor (labelled a.) The floor for graphs with a k-factor is <$P_{2e}$ J——— 5l $_{2e}$ >.

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 uniabelled 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 $(u_1,\ldots,u_k)$ 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:

$$f^{-1} = (Z_{xy\alpha}A_{xy} + EM_{ss_1\ldots s_k} + OM_{vwv_1\ldots v_{k+1}w_1\ldots w_{k-1}} + CM_{u_1\ldots u_k} + A_{y_2q}A_{y_1p}D_{y_1y_2}CM_{y_1\ldots y_{k+1}}D_{pq} + FR_{z_1\ldots z_k})^{-1}$$

$$= (Z_{xy\alpha}A_{xy})^{-1} + EM^{-1}_{ss_1\ldots s_k} + OM^{-1}_{vwv_1\ldots v_{k+1}w_1\ldots w_{k-1}} + CM^{-1}_{u_1\ldots u_k} + A_{y_2q}A_{y_1p}D_{y_1y_2}CM_{y_1\ldots y_{k+1}}D_{pq} + FR^{-1}_{z_1\ldots z_k}$$

$$= D_{xy}Z_{xy\lambda} + EM^{-1}_{ss_1\ldots s_k} + OM^{-1}_{vwv_1\ldots v_{k+1}w_1\ldots w_{k-1}} + CM^{-1}_{u_1\ldots u_k} + A_{pq}CM^{-1}_{y_1\ldots y_{k+1}}A_{y_1y_2}D_{y_1p}D_{y_1q} + FR^{-1}_{z_1\ldots z_k}$$

$\sigma_{pre}$ = $x,y \in V$, $xy \notin E$

distinct $s_1,s_2,\ldots,s_k \in V$, $s \notin V$, $s_{2j-1}s_{2j} \in E$,

$c(s_{2j-1}s_{2j}) \neq \alpha$, $j = 1,2,\ldots,(k+1)/2$; $j = 1,2,\ldots,(k-1)/2$;

k odd, $d(s_i) = k$, $i = 1,2,\ldots,k$

distinct $v_1,v_2,\ldots,v_{k+1} \in V$, distinct $w_1,w_2,\ldots,w_{k-1} \in V$,

distinct $v,w \notin V$, $v_{2i-1}v_{2i}w_{2j-1}w_{2j} \in E$,

$c(v_{2i-1}v_{2i}) \neq \alpha$, $c(w_{2j-1}w_{2j}) \neq \alpha$,

$i = 1,2,\ldots,(k+1)/2$; $j = 1,2,\ldots,(k-1)/2$; k odd; $d(v_i) = k$,

$i = 1,2,\ldots k+1$; $d(w_i) = k$, $i = 1,2,\ldots,k-1$

distinct $u_1,u_2,\ldots,u_k \notin V$

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

$d(p) = k$, $d(q) = k$

distinct $x_1,x_2,\ldots,x_k,z_1 \in V$, distinct $z_2,z_3,\ldots,z_k \notin V$,

$z_1x_i \in E$, $i = 1,2,\ldots,k$, $d(z_1) = k$

$\sigma^{-1}$ = $x,y \in V$, $xy \in E$, $c(xy) = \alpha$

distinct $s,s_1,s_2,\ldots,s_k \in V$, $s_{2j-1}s_{2j} \in E$, $c(s_{2j-1}s_{2j}) \neq \alpha$,
$j = 1,2,\ldots,k/2$; k even, $d(s) = k$

distinct $v,v_1,v_2,\ldots,v_{k+1} \in V$, distinct $w,w_1,w_2,\ldots,w_{k-1} \in V$,
$vw,vv_i,ww_j \in E$, $v_{2i-1}v_{2i},w_{2j-1}w_{2j} \notin E$,
$c(v_{2i-1}v_{2i}) \neq \alpha$, $c(w_{2j-1}w_{2j}) \neq \alpha$, k odd, $d(v) = k$,
$d(w) = k$, $d(v_i) = k$, $d(w_j) = k$, $i = 1,2,\ldots,(k+1)/2$;
$j = 1,2,\ldots,(k-1)/2$;

distinct $u_1,u_2,\ldots,u_k \in V$,
$|\{pu_i \mid p \in V\}| = |\{u_iu_j \mid u_iu_j \in E\}| = k(k-1)/2$

distinct $p,q,y_1,y_2,\ldots,y_{k+1} \in V$, $pq \notin V$, $y_iy_j \in E$, $d(y_i) = k$,
$i = 1,2,\ldots,k+1$, $j = 1,2,\ldots,k+1$ except $y_1y_2$;
$y_1p$, $y_2q \in E$

distinct $x_i,z_i \in V$, $x_iz_i \in E$, $d(x_i) = k$, $d(z_i) = k$, $i = 1,2,\ldots,k$;
$x_1z_j \notin E$, $j = 2,3,\ldots,k$; $z_{rs} \in E$, $r,s = 1,2,\ldots,k$

The floor shifts to $\langle P_{2e},L_{1ne},\Sigma_{3e}\rangle$. Figure 4-46 shows K-FACTOR$^{-1}$ operating on a graph $G \in G_p$ and a graph $G \notin G_p$ for k = 2. In the figure the unlabelled factor edges are darkened. On a graph with a correctly (un)labelled k-factor, K-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 $G \notin G_p$, K-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-FACTOR$^{-1}$ is correct and K-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.

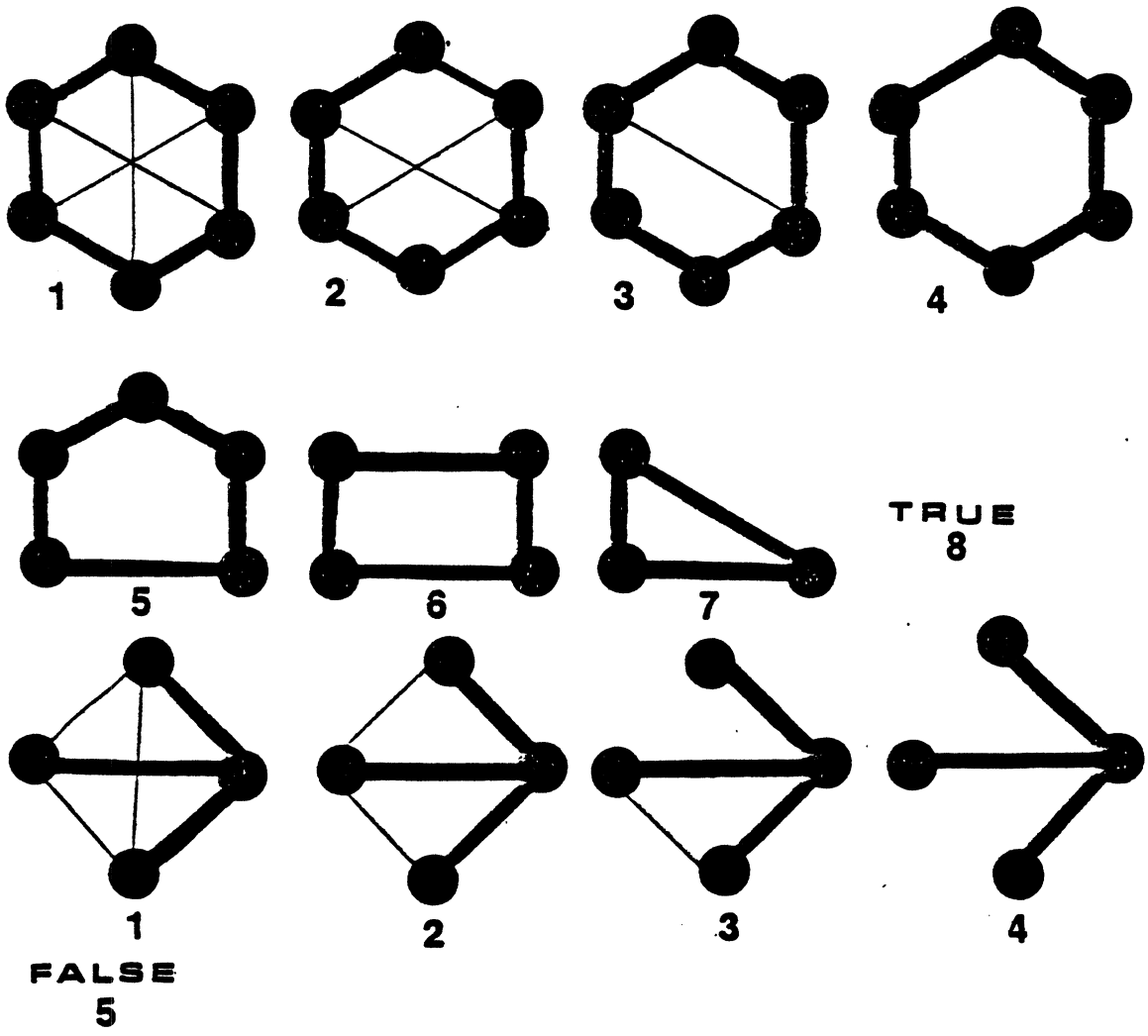



**Figure 4-46:** 2-FACTOR$^{-1}$ in Operation

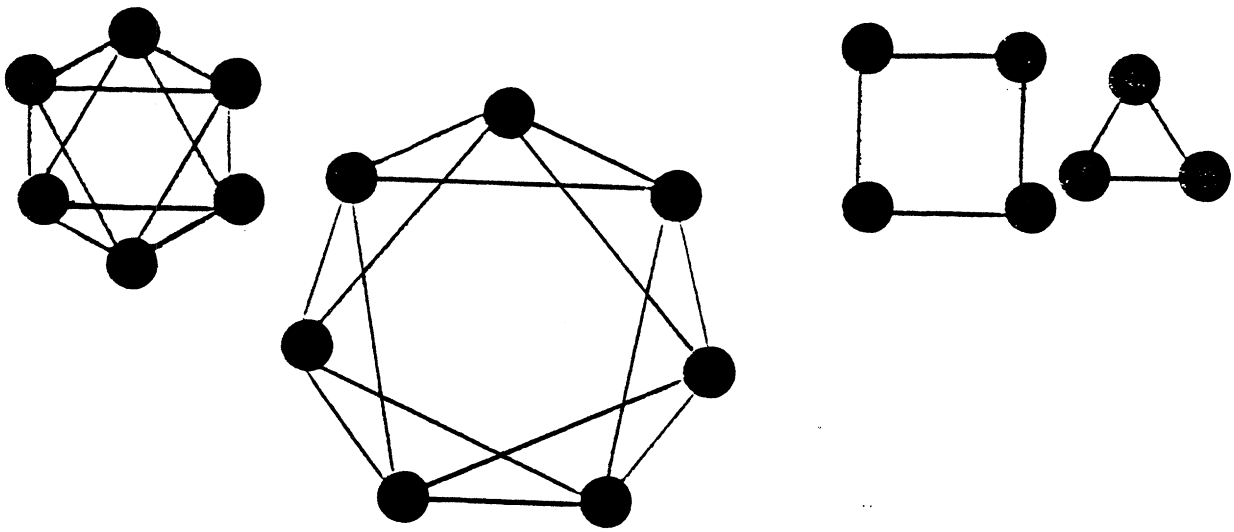


**Figure 4-47:** Some 2-Factorable Graphs

$(Z_{x_1x_{2^{p+1}}}\ldots Z_{x_{nk/2-1}x_{nk/2^{p+1}}} A_{x_1x_2}\ldots A_{x_{nk/2-1}x_{nk/2}} (\Pi_{i=1}^{p} f^i)^*(\Pi_{i=1}^{p} g^i)^*(Q_{k+1}, 1)$

where $x_i \in V$, $|\{x_i\}| = n$, every $x \in V$ appears exactly k times in $\{x_i\}$,

$x_{2j-1}x_{2j} \notin E$, $j = 1,2,\ldots,nk/4$

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 ith 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 $\Pi_{i=1}^{p} f^i$ by each application of f. The third operator, $g^i$, denotes an application of ODD-REGULAR in which the ith 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 $\Pi_{i=1}^{p} g^i$ 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:

$$f^{-1} = (Z_{x_1x_{2^{p+1}}}\ldots Z_{x_{nk/2-1}x_{nk/2^{p+1}}} A_{x_1x_2}\ldots A_{x_{nk/2-1}x_{nk/2}} (\Pi_{i=1}^{p} f^i)^*(\Pi_{i=1}^{p} g^i)^*)^{-1}.$$

$$= (\Pi_{i=p}^{1}(g^i)^{-1})^e(\Pi_{i=p}^{1}(f^i)^{-1})^e(Z_{x_1x_{2^{p+1}}}\ldots Z_{x_{nk/2-1}x_{nk/2^{p+1}}} A_{x_1x_2}\ldots A_{x_{nk/2-1}x_{nk/2}})^{-1}$$

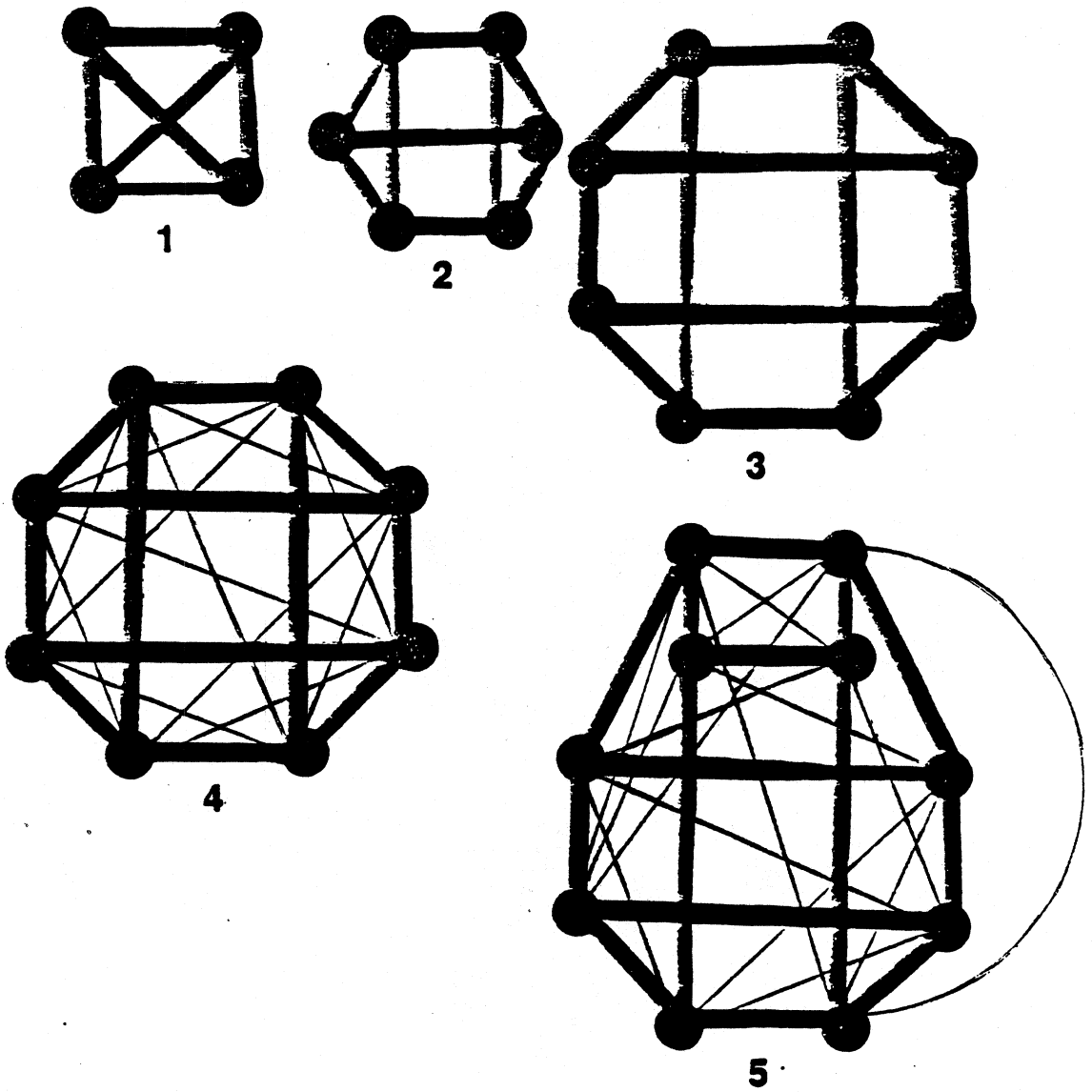



**Figure 4-48:** 3-FACTORABLE in Operation

$= (\Pi_{i=p}^{1}(g^{-1})^{i_i})^{e}(\Pi_{i=p}^{1}(f^{-1})^{i_i})^{e} D_{x_{nk/2-1}x_{nk/2}} \ldots D_{x_1x_2}$

$Z_{x_{nk/2-1}x_{nk/2}\lambda} \ldots Z_{x_1x_2\lambda}$

$\sigma_{pre} = \sigma$ $= x_i \in V$, $|\{x_i\}| = n$, every $x \in V$ appears exactly k times in $\{x_i\}$, $x_{2j-1}x_{2j} \notin E$, $j = 1,2,\ldots,nk/4$

f is applied according to EVEN-REGULAR, k even, same vertices restriction

g is applied according to ODD-REGULAR, k odd, same vertices restriction

$\sigma^{-1}$ $= x_i \in V$, $|\{x_i\}| = n$, every $x \in V$

appears exactly k times in $\{x_i\}$, $*_{2i-1}^{X}2_i \in E$,

$c(x_1, \ldots x_n) = p$, $i = 1,2,\ldots,nk/4$

$f^{-1}$ 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 <p2e'Line/^5e>' Rgure 4~49 show$ K-FACTORABLE$^{-1}$ operating on a graph G s $G_p$ and a graph G 2 $G_p$ for k = 2.

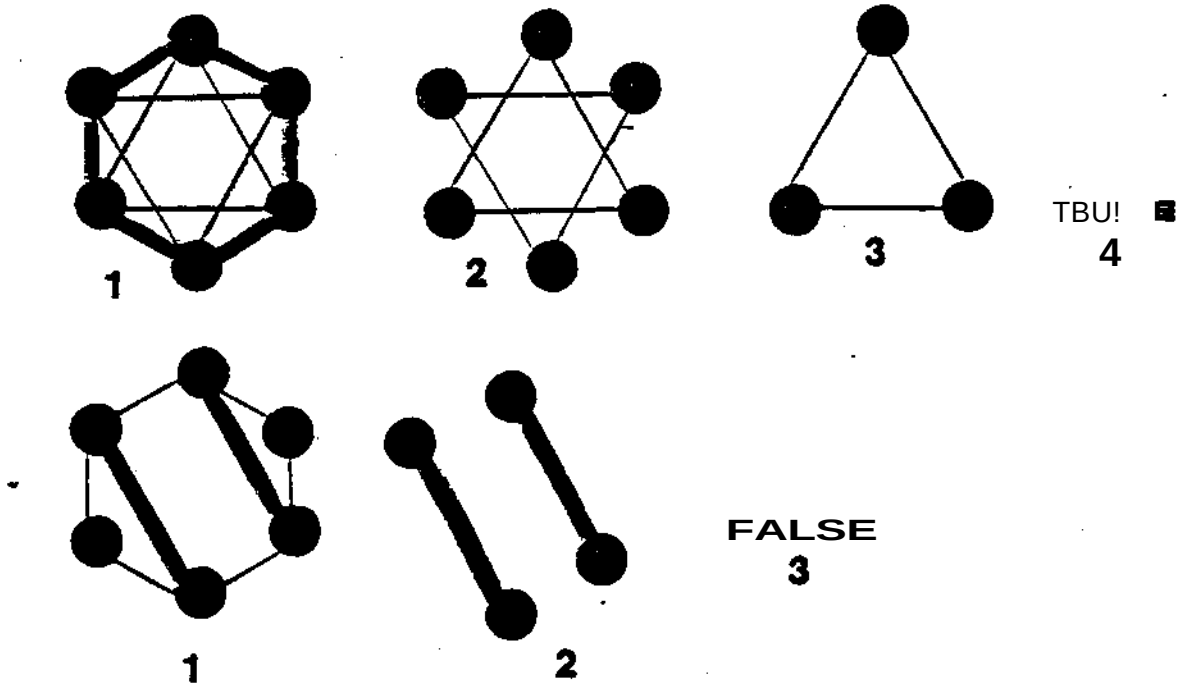



**Figure 4-49:** 2-FACTORABLE$^{-1}$ in Operation

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 * G and will fail. K-FACTORABLE$^{-1}$ is correct and P 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 $R^e$-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 $S \in S_2$
- $\sigma_2$ is subsumed by $\sigma_1$

Thus we need only specify how operators and selectors subsume each other. We first define selector subsumption. Let $\sigma_1$ and $\sigma_2$ be selectors which select vertex sets $V_1$ and $V_2$ with respect to a graph G. We say that selector $\sigma_1$ subsumes selector $\sigma_2$ if and only if there exists a mapping $\phi: V_2 \rightarrow V_1$ such that

- $\phi$ is a function (i.e., $\phi(v)$ is unique for each $v \in V_2$)
- $\phi$ is one-to-one (i.e., $\phi(v) = \phi(v')$ if and only if $v = v'$)
- $\phi$ preserves the following relationships:
  - $\in V$
  - $\notin V$
  - vertex color
  - edge color
- a vertex v selected by $\sigma_2$ will always be accepted as $\phi(v)$ by $\sigma_1$

We can be certain that $\sigma_1$ will accept v as $\phi(v)$ if and only if the description of $\phi(v)$ in $\sigma_1$ is consistent with and no more restrictive than the description of v in $\sigma_2$. 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 e V" is less restrictive than "v s V, vw e E" The first permits v to be isolated, the second does not
- "expr < $k_1$" is less restrictive than "expr < $k_2$" if $k_1 > k_2$.
- "expr £ k^' is less restrictive than "expr ≤ $k_2$" if $k_1 > k_2$.
- "expr > k.j" is less restrictive than "expr > $k_2$" if $k_1 < k_2$.
- "expr £ k^' is less restrictive than "expr ≥ $k_2$" if $k_1 < k_2$.
- "expr ^ k" is less restrictive than *"expr* < k".
- *"expr* ≤ k" is less restrictive than "expr = k".
- "expr £ k" is less restrictive than "expr > k".
- "expr £ k" is less restrictive than "expr = k".
- "d(v) = max" is consistent with "d(v) rel k" only if rel is = and k is max or rel is £, >, :£, < when k £ max, k < max, k £ max, k > max, respectively
- "d(v> < *max"* is consistent with "d(v) = k" only if k < max.
- 'd(v) < max" is consistent with "d(v) > k" only if there exists an integer i such that k < i < max
- "d(v) < max" is consistent with "d(v> £ k" only if k £ max.

We offer the following example of selector subsumption:

*oj* x,y 6 V, xy * E, d<x) £ 2

$a_2$: x,y s V, xy * E, d(y) > 2# d(x) = 1

$o_J$ subsumes $o$-$_2$ under the mapping <£(x) = y and <*t>(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

$f_1 - f^3 + fg + g^{**2} + g^{\#}g$

Then we can show that $f_1$ subsumes, among others, each of the following.

$f^3$

$g^5 f^2$

(f + g V

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):

$B^*_{xy}(K_i)$ where x s $V_{\#}$ y « V

and $p_2$ (CHAIN):

$B^*_{xy}(K_i)$ where x « V, y « V, d(x) = 1

$B_{xy}$ subsumes itself. $p_2^{-1}(K.)$ returns TRUE Define 0(x) « x, 0<y) = y. Thus our R-language "knows" the relationship between trees and chains. The RHanguage 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 $f_2$, $\sigma_1$ subsumes $\sigma_2$, 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:

$p_1$: $(B_{xy} + A_{wz})^*(K_3, K_4, K_5)$

where $x \in V$, $y \notin V$, $d(x) > 2$

$w, z \in V$

$p_2$: $B^*_{xy}(K_3)$ where $x \in V$, $y \notin V$, $d(x)$ 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 $\sigma$ subsumes the other and it is their combination we desire, i.e.,

p: $B^*_{xy}(K_3)$ where $x \in V$, $y \notin V$, $d(x) > 2$, $d(x)$ 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 $\sigma$ eliminates any references to variables not in $\sigma_2$. If $\sigma_2$ subsumes $\sigma_1$, $\sigma$ will be simply $\sigma_1$.

Consider next the example:

$p_1$: $(B_{xy} + A_{wz})^*(K_3, K_4)$

where $x \in V$, $y \notin V$

$w, z \in V$

$p_2$: $B^*_{xy}(K_3, K_5)$ where $x \in V$, $y \notin V$, $d(x)$ 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: $B^*_{xy}(K_3)$ where $x \in V$, $y \notin V$, $d(x)$ even

We can now postulate:

**PRINCIPLE 3**

If $f_1$ subsumes $f_2$, $\sigma_1$ subsumes $\sigma_2$ and $S_1 \cap S_2 \neq \phi$, then the merger p is $<f_2, S_1 \cap S_2, \sigma_2>$. 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:

$p_1$: $(A_x A_y A_z + A_{pq})^*(K_1)$

where distinct $x,y,z \notin V$

distinct $p,q \in V$

$p_2$: $(A_x A_y + A_{pq})^*(K_1)$

where distinct $x,y \notin V$

distinct $p,q \in V$

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

p: $(A_{x_1} A_{x_2} \ldots A_{x_6} + A_{pq})^*(K_1)$ where

distinct $x_i \notin V$, $i = 1,2,\ldots,6$

distinct $p,q \in V$

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 $\Delta n_i$, and the change in m as $\Delta m_i$. We define $\Delta n$ and $\Delta m$ correspondingly for property p.

**PRINCIPLE 4**

If p is the merger of $p_1$ and $p_2$, $\Delta n$ is the least common multiple of $\Delta n_1$ and $\Delta n_2$, and $\Delta m$ of $\Delta m_1$ and $\Delta m_2$.

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 $\Delta n_i$'s and $\Delta m_i$'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:

- $f^*_{a,b,\ldots} \Rightarrow (f_{a_1,b_1,\ldots} f_{a_2,b_2,\ldots} \ldots f_{a_k,b_k,\ldots})^*$

  This restricts the number of times f is applied within an iteration to some multiple of k. Subscripts are presumed distinct.

- $f^*_{a,b,\ldots} \Rightarrow f_{a_1,b_1,\ldots} f_{a_2,b_2,\ldots} \ldots f_{a_k,b_k,\ldots}$

  This fixes the number of times f is applied within an iteration to exactly k.

- $f^*_{a,b,\ldots} \Rightarrow f^+_{a,b,\ldots}$

  This denotes "at least one iteration is required."

- $f^* \Rightarrow N$

  This means f is not iterated at all.

- $(f + g)^* \Rightarrow f^*g^*$

  $(f + g)^* \Rightarrow g^*f^*$

  These require that the applications of f and g appear in a specific order.

- $(f + g)^* \Rightarrow (f + fg)^*$

  $(f + g)^* \Rightarrow (f + gf)^*$

  $(f + g)^* \Rightarrow (g + fg)^*$

  $(f + g)^* \Rightarrow (g + gf)^*$

  These insist that some alternatives may not occur alone.

- $(f + g)^* \Rightarrow f^*$

  $(f + g)^* \Rightarrow g^*$

  These eliminate an option.

- $f_{\alpha,\beta,\ldots} \Longrightarrow f_{a,b,\ldots}$

  This represents a consistent substitution of variable a for $\alpha$, b for $\beta$,..., within the constraints of $\sigma$. For example, if $\sigma$ 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 $f_2$, $\sigma_1$ and $\sigma_2$ 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 ($p_2$):

$p_1$: $B^*_{xy}(K_1)$ where $x \in V$, $y \notin V$

$p_2$: $(A_{xw} + A_yA_z)^*(K_1)$

where $x,w \in V$

distinct $y,z \notin V$

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 $\Delta n_1$ with $\Delta n_2$, and $\Delta m_1$ with $\Delta m_2$. First we push $f_1$ toward $\Delta n = 2$:

$f_1^* = (A_{xy}A_y)^* \Longrightarrow (f_1^2)^* = (A_{xy}A_yA_{wz}A_z)^*$

$\sigma_1 = x \in V, y \notin V \Longrightarrow x,w \in V$, distinct $y,z \notin V$

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 $\Delta m = 2$:

$f_2^* = (g + hh)^* = (A_{xw} + A_yA_z)^* \Longrightarrow (gghh)^* = (A_{xw}A_{pq}A_yA_z)^*$

$a_1$ = x,w 6 V, distinct y,z £ V => x,w,p,q e V, distinct y,z 2 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

- $f_{a,b,\ldots} f_{a,b,\ldots} \equiv f_{a,b,\ldots}$

  Note that the subscripts are identical hence the lack of impact on the graph.

- fN a f

  Nf a f

  The null operator may be ignored

- $f_{4b\ldots}$ r j... s N

  $f_{a,b,\ldots}$ $f'_{a,b\ldots}$ a N

  An operator and its inverse cancel each other out as long as they are applied to the same vertices/edges.

- f*f* s f*

  This is a notations! equivalent

- $(f + g)^{\#}$ 3 f*(f + g)*

  $(f + g)^{\#}$ • (f + g)V

  These are simplifications.

- (f + g) 3 (g + f)

  This is the inherent commutativity in the iteration choice.

- f i g where g is the defined primary equivalent of the composite f, a verification we assumed informally above.

- $A_x f_{ab\ldots}$ a $f_{ab\ldots} A_x$ if *a* prevents x from being a,b,.~.

  This is a very limited form of commutativity.

- $f_{\alpha,\beta,\ldots}$ $f^*_{b\ldots}$ a $f^* Q_{\alpha,\ldots}$ if *a* permits a = a, p = b,.~

  $Cb\ldots. f^*_{a,p\ldots.}$ *m* C j U if o permitS a = 3, P = b -

  These are principles of absorptioa

- $ff_{a_1,b_1,\ldots} {}^{a_2,b_2,\ldots 2} \wedge u_{a,b,\ldots}$ *Ma* can be changed to select variables

appropriately.

Now continuing with our example, we can rewrite $f_2$ (and $\sigma_2$) 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

$(A_{xy}A_{wz}A_yA_z)^*$ where x,y,w,z ∈ V, distinct y,z ∉ V

Because y and z are added during the iteration, y,z ∈ V is irrelevant and we now have a specialization of $f_2$ and $\sigma_2$ that is

$(A_{xy}A_{wz}A_yA_z)^*$ where x,w ∈ V, distinct y,z ∉ V

When we contrast this with the specialization of $f_1$ and $\sigma_1$:

$(A_{xy}A_yA_{wx}A_z)^*$ where x,w ∈ V, distinct y,z ∈ 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: $(A_{xy}A_yA_{wz}A_z)^*(K_1)$ where x,w ∈ V, distinct y,z ∉ V

As our next real example, we offer the merger for complete ($p_1$) Eulerian ($p_2$) graphs:

$p_1$: $F_x^*(K_1)$ where x ∉ V, distinct $v_i$ ∈ V, $|\{v_i\}| = n$

$p_2$: $(S_{wvz} + Y_{v_1 \ldots v_k})^*(K_3)$

where w,z ∈ V, wz ∈ E,

$|\{v_i\} \cap V| \geq 1$, distinct $v_i \in V$, $v_iv_{i+1}, v_kv_1 \notin E$, $i = 1,2,\ldots,k$

We rewrite $f_1$ and $f_2$ as:

$f_1$: $A_{xv_1} \ldots A_{xv_n}A_x$

$f_2$: $D_{wz}A_{wv}A_{vz}A_z + A_{v_1v_2} \ldots A_{v_{k-1}v_k}A_{v_kv_1}A_{v_1} \ldots A_{v_k}$

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 $\Delta n_2$ to 1, since all the cycle edges must be new to the graph. Thus we specialize $f_1$ (and $\sigma_1$) to $f_1^2$:

$f_1$: $F_xF_y$ distinct x,y ∉ V, distinct $v_i$ ∈ V, $|\{v_i\}| = n$

Now $\Delta n_1 = 2$, $\Delta m_1 = 2n + 1$. A specialization of $f_2$ is

$Y_{v_1 xy} Y_{v_2 x} Y_{v_3 x} \ldots Y_{v_i x} V$

This set of cycle additions is equivalent to the specialized $f_y$ 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 euierian graphs, is

p: $(F \wedge F_y)^*(K_3)$ where distinct x,y * V, distinct $v_i$ e 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^ bipartite ($p_2$) graphs:

$p_r$: $B^*_{xy}(K \wedge$ where x s v

$p_2$: $(A_x + A_{xx}A_x + A_{yz})^*(<\{1,2\}.\{11\}>)$

where x * V

y,z * V, $|\{yy.zz\}|$ n E| » -1

$f_1$ may be rewritten as $A \wedge_y A \wedge$. Noting that $An_1 = 0$ or 1 (depending on whether y is or is not already in V), $Anr \wedge = 1$, we have the following alternatives from $f_2$:

$\&n_2 = 1$ $Am_2 * 0$

$An_2 s 1$ $Am \wedge s 0$

$An_2 * 0$ $Am_2 = 1$

Note that we choose *not* to count loops in any Am. The first two alternatives require specialization to match $f_1$, so we specialize $f_2$ (and *aj* to:

$f_2$: $A_{xv} A_x + A_{xw} A_{xx} A_x + A_{yz}$

where x * V, v € V, w « E

x ^ V, w s v, ww *0* E

y,z s V, |{yy,zz} n E| « 1

Examining $f_1$, we see that $B_{xv}$ is equivalent to the first alternative (for v e V), $B_w B_{xv}$ to the second (for v * V), and $B_{yz}$ (for y,z e V) to the third Thus $f_1$ is equivalent to the specialized *f*^ The seed for the merger is the minimal bipartite connected graph <{1,2}/{1 U 2}>. (Again, we refer the reader to Chapter 5 for a discussion of seed choice.) The final merger, to generate connected, bipartite graphs

is

p: $(A_{xv}A_x + A_{xw}A_{xx}A_x + A_{yz})^*(<\{1,2\},\{11,12\}>)$

where $x \notin V$, $v \in V$, $vv \in E$

$x \notin V$, $w \in V$, $ww \notin E$

$y,z \in V$, $|\{yy,zz\} \cap E| = 1$

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 ($p_1$):

$p_1$: $(A_{xy} + A_wA_z)^*(K_1)$ where $x,y \in V$, distinct $w,z \notin V$

$p_2$: $(OM_{vwv_1\cdots v_{k+1}w_1\cdots w_{k-1}} + A_{y_2q}A_{y_1p}D_{y_1y_2}CM_{y_1\cdots y_{k+1}}D_{pq} + FR_{z_1\cdots z_k})^*$

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

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

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

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

distinct $x_1,x_2,\ldots,x_k,z_1 \in V$, distinct $z_2,z_3,\ldots,z_k \notin V$, $z_1x_i \in E$, $i = 1,2,\ldots,k$

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 $k+1$, $k+3$,..., 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 $L_\Omega$, 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:

$A^*_{xy}(C)$ 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.,

$C = \{C_k \mid k = 1,2,\ldots\}$.

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:

$f^{-1} = A^{-1}_{xy}$

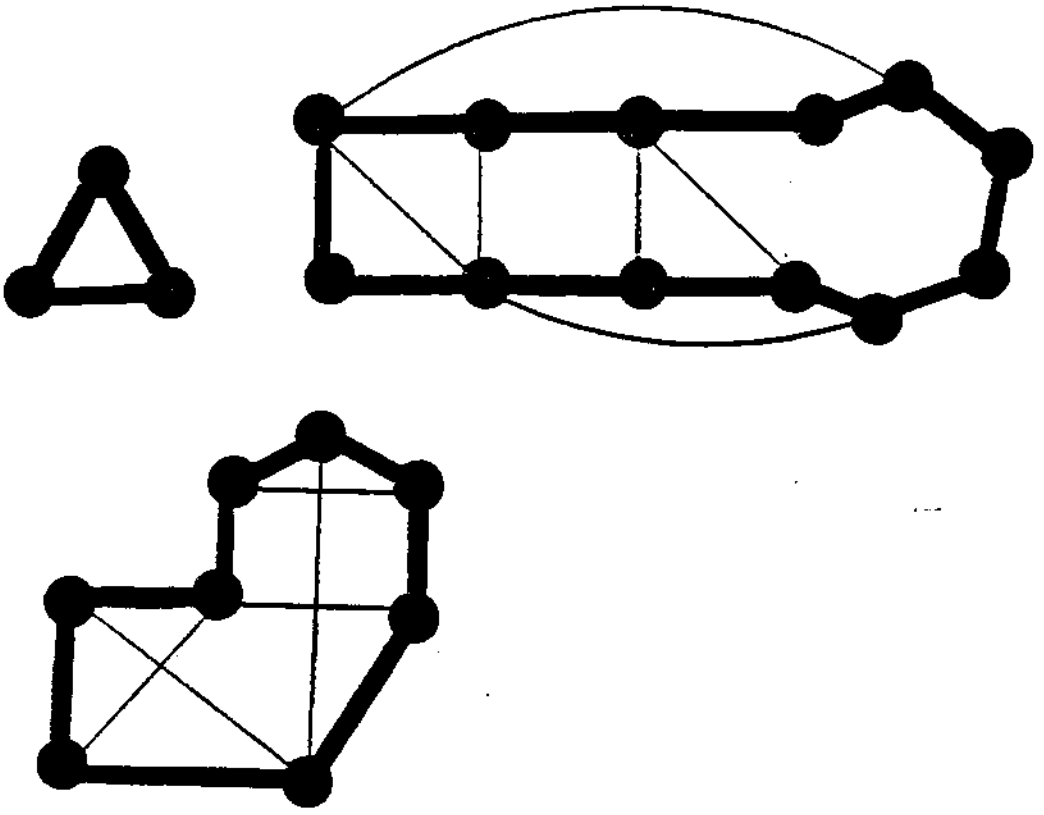


**Figure 4-50:** Some Hamiltonian Graphs

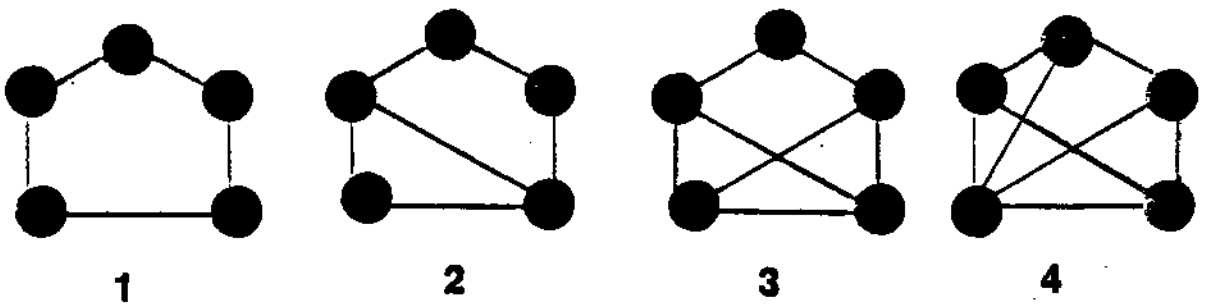



**Figure 4-51:** HAMILTONIAN in Operation

= D
xy

$\sigma_{pre}$
$\sigma^{-1}$ = x,y e V, xy * E, d(x) £ 2, d(y) £ 2

= x,y e V, xy 6 E,

there exists some largest cycle not including xy

The floors shift to `<$P_r$<$P_2$^2,$Z_1$>,$E_6$> and <$P_2$'<P2'Lm^i>^8>* Rgure 4-52 shows HAMILTONIAN""[1] operating on a graph G € $G_p$ and a graph G * $G_p$. If xy does not

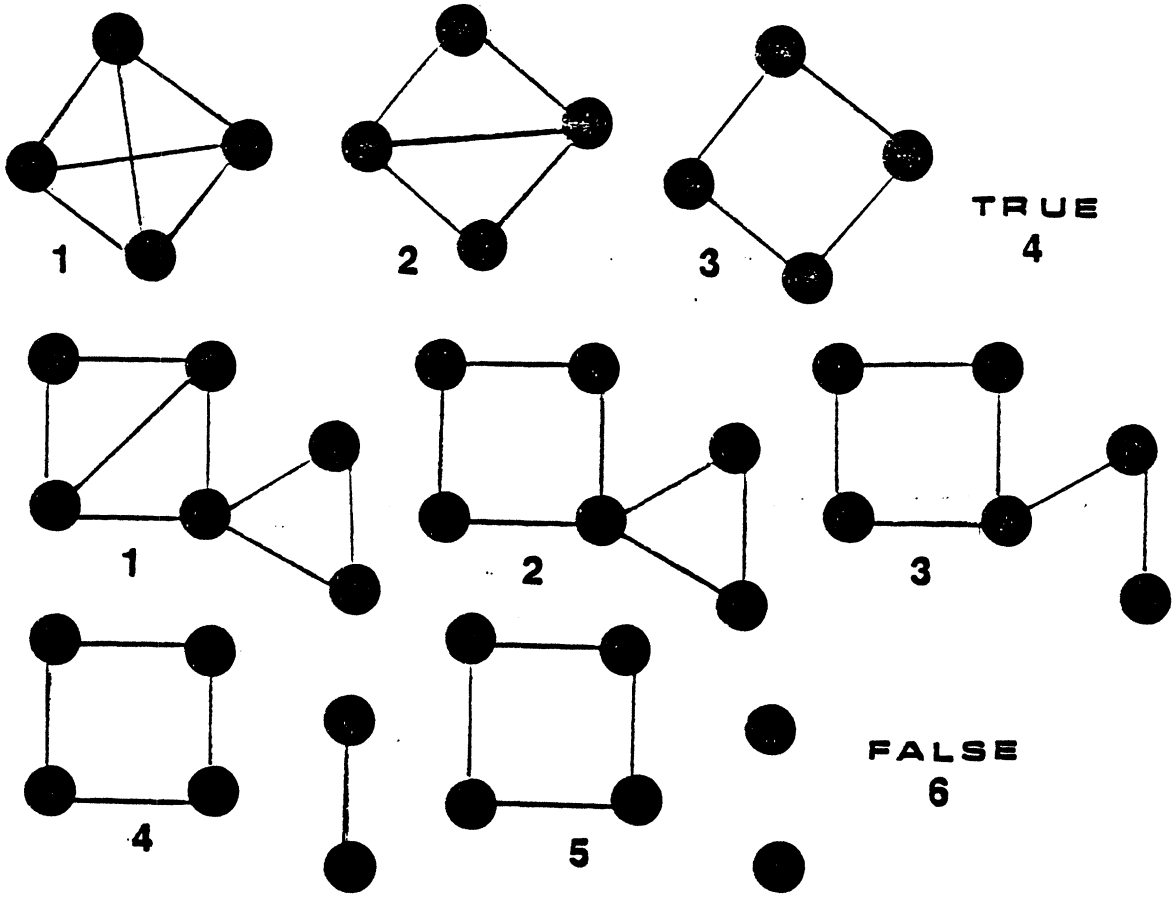



**Figure 4-52:** HAMILTONIAN$^{-1}$ in Operation

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 $\sigma$, i.e,

$$A^*_{xy}(S^*_{wvz}(K_3))$$

where $x,y \in V$

$w,z \in V$, $v \notin V$, $wz \in E$

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 $\sigma$

$y,z \in V$

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 $G = \langle V,\phi \rangle$. 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 $y,z \in V$

$v \in V,\ w \notin V,\ d(v) < k$

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 $\Sigma$-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 $L_{\Omega}$. In some instances, although the L-language is $L_{\Omega}$, 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 $L_{\Omega}$ 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 $CM_{v_1 \cdots v_{k+1}}$ 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 $\sigma_{pre}$ and the preservation after f of the information $\sigma_{pre}$ contained. If we cannot construct $\sigma_{pre}$ 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 $\Sigma$) state what data is explicitly represented and representable. If, for example, a graph were characterized in L by describing its cycles and $\Sigma$ 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 sets behavior under simple manipulations on a fixed number of vertices. We explored the edge-set languages $L_r$ $L_{1n/}$ $L_2$, $L_{2n}$, $L_3$, $L_{3n}$ 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 ail 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,... | 4 | 12 |
| $L_{2n}$, n ≤ 25 | 27 | 106 |
| $L_{3n}$, n ≤ 25 | 229 | 259 |

| **Directed Language** | **Properties** | **Characterizations** |
|---|---|---|
| $L_{1n}$, n = 1,2,... | 6 | 24 |
| $L_{2n}$, n ≤ 25 | 202 | 4849 |
| $L_{3n}$, n ≤ 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 $<P,L,\Sigma>$ 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 $<P_1,L_1,\Sigma_1>$ 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 $<P,L,\Sigma>$ 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 RHanguage, 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 <f,S,a> description for a property

or where

- we could find an <f,S,a> 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 RHanguages, 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 RHanguage 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 <P,L,Σ> Formulation

In effect, we developed a hierarchy of R-languages. Each language is based on a triple <P,L,Σ>, 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 $<P_1,L_1,\Sigma_2>$ is less complex than $<P_2,L_3\Sigma_3>$, but not comparable with $<P_1,L_2,\Sigma_1>$. 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 $L_\Omega$. This reliance on $L_\Omega$ may be an intrinsic limitation of the edge-set languages as we define them. The Σ-languages are adequate, although $\Sigma_6$ 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

| | | | |
|---|---|---|---|
| $\langle P_4, L_\Omega, \Sigma_5 \rangle$ | $\langle P_4, L_2, \Sigma_5 \rangle$ | $\langle P_4, L_{1n}, \Sigma_5 \rangle$ | $\langle P_2, L_{1n}, \Sigma_5 \rangle$ |
| K-EDGES | BICONNECTED | BICONNECTED | K-INDEPENDENT |
| | | K-CONNECTED | EULERIAN |
| $\langle P_2, L_\Omega, \Sigma_5 \rangle$ | $\langle P_2, L_\Omega, \Sigma_2 \rangle$ | $\langle P_4, L_{1n}, \Sigma_3 \rangle$ | $\langle P_2, L_2, \Sigma_5 \rangle$ |
| PINWHEEL | EVEN-REGULAR | K-COMPONENTS | EULERIAN |
| | ODD-REGULAR | CONNECTED | |
| $\langle P_2, L_{1n}, \Sigma_4 \rangle$ | $\langle P_2, L_2, \Sigma_2 \rangle$ | $\langle P_2, L_{1n}, \Sigma_2 \rangle$ | $\langle P_3, L_{1n}, \Sigma_2 \rangle$ |
| WHEEL | CYCLE | CYCLE | MAX |
| $\langle P_2, L_{1n}, \Sigma_1 \rangle$ | $\langle P_2, L_3, \Sigma_1 \rangle$ | $\langle P_1, L_2, \Sigma_5 \rangle$ | $\langle P_1, L_{1n}, \Sigma_5 \rangle$ |
| CHAIN | CHAIN | BIPARTITE | BIPARTITE |
| | | COMPLETE- | COMPLETE- |
| | | BIPARTITE | BIPARTITE |
| | | | K-VERTEX-COVERED |
| $\langle P_1, L_\Omega, \Sigma_4 \rangle$ | $\langle P_1, L_\Omega, \Sigma_3 \rangle$ | $\langle P_1, L_{1n}, \Sigma_3 \rangle$ | $\langle P_1, L_1, \Sigma_5 \rangle$ |
| STAR | MAX-K | MIN-K | COMPLETE |
| | | $CHAIN_2$ | |
| $\langle P_1, L_3, \Sigma_3 \rangle$ | $\langle P_1, L_3, \Sigma_2 \rangle$ | $\langle P_1, L_{1n}, \Sigma_2 \rangle$ | $\langle P_1, L_1, \Sigma_2 \rangle$ |
| ODD-M | EVEN-N | EVEN-N | ODD-N |
| $CHAIN_2$ | ODD-M | ODD-N | LOOPFREE |
| | | ODD-M | EVEN-M |
| | | | DEGREE |

**Figure 5-1:** Graph Properties with Edge-Set L-Language Grouped by Floors

| $<^{p}r^{L}mV$ | $\wedge r V$ | $\wedge r \wedge$ |
|---|---|---|
| K-VERT1CES | 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 LHanguages appears in Figure 5-2. This hierarchy is based only on P-languages and ^-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 $P_3$ for generation.
- $Z_3$ might be replaced with two E-languages to improve the differentiation between $E_4$ and $E_5$.
- 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 $P_y$ 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 E from p to $p^{-1}$ is somewhat more difficult to deal with. With

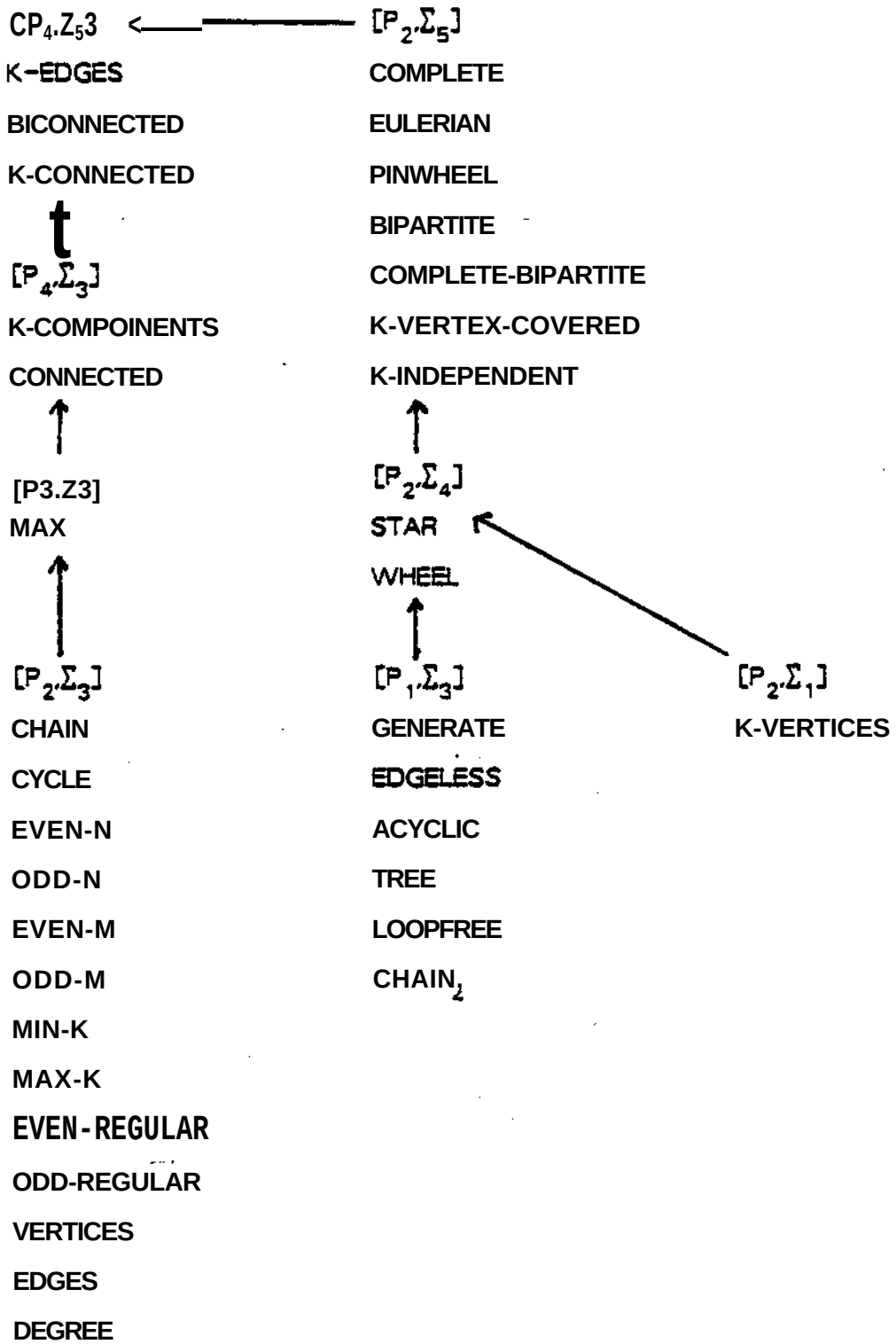



**Figure* 5-2: Graph Properties with Edge-Set L-Language Ranked by P-Language and I-Language**

the exception of K-VERTICES (which never deletes a vertex) every inverse requires a $\Sigma$-language of at least $\Sigma_3$. With the exception of EULERIAN, no inverse has a simpler $\Sigma$-language than its generator. With the exception of CIRCUMFERENCE-K, a $\Sigma$-language no more complex than $\Sigma_3$ 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 $\Sigma$-languages no more complex than $\Sigma_4$. An R-language using $\Sigma_5$ could probably be automated by skillful programming. The few properties with inverses in $\Sigma_6$, 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.

---

[1]The 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 $\sigma$, how do we find a common seed set when $S_1 \cap S_2 = \phi$ ?" 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 $\Sigma_1$, $\Sigma_2$, $\Sigma_3$ or $\Sigma_4$ can certainly be achieved in linear time with properly constructed (not necessarily linear) storage. Because $\Sigma_5$ and $\Sigma_6$ 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 $\Sigma_1$, $\Sigma_2$, $\Sigma_3$ and $\Sigma_4$, but probably not for many instances of $\Sigma_5$ and $\Sigma_6$.

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.

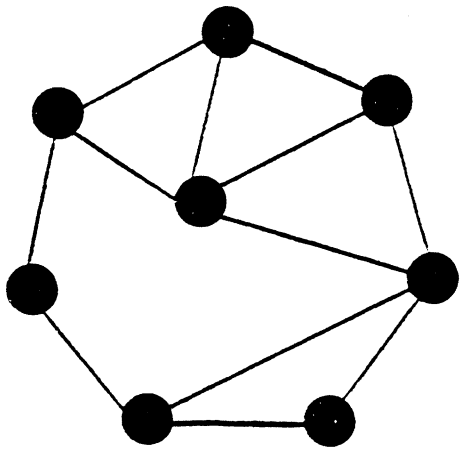


**Figure 5-3:** A Graph with Variable Testing Time

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) graphs[2]. 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

[2]This 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

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

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 RHanguage 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 $K_g$ 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 Kg. 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.

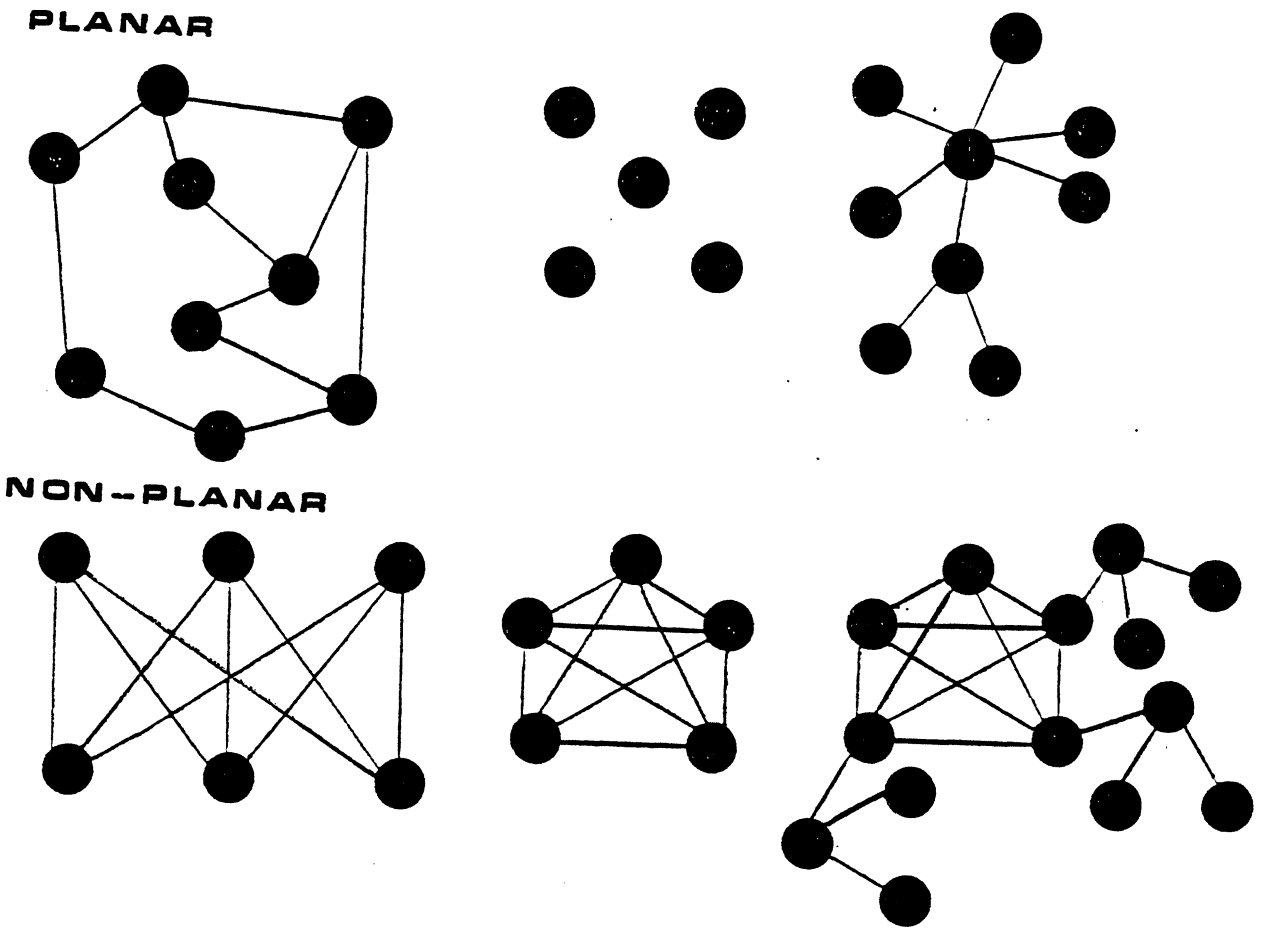



**Figure 5-4:** Some Graphs and Their Planarity

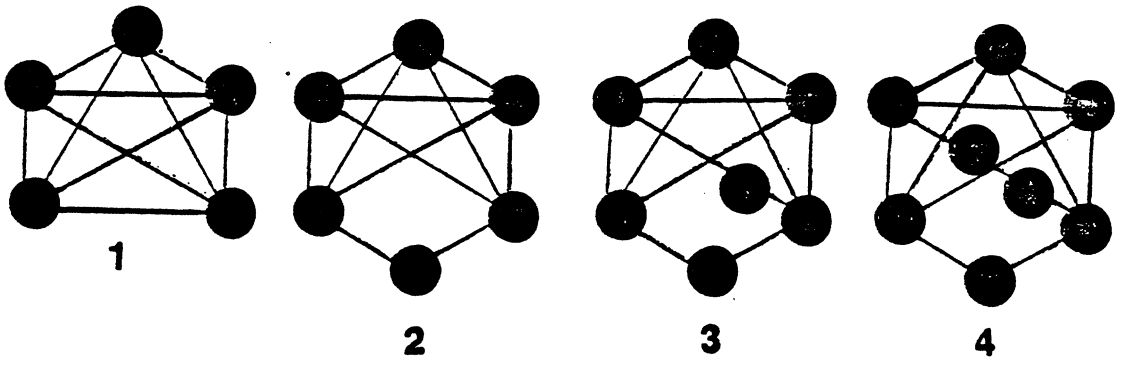



**Figure 5-5:** The Construction of a Homeomorph to $K_5$

The algorithm NON-PLANAR is

$$(A_x + A_{yz} + S_{pvq})^*(K_5, K_{3,3})$$

where $y,z \in V$

$p,q \in V,\ v \notin V,\ pq \in E$

Figure 5-6 shows the iterative steps in a sample run of NON-PLANAR. The floor for non-planar graphs is $\langle P_2, L_\Omega, \Sigma_1 \rangle$.

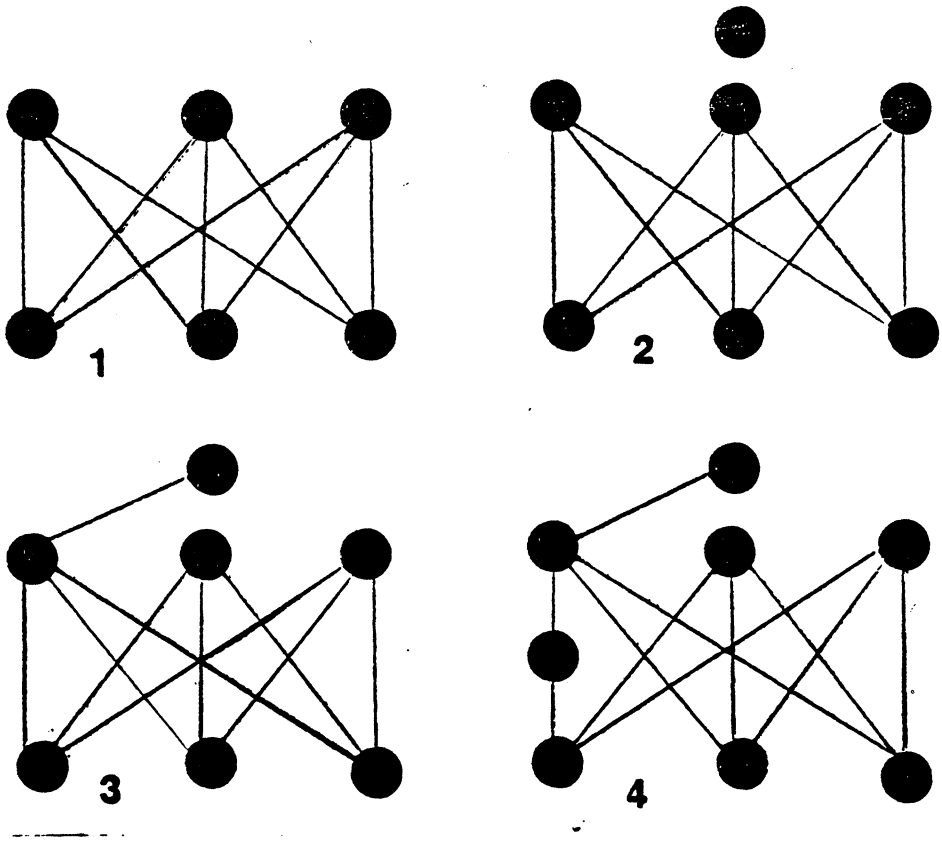



**Figure 5-6:** A Sample Run of NON-PLANAR

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 $\Sigma_6$-based formulation:

$$f^{-1} = (D_x + D_{yz} + D_v D_{vq} D_{pv} A_{pq})$$

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

$$y,z \in V,\ yz \in E,\ yz \text{ is not in every subgraph of } G$$

$$\text{homeomorphic to } K_5 \text{ or } K_{3,3}$$

$$p,v,q \in V,\ pv,vq \in E,\ pq \notin E,\ d(v) = 2$$

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) chaia (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

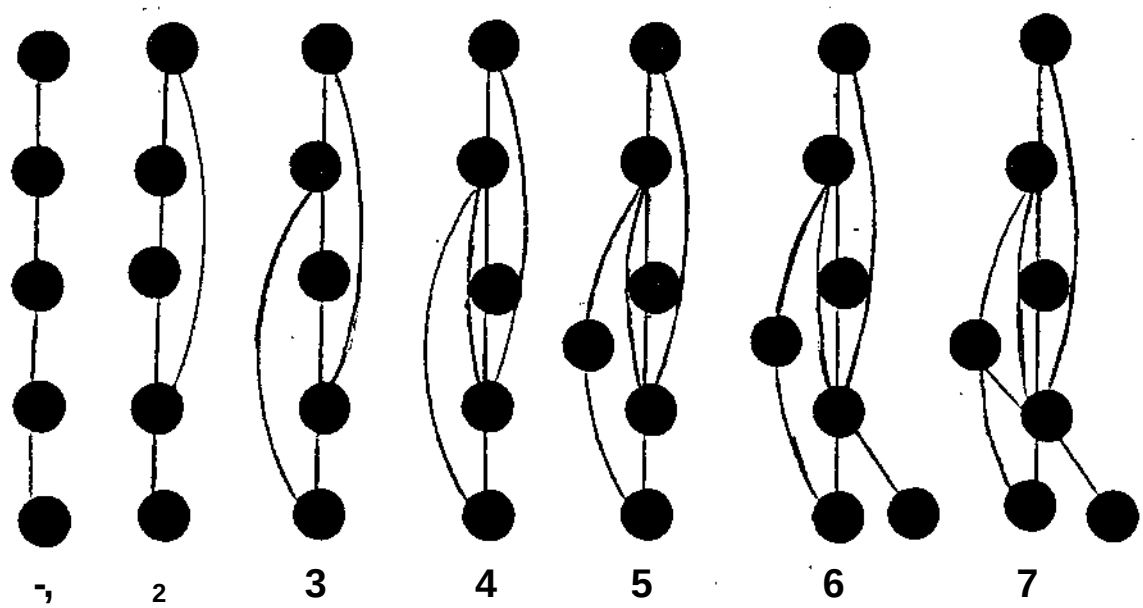



**Figure 5-7:** A Sample Run of PLANAR

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 $\langle p_2, L_{in}, i \rangle$ and $\langle P_2, L_2, 1 \rangle$ is complete and correct

where y,z s v

p.q s V, v 0 V, pq s E

it too bears a disappointingly unclear relationship to its opposite, the algorithm ACYCLIC with floor $\langle P_1.L_rE_1 \rangle$ of 3.7.1:

$B^*_{xy}(\langle V,* \rangle)$ where x *m* V, y * 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.)

[3]In 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 = <f,S,$\sigma$> 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 $\lambda$-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 $\sigma$-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 $\Sigma_1$, $\Sigma_2$, $\Sigma_3$ or $\Sigma_4$ 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 |
| $C_k$ | Cycie 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 |
| $F_{xvv_1v_2\sim v_k}$ | Fragment vertex x into vertices x and y | 66 |
| $FR_{v_1--v_k}$ | Fracture vertex $v_1$ into $K_R$ maintaining previous adjacencies | 131 |
| G | "Graph | 12 |
| $'^{xv}r\text{-}^{v}k$ | Identify vertices $x,v_r...,v_k$ | 66 |
| $K_k$ | Complete graph on k vertices | 101 |
| L | Loop on ail vertices | 66 |
| L | Unloop on all vertices | 66 |
| L | 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 |
| 0 | Order complexity | 69 |
| P | Primitive language | 64 |
| $Q_k$ | Quantity of disjoint complete graphs | 132 |
| $S_{xvy}$ | Subdivide edge xy by vertex v | 63 |
| $S_{xvwy}$ | Subdivide edge xy by vertices v and w | 171 |
| T | Replacement system for testing equivalence of L-expressions | 14 |
| | Complete graph on k different-colored vertices | 171 |
| U | Set of ail finite graphs closed under isomorphism | 13 |
| $U_k$ | Edgeless graph on k different-colored vertices | 168 |
| V | **Vertex set of the nranh** | **1O** |

| | | |
|---|---|---|
| $W_{h,r}$ | Pinwheel on h hubs and r rims | 123 |
| $X_{v_1^{s_1}\cdots v_r^{s_r}}$ | Surrogate operator | 171 |
| $Y_{u_1\cdots u_k}$ | Add cycle $u_1u_2 \ldots u_ku_1$ to the graph | 68 |
| $\underline{Y}_{u_1\cdots u_k}$ | Delete cycle $u_1u_2 \ldots u_ku_1$ from the graph | 68 |
| Z | Label vertex x with <x | 167 |
| $Z_{xy\alpha}$ | Label edge xy with a | 184 |
| *a* | Selector, element of I | 64 |
| I | 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.
        INTEGER N,A,C,D,F,I,LAST
        INTEGER S(27),FAKE,G(107),CT,MAT(107),THISC(107)
        INTEGER ISUM(25),IMAX(25)
C       DUMMY VARIABLES
        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
      00 1 K-1,27
      FAKE-FAKE*2+1
    1 CONTINUE
      G(1)»FAKE

C     GENERATE HEADER AND FIRST DATA LINE
      TYPE 2
    2 FORMATC VERTICES  CASES     CLASSES   LARGEST   DENSITY[1])
      TYPE 1*50, ZERO ,CT,CT,CT,HUNO

C     MAJOR LOOP ON I » # VERTICES
      DO 500 I«1,N
      00 5 K - 1,107
            THISC(K) - 0
    5 CONTINUE
      T=i*(I-1)/2

C     LOOPS ON A ANO 0 TO CREATE CASES
      DO 400 A-O.T
      DO 300 0-0,1
C     VALUES FOR C ANO F CACULATEO FROM A,D AND I
            C-T-A
            F-l-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.O) S(2)-1
            IF (C.EQ.O) S(3)-1
            IF (D.EQ.O) S(i»)-1
            IF (F.EQ.O) S(5)-1
            IF (A.EQ.C) S(6)-1
            IF (A.EQ.O) S(7)-1
            IF (A.EQ.F) S(8)-1
            IF (C.EQ.O) 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(U)-1
            IF (A+O.EQ.C) S(15)-1
            IF (C+D.EQ.A) S(16)-1
            IF (C+O.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+O.EQ.C+F) S(23)-1
                IF (A+F.EQ.C+O) S(24)-1
                IF (A.EQ.C+O+F) S(25)-1
                IF (C.EQ.A+O+F) S(26)-1
                IF (A+C.EQ.O) S(27)-1

C       PACKING SIGNATURE S INTO FAKE TO SAVE SPACE, 1 GROUP OF 30
                FAKE-0
                DO kk L1-1.27
                FAKE-FAKE*2+S(L1)
   kk           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 -I.
C       ONCE SIGNATURE RECURS, MAT(FOO) IS NUMBER OF DIFFERENT I
C       VALUES FOR WHICH SIGNATURE OCCURS.
                IF (MAT(F2) .GT.O) MAT(F2)-MAT(F2)+1
                F3 - -I
                IF ((MAT(F2) .LT.O) .ANO. (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) — I
                THISC(CT)»1
                LAST-I
  300           CONTINUE
  1*00          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 kkS FF-1.CT
                IF (THISC(FF) .GT.O) ISUM(I)-ISUM(I)+1
                IF (THISC(FF) .GT.IMAX(I)) IMAX(I)-THISC(FF)
  Mt5           CONTINUE

C       COMPUTE ANO PRINT OUTPUT LINE FOR I
                IDUM=(I+1)*(1+I*(I-1)/2)
                IDUM2-100.0*IMAX(I)/1DUM+.5
                TYPE 450,I,IDUM,ISUM(I),IMAX(I),IDUM2
  450           FORMAT(5MO)
  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
```

### 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.CMD.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 OCCURING CASE.
        INTEGER N,A,C,D,F,I,LAST
        INTEGER S(90),FAKE(3),G(5000,3),CT,MAT(5000),THISC(5000)
        INTEGER ISUM(25),IMAX(25)
C       DUMMY VARIABLES
        INTEGER T,K,F2,F3,HUND,IDUM,IDUM2,FF,ZERO,D2
```

```
      DATA N,A,C,D,F,I,CT,LAST/8*0/
      DATA HUND,2ER0/100,0/
      DATA (S(J) ,J«1,9O)/9O*O/
      DATA (FAKE(J) ,J»1,3)/3*O/
      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*O/
      DATA (I MAX (J),J-1,25)/25*0/
      N-25

C     INITIALIZE CASE FOR GRAPH ON NO VERTICES
      CT - 1
      F2-0
      00 1 K-1,30
      F2-F2*2+1
    1 CONTINUE
      DO 2 J-1,3
      G(1,J)-F2
    2 CONTINUE

C     GENERATE HEADER ANO FIRST OATA LINE
      TYPE 3
    3 FORMAT (' VERTICES  CASES     CLASSES   LARGEST   DENSITY[1])
      TYPE i»50,ZERO,CT.CT.CT,HUND

C     MAJOR LOOP ON I - # VERTICES
      00 500 I-1.N
      DO 5 K - 1,5000
            THISC(K) - 0
    5 CONTINUE
      T=|*(|-1)/2

C     LOOPS ON A ,B ANO 0 TO CREATE CASES
      00 400 A-0,T
      00 300 0-0,l
            D2- (T-A)/2
      00 200 B-0,D2
C     VALUES FOR C ANO F CACULATEO
            C-T-A-2*B
            F-I-O
C     ZERO OUT S IGNATURE
      00 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+O.EQ.F) S(12)»1
IF (A+F.EQ.C) S(13)-1
IF (A+F.EQ.O) S(14)»1
IF (A+O.EQ.C) S(15)-1
IF (C+O.EQ.A) S(16)-1
IF (C+O.EQ.F) S(17)»1
IF (C+F.EQ.A) S(18)-1
IF (C+F.EQ.D) S(19)-1
IF (O+F.EQ.A) S(20)»1
IF (O+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+O) S(2*0-1
IF (A.EQ.C+O+F) S(25)-1
IF (C.EQ.A+O+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(3D-1
IF (B.EQ.F) S(32)-1
IF (A.EQ.B+C) S(33)-1
IF (A»EQ.B+D) S(3*0-1
IF (A.EQ.B+F) S(35)-1
IF (B.EQ.A+C) S(36)-1
IF (B.EQ.A+O) S(37)-1
IF (B.EQ.A+F) S(38)-1
IF (B.EQ.C+O) S(39)»1
IF (B.EQ.C+F) S(MJ)-1
IF (B.EQ.O+F) S(i»1)-1
IF (C.EQ.A+B) S(l»2)-1
IF (C.EQ.B+O) S(43)-1
IF (C.EQ.B+F) S(U)»1
IF (O.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+O) S(48)-1
IF (A+B.EQ.C+F) S(i»9)»1
IF (A+B.EQ.D+F) S(50)-1
IF (A+C.EQ.B+O) S(5D-1
IF (A+C.EQ.B+F) S(52)-1
IF (A+D.EQ.B+C) S(53)-1
IF (A+O.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.O+F) S(57)-1
IF (B+O.EQ.C+F) S(58)-1
IF (B+F.EQ.C+O) S(59)-1
IF (A+B+C.EQ.0) S(60)»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 -I.
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=1,3
                G(CT,FF)=FAKE(FF)
   60           CONTINUE
                MAT(CT)=-1
                THISC(CT)=1
                LAST=I
  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 ',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
```

## 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                              TIME-LIMIT: 10:00:00
        UNIQUE:  Yes                                BATCH-LOG:  Append
        RESTART: Yes                                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.CMD.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   VERTICES  CASES     CLASSES   LARGEST   DENSITY
10:39:09 USER          0        1         1         1       100
10:39:09 USER          1        2         2         1        50
10:39:09 USER          2        6         6         1        17
10:39:09 USER          3       24        20         2         8
10:39:09 USER          4       80        78         2         3
10:39:10 USER          5      216       141         8         4
10:39:13 USER          6      504       336        24         5
10:39:23 USER          7     1056       484        48         5
10:39:49 USER          8     2025       956        76         4
10:40:31 USER          9     3610       911       196         5
10:42:09 USER         10     6072      1065       416         7
10:44:27 USER         11     9744      1045      1086        11
10:48:01 USER         12    15028      1750      2496        17
```

```
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 25
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 OCCURING CASE.
        INTEGER N,A,C,D,F,I,LAST
        INTEGER S(14),S1(14),S2(14),Q(27),G(300),CT,MAT(300)
        INTEGER THISC(300),FAKE,ISUM(25),IMAX(25)
C       DUMMY VARIABLES
        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 CACULATED 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 ASSEMBLEED
            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 -I.
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 = -I
                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)=-I
                THISC(CT)»1
                LAST-I
  300           CONTINUE
  400           CONTINUE

C       CASE BY CASE OUTPUT ROUTINE.
C       I MAX IS THE LARGEST CLASS SIZE FOR FIXED I.
C       I SUM IS THE NUMBER OF CASES OCCURRING FOR FIXED I.
C      CALCULATING I MAX AND I SUM FROM THISC.
                DO 445 FF-1.CT
                IF (THISC(FF) .GT.0) I SUM (I)-I SUM(I)+1
                IF (THISC(FF) .GT.IMAX(I)) IMAX (I) »THI SC (FF)
  445           CONTINUE

C       COMPUTE AND PRINT OUTPUT LINE FOR I
                IDUM=(I+1)*(1+I*(I-1)/2)
                IDUM2-100.0*I MAX(I)/IDUM+.5
                TYPE 450, I ,IDUM, I SUM (I) , IMAX (I) ,IDUM2
  450           FORMAT (5110)
  500           CONTINUE

C        SUMMARY STATISTICS
         TYPE 520,CT
  520    FORMAT (' NUMBER OF SIGNATURES IS M5 )
         F2-0
C       THE SIGNATURE FOR I-O IS UNIQUE TO THAT I VALUE.
         DO 525 K-2.CT
         IF (MAT (K) .GT.O) 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 M5 )
         TYPE 550,LAST
  550    FORMAT (' LAST NEW SIGNATURE OCCURS AT I » M5 )
         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                        TIME-LIMIT:  2:00:00
       UNIQUE:  Yes                          BATCH-LOG:   Append
```

```
        RESTART:  Yes                                   ASSISTANCE:  Yes
                                                        SEQUENCE:    2102

        Input 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-SRIOHARAN
22:51:06 MONTR   Job 12 on TTY254 28-0ec-82 22:51:06
22:51:07 MONTR   Last login on 28-0ec-82 at 13:07:41
22:51:07 MONTR   End of C0MAN0.CMD.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.F0R
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 L3 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 SIGNATURRE,
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 OCCURING CASE.
        INTEGER N,A,C,D,F,I,LAST
        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,MI,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 CACULATED
```

```
                C=T-A-2*B
                F=1-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
                MI=Y
                DO 25 Z=Y+1,30
                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(M1)
                S1(M1)-TEMP
   30           CONTINUE
                Q(30)-S2(30)
C       COMPARING Q VALUES BY BITS
                DO kO J-1,29
                Q(J+30)-0
                Q(J)-S2(J)
                IF (S(S2(J)).EQ.S(S2(J+1)))  Q(J+30)-1
   kO           CONTINUE
C       PACKING SIGNATURE Q INTO FAKE TO SAVE SPACE, 6 GROUPS OF 5
C       AND 1 GROUP OF 29
                00 kk 11-1,6
                FAKE(L1)»Q((L1-1)*5+D
                DO U3 L2-(L1-1)*5+2,L1*5
                FAKE(11)-FAKE(L1)*100+Q(L2)
   43           CONTINUE
   kk           CONTINUE
                FAKE2-0
                DO kS L1-31.59
                FAKE2-FAKE2*2+Q(L1)
   k$           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 -I.
C       ONCE SIGNATURE RECURS, MAT(FOO) IS NUMBER OF DIFFERENT I
C       VALUES FOR WHICH SIGNATURE OCCURS.
                1F (MAT(F2).GT.0) MAT(F2)-MAT(F2)+1
              .  F3 - -I
                IF ((MAT(F2).LT.O).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) — I
                THISC(CT)-1
                LAST-I
                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 I MAX AND I SUM FROM THISC.
                DO 445 FF-1.CT
                IF (THISC (FF) .GT.O) I SUM (I) -I SUM (I )+1
                IF (THISC(FF) .GT. IMAX(I)) I MAX (I) -THISC (FF)
  445           CONTINUE

C .     COMPUTE AND PRINT OUTPUT LINE FOR I
                F2-T/2*2
                IF (F2.EQ.T) IOUM»(T**2/4+T+i)ft(|+i)
                IF (F2.NE.T) IDUM-((T+1)**2/4+(T+1)/2)*(l+1)
                10UM2-100.0*I MAX(I)/10UM+.5
                TYPE 450,I,IDUM,I SUM (I),I MAX(I),IDUM2
  450           FORMAT (5110)
  500           CONTINUE
        GOTO521

C       SUMMARY STATISTICS
  519   TYPE 520
  520   FORMAT (' 20000 SIGNATURES DISCOVERED, MATRIX FULL[1])
  521   TYPE 522,CT
  522   FORMAT (' NUMBER OF SIGNATURES IS ',15)
        F2-0
C       THE SIGNATURE FOR I-O IS UNIQUE TO THAT I VALUE.
        00 525 K-2.CT
        IF (MAT (K) .GT.O) 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 - \I5)
        END
```

### E.2. L3DI Output

The following is the output listing from program L30I.

```
                        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                        TIME-LIMIT:  1:00:00
       UNIQUE:  Yes                          BATCH-LOG:   Append
       RESTART: No                           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.CMD.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   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


C 75] *An Intreduction to Model/ing Using Mixed integer Programming*
third edition, IBM, Amsterdam, 1975.

[Amarei 81] Amarei, 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.

[Angiuin 79] Angiuin, 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.

[Corneil 70] Corneil, D. and Gotlieb, C
An Efficient Algorithm for Graph Isomorphism.
*JACM* 17(1*51-64, January, 1970.
Used for general reference only. Not cited

[Fahiman 77] Fahiman, &
*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 251-283chapter 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, N.J., 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

**Susan Lynn Epstein**

| | |
|---|---|
| 1944 | Born April 8, 1944 in New York, New York |
| 1961 | Graduated New Rochelle High School New Rocheile, 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 Montciair State College, Montciair, New Jersey |
| 1978-1983 | Attended Department of Computer Science, Rutgers University, New Brunswick, New Jersey |
| 1981 | Contributor, *The Handbook of Artificial Intelligence,* 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 | PftD. in Computer Science, Rutgers University |