Combinatorial Algorithms on Partially Ordered Sets

by

Yasunori Koda

M.Math., University of Waterloo, 1987

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

We accept this thesis as conforming
to the required standard

Dr. M . R . Fellows, Supervisor (Department of Computer Science)

Dr. J. A. Ellis, Departmental Member (Department of Computer Science)

Dr. W. J . Myrvold, Departmental Member (Department of Computer Science)

Dr. R. Odeh, Outside Member (Department of Mathematics)

Dr. R. C. Read, External Examiner (Department of Combinatorics and Optimization, University of Waterloo)

Supervisor: Dr. Michael Fellows

Abstract

The main results of this dissertation are various algorithms related to partially ordered sets. The dissertation basically consists of two parts. The first part treats algorithms that generate ideals of partially ordered sets. The second part concerns the generation of partially ordered sets themselves.

First, we present two algorithms for listing ideals of a forest poset. These algorithms generate ideals in a Gray Code manner; that is, consecutive ideals differ by exactly one element. Both algorithms use storage $O(n)$, where n is the number of elements in the poset. The first algorithm traverses, at each phase, the current ideal being listed and runs in time $O(nN)$, where N is the number of ideals of the poset. The second algorithm mimics the first but eliminates the traversal and runs in time $O(N)$. This algorithm has the property that the amount of computation between successive ideals is $O(1)$.

Secondly, we give orderly algorithms for constructing acyclic digraphs, acyclic transitive digraphs, finite topologies and finite lattices. For the first time we show that the number of finite lattices on 11, 12, and 13 elements are 37622, 262775, and 2018442, respectively, and the number of finite topologies on 8 and 9 elements are 35979 and 363083, respectively.

We also describe orderly algorithms for generating k-colored graphs. We present, in particular, an algorithm for generating connected bicolorable graphs. We also prove some properties of a canonic matrix which might be generally useful for improving the efficiency of orderly algorithms.

Examiners:

Dr. M . R . Fellows, Supervisor (Department of Computer Science)

Dr. J. A. Ellis, Departmental Member (Department of Computer Science)

Dr. W. J . Myrvold , Departmental Member (Department of Computer Science)

Dr. R. Odeh, Outside Member (Department of Mathematics)

Dr. R. C. Read, External Examiner (Department of Combinatorics and Optimization, University of Waterloo)

Acknowledgements

George Polya said that new knowledge is obtained through induction (in the ordinary English sense). This thesis contains new knowledge acquired through observation and induction.

I would like to express my thanks to Prof. Mike Fellows for his careful supervision on this dissertation.

I would like to express my special thanks to Prof. Ron Read for introducing me to the problems, and continuous support and encouragement for solving the problems. My research in this thesis was actually started when I was at the University of Waterloo.

I would like to express my thanks to Prof. Teofilo Gonzalez for not providing me any hints except saying that if we try the small cases, we can generalize and figure out the solution. His teaching gave me a sense of the research attitude.

I would like to express my thanks to Prof. Frank Harary for providing me with the opportunity to solve graph problems in his enjoyable courses and to Dr. Heather Silverman for providing me the time to discuss my future patiently and logically.

I owe final thanks to my parents and my brothers for encouragement to continue my research.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The Problems

This dissertation describes various new algorithms for *combinatorial generation.* One of the fundamental combinatorial problems, the so-called *enumeration* problem, asks how many combinatorial configurations there are for a given set of parameters. *Enumeration* simply means to count the combinatorial configurations for a given set of parameters, whereas *generation* implies constructing the actual objects as well as counting them. Hence, for each enumeration problem, there is a corresponding generation problem. In this dissertation, we assume that the reader is familiar with basic terms and concepts in graph theory (see Harary [20]) and poset (partially ordered set) theory (see Stanley [52]). There are two kinds of combinatorial enumeration problems. The first type of enumeration problem asks, for example, how many graphs there are for a given number of vertices (see Figure 1.1), or how many different ways there are to arrange n distinct objects.

The second type of enumeration problem asks, for instance, how many spanning trees there are for a given graph (see Figure 1.2), how many maximum matchings there are for a given graphs, or how many distinct independent column vectors there are for a given matrix.

In this thesis, we treat the above two types of enumeration problems. We will further ask how to efficiently generate those combinatorial objects. In 1960 Frank Harary

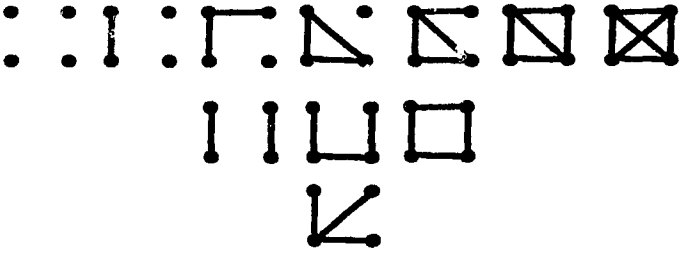

Figure 1.1: Graphs with four vertices

published a list of 27 unsolved enumeration problems [18], which has been subsequently updated and revised [19] [21]. Among the "surviving" unsolved problems are the enumeration of acyclic transitive digraphs (or partially ordered sets), finite lattices, and transitive digraphs (also known as finite topologies). See Figure 1.3 for a graphical representation of all finite lattices with up to 5 elements.

There exists transitivity among the vertices in acyclic transitive digraphs, and the graphical forms of finite lattices and transitive digraphs. As of yet, there are no known closed analytic formulae for the enumeration problem of these digraph families involving transitivity. Thus, in order to count those graphs we have to resort to a constructive method. We use orderly algorithms to construct or to generate these graphs. Orderly algorithms generate a list of the combinatorial configurations of a given parameter value (such as the number of vertices or edges) from lists of the configurations having a smaller parameter value.

We will find later that the representation of $k-$colored graphs is the same as the

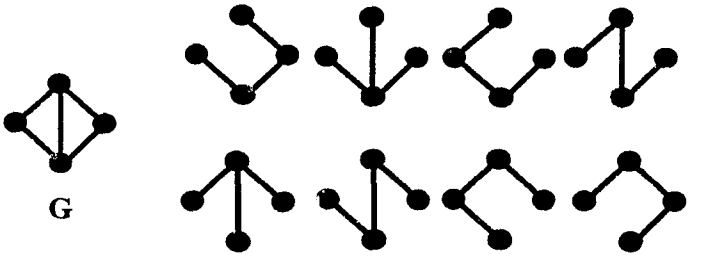

Figure 1.2: A labeled graph G and its spanning trees

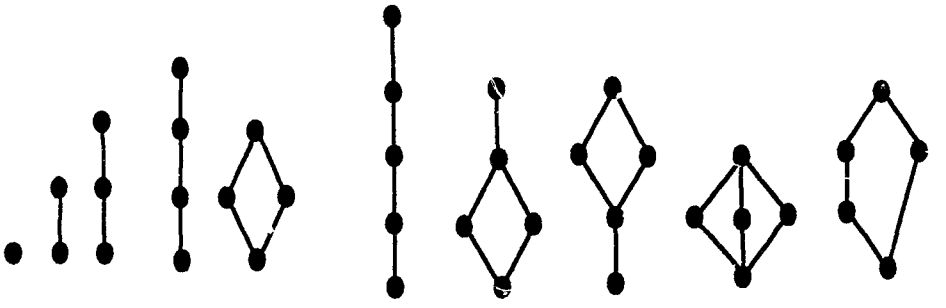

Figure 1.3: Hasse diagrams of finite lattices with up to 5 elements

representation of acyclic transitive graphs; hence, we can also develop orderly algorithms to construct bicolored graphs, and more generally k-colored graphs.

If we represent n distinct objects by n binary bits, that is, 0 stands for the absence of an object and 1 stands for the presence of an object, all the subsets of n distinct objects are easily represented by all distinct 2^n $n-$bit strings. One of the most efficient ways to generate these $n-$bit strings is to generate them as a Gray code. See the example of a Gray code in Figure 1.4.

```
0 0     0 0 0
0 1     0 0 1
1 1     0 1 1
1 0     0 1 0
        1 1 0
        1 1 1
        1 0 1
        1 0 0
```

Figure 1.4: Binary reflected codes of lengths 2 and 3

This Gray code has the property that adjacent strings in a list have exactly one bit different. A Gray code can be viewed as a linearly ordered set of all combinations chosen from a set of n unrelated elements (*antichain*) such that two successive combinations differ

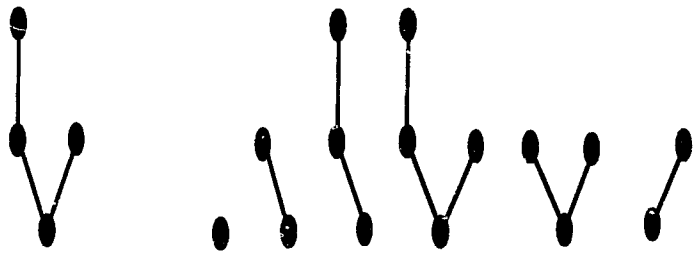

Figure 1.5: A tree poset and its lower ideals

by one element.

Now, we consider the more general case in which the set of elements from which a combination (*a code*) to be chosen is structured. In our case, the elements are chosen from a poset $\mathcal{P}$ whose Hasse diagram is a forest. The combinations that we generate are lower ideals of $\mathcal{P}$.

A lower ideal I of a poset $\mathcal{P}$ is a subset of $\mathcal{P}$ satisfying the conditions that if $x \in I$ and $y \leq x$, then $y \in I$. The minimal element x of a poset $\mathcal{P}$ is an element of $\mathcal{P}$ such that there is no element y, $y < x$. If the Hasse diagram of $\mathcal{P}$ is a forest $\mathcal{F}$, the minimal elements are called the *roots* and a set of lower ideals of $\mathcal{P}$ has a one-to-one correspondence with the set of subtrees which contain the roots of $\mathcal{P}$. In this dissertation, an ideal means a lower ideal. A *Gray code of ideals* is a linearly ordered set of ideals such that successive ideals differ by a single poset element. In Figure 1.5 we show a poset and all its lower ideals. The empty set is always an ideal by definition.

We will describe new efficient algorithms for generating the lower ideals of a forest poset. In other words, a generalized Gray code is a list of any combinatorial objects with the same parameter in which two consecutive objects differ by small amount of changes of the elements which constitute an object. Thus, we may consider a list of lower ideals as a generalized Gray code. Such a generalized Gray code has recently been an actively studied topic in the area of combinatorial algorithms. An excellent survey of recent developments concerning various generalized Gray codes is given in Wilf [56].

1.2 A Gray Code

An n-bit *Gray code* is a non-repeating sequence of the 2^n distinct $n-$bit strings (codewords) over $\{0, 1\}$ such that successive codewords in the sequence differ by the complementation of a single bit. The French engineer Baudot invented this code as a solution to the problem of minimizing error in reading a binary telegraph transmission message. A transmitter keyboard employing a five digit code was exhibited at the Universal Exposition in Paris in 1878. For an account of the work of Baudot, see Heath [26]. Frank Gray [16], apparently without knowing the work of Baudot, first published in 1953 a description of the code for use in preventing errors in certain signal transmissions [16].

Gray codes were, in fact, known in the nineteenth century in the form of a solution to a number of puzzles such as "The Chinese Ring" and "The Tower of Hanoi" (See the paper by Martin Gardner [13]).

If you consider the *$n-$cube* whose vertices represent all the binary strings of length n and whose edges connect a pair of vertices if their binary strings differ by one bit, then the Gray code gives a *Hamiltonian path* or *cycle* in the $n-$cube. In fact, any Hamiltonian path in the $n-$cube graph gives a Gray code. Counting the Hamiltonian cycles and paths in the $n-$cube is still an open problem. The first work on this problem was by Gilbert [14]. The most recent results on the problem are given in Douglas [10].

Gray codes have numerous important applications in engineering and science. For a survey see Goddyn et al. [15]. There are many interpretations of a Gray code. Here, we describe a Gray code as a list generated recursively. From Figure 1.4, we can easily deduce that a Gray code of size n can be constructed from one of size $n-1$. Let L_n denote the list of all $n-$bit strings, ordered as a binary reflected Gray code. The recursive procedure to construct such a *Binary Reflected Gray code* is summarized in Figure 1.6.

Note that in the algorithm shown in Figure 1.6, a list of smaller size is read twice, the second time in reverse order. In a later chapter, we introduce the algorithm by Ehrlich [11] which generates the code without using a list. This algorithm has been improved by Bitner, Ehrlich and Reingold [4] so that the time to determine the next codeword is bounded by some constant.

1. The list L_0 is empty.
2. For each $n = 1, 2, \ldots$, having obtained L_{n-1}, then
   2.1 write out list L_{n-1} and prefix each string with a bit "0";
   2.2 write out list L_{n-1} in reverse order, and prefix each string with a bit "1";
3. The list L_n is the result of concatenating the two lists that were formed in step 2.

Figure 1.6: Gray code generation

Ehrlich [11] views algorithms for generating permutations, combinations, partitions of n, and Gray codes as applications of the generalized Johnson-Trotter algorithm. The Johnson-Trotter algorithm generates all permutations in which a permutation differs from its predecessor by the interchange of two adjacent elements [27] [55]. From an algorithmic point of view, we will see that our algorithms for generating the ideals of a forest poset are of the same basic kind.

1.3 Orderly Algorithms

Orderly algorithms were introduced by Read in [41] and [42] in efforts to improve naive graph generation techniques such as those used, for example, in Heap [25]. A typical classical algorithm for graph generation may be illustrated as follows. Consider the problem of making a list of graphs with $q+1$ edges from a list of those with q edges. See Figure 1.7. In the classical algorithm, we note that in step 2, isomorphic graphs are usually generated many times, and in step 3, we have to search for G in the entire list L_{q+1}, conducting an isomorphism test of G against all graphs already in the list L_{q+1}.

Now we will describe a typical orderly algorithm. Refer to Figure 1.9. First, we

1. Start with a list L_q of all graphs with q edges.
2. Pick each graph of L_q in turn and from it generate candidates for the list L_{q+1} by the addition of a new edge in all possible ways.
3. As each candidate graph G for L_{q+1} is produced determine whether L_{q+1} contains some graph isomorphic to G. If it is the case, reject G and continue processing the next candidate; if not, add G to L_{q+1}.

Figure 1.7: Graph generation by a classical method

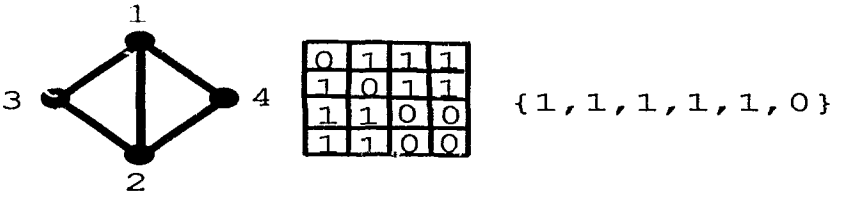

Figure 1.8: The code of a graph

consider the "*code*" of a graph, the *canonical graph*. The canonical graph represents the isomorphism class to which it belongs. A typical code for graphs is that defined from the adjacency matrix as in Figure 1.8. The upper triangular elements of an adjacency matrix are read off by columns to give a binary string. This string changes according to how the vertices of the graph are numbered. Among all the possible ways to number the vertices, we choose the one which makes the string maximal (e.g., lexically). The maximal string is called the *code* of the graph. If a graph is encoded in this way, the task in step 3 of the classical generation method reduces to checking if the code is already in L_{q+1}.

Further we define an order relation (the "*listorder*" denoted by " $\prec$") over the elements of L_q. If $A \prec B$, we say that "A comes before B" or "A precedes B" or "B comes after A".

1. Start with L_q of canonical graphs arranged in list order " $\prec$ ".
   The list L_{q+1} is initially empty.
2. Take each graph of L_q in order and apply on it with
   the augmenting operation to get a sequence of candidates for L_{q+1}
3. As each candidate G is produced, test if the candidate graph G is
   canonical, and if it comes after the last graph at present in L_{q+1}
   If it is so, add it to L_{q+1}; otherwise ignore it.

Figure 1.9: Graph generation by an orderly algorithm

A typical example of such an ordering is the lexical ordering. If each graph is represented by its code defined as in the above example, the lexical order of the codes of L_q is easily understood. Note that as opposed to the classical algorithms, orderly algorithms do not require a search of the list L_{q+1} in order to determine whether or not a graph has already been generated.

There are three things with which we have to be concerned when we design orderly algorithms. First, we must define the code for each isomorphism class over the set of (labeled) graphs with the parameter q. Secondly, we must define the nature of the augmenting operations. Thirdly, we must specify the three orderings: (1) the order in which the codes of the graphs appear in the lists L_q and L_{q+1}; (2) the order in which the augmenting process creates candidates for L_{q+1}; (3) the order used in the definition of the code (the order of isomorphs).

Now, suppose we have an orderly algorithm creating L_{p+1} from L_p. Consider a mapping f from a canonical graph G in L_{p+1} to a canonical graph G' in L_p such that G' is the first graph in L_p which produced G. Such a mapping must satisfy the following three conditions: (1) any canonical graph in L_{p+1} can be produced by the augmenting operations from at least one canonical graph in L_p; (2) the mapping f is *weakly monotonic*; that is, if $X, Y \in L_{q+1}$ and $X \prec Y$, then $f(X) \prec f(Y)$; (3) the order of augmenting a canonical

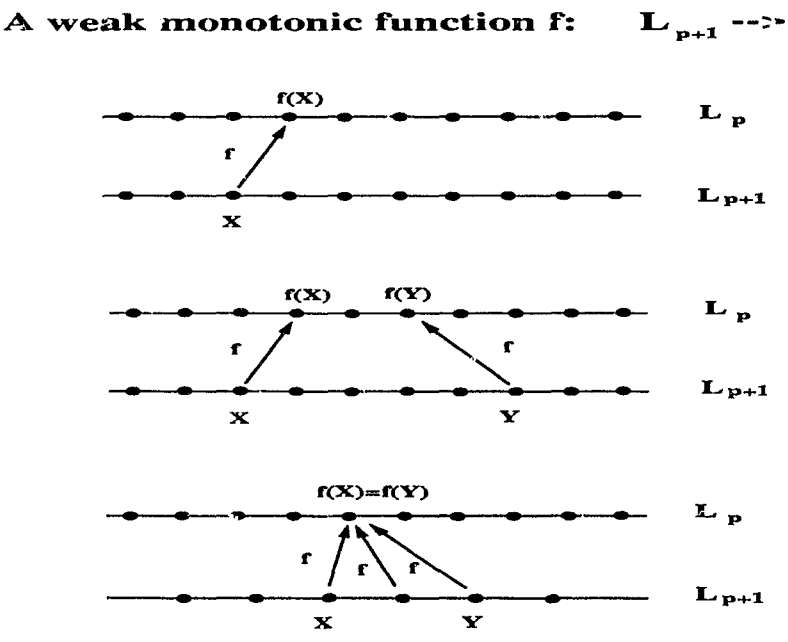

Figure 1.10: Necessary and sufficient conditions for the existence of orderly algorithms

graph in L_p conforms to the order of graphs in the list (Strictly speaking, graphs that are not added to L_{p+1} can be generated in any order). These conditions are illustrated in Figure 1.10. Read [41] proved that these are not only sufficient but also necessary conditions for the existence of orderly algorithms. Thus, designing an orderly algorithm is equivalent to finding such a weak monotonic function.

Let us clarify the complexity of orderly algorithms. Let L_p denote the list of all non-isomorphic canonical graphs with a parameter p. Let $I(p)$ be the time required to do an isomorphism test for two graphs with a parameter p and $A(p)$ the time required to do a canonicity test for a graph with a parameter p. A canonicity test basically involves canonically labeling the vertices of a graph and determining whether or not the original labeling of the given graph is same as this canonical labeling. Finding a canonical labeling usually involves comparing the labelings for all the automorphisms of a given graph. Let L_p^a be the list of graphs augmented from each canonical graph of L_p. (Hence, an augmented graph in L_p^a is either canonical or non-canonical; therefore, precisely speaking L_p^a is not

a list but a set. However, our main concern is the cardinality of a set. Thus, we use the term *list* instead of *set* for convenience.) We denote the cardinality of a set X by $|X|$. In a classical generation scheme, each augmented element of L_p is tested against all the graphs in the current L_{p+1} being created. Therefore, the total time to complete the list L_{p+1} is at most $|L_p^a||L_{p+1}|I(p+1)$. On the other hand, the total time to create the list by an orderly algorithm is at most $|L_p^a|A(p+1)$. Noting that $|L_p|$ is typically exponential or super-exponential, we realize that the orderly algorithm shows a significant improvement over the classical one. Note that at the moment of writing this dissertation, it is not known whether or not isomorphism test can be done in polynomial time in the size of a graph. However, we can replace an isomorphism test by a procedure to find a canonic form of a graph. It is not necessary to find a canonic form of a graph in order to determine whether a representation of a graph is canonic, however, a procedure to find a canonic form of a graph provides one means by which a canonicity test can be performed.

Our orderly algorithms described initially in this dissertation are slightly different from Read's original orderly algorithm with respect to the order within a list. In Read's algorithm, all the graphs were ordered in some order within a list. Suppose that we do not concern ourselves with the order of graphs in a list but all the graphs in the list have the same parameter, say, p. So, precisely speaking, the term "list" is not correct, but for convenience we use "list" instead of "set". Can we generate a list L_p so that the order of the graphs in the list does not matter? This is feasible since a canonicity test depends solely on the graphs, but not on the order of graphs in the list L_p. (If one needs an ordered list, we may sort the list later.) Such an orderly algorithm without the condition of linear order within a list is called a *weak-orderly algorithm.* In the final chapter, we will describe a technique that can be used to transform weak-orderly algorithms to orderly algorithms with the property that all graphs in each generated list are ordered.

Another difference between our orderly algorithms and Read's original orderly algorithm is the method of augmentation. A graph can be classified by two natural parameters, namely, the number of vertices p and the number of edges q. The original orderly algorithm by Read [41] uses q as the parameter of the list while p is fixed. In our orderly

algorithms in this dissertation, we use the number of vertices p as the parameter of the list and q varies as well. The number of edges varies since adding one vertex can induce a different number of new edges (0 to p) incident with the augmented vertex.

Finally, we briefly describe that the relationship among orderly algorithms, the graph coding problem, and the isomorphism problem. Graph generation actually constructs graphs as well as counts them. Thus, graph generation requires efficient data representation to avoid a membership test, i.e., a test to determine whether or not a given graph has the desired properties such as acyclicity, planarity and colorability (see Colburn and Read[6]). Finding a good code for a specific graph family involves finding an efficient data representation of the graph. Thus, orderly algorithms are also closely related to the graph coding problem (see Read and Colburn [45]). The graph coding problem is, practically speaking, the problem of canonically labeling the vertices of a given graph by integers so that two graphs are isomorphic if and only if they have the same labeling (the code). Hence, if the graph coding problem is solved efficiently, we can also solve the isomorphism problem efficiently. Applications of graph coding can be found in the field of chemistry (see Read [43], [44]).

1.4 Previous Work and Applications

Generating the ideals of a poset has several applications in Operations Research and Optimization. Steiner [53] treats the problem of generating ideals as an integral part of dynamic programming algorithms for precedence-constrained scheduling problems, assembly line balancing, project scheduling, and reachability in reliability networks. The problem of generating the ideals of a tree poset lexically was first considered by Ruskey [49]. The algorithm for ideal generation of [49] runs in linear average time. The algorithm of Steiner [53] for generating all ideals of an arbitrary poset runs in linear average time.

Algorithms for generating graphs are useful in exploring certain conjectures in graph theory and in obtaining census information for graphical enumeration problems. Listing some families of chemical compounds can be done by generating graphs representing the

chemical structures. For example, an *alternant molecule* is essentially represented by a bicolored graph (see Balaban [1]). Chemists are often interested in having some computerized system for cataloging chemicals so that a given chemical can be easily identified, making chemical patent searching, for example, less tedious. The relationship between Chemistry and Graph Theory is exposited at length in Balaban [1]. Bicolored graphs without isolated vertices are also used to represent simplical complexes which in turn represent some architecture design problems. One architectural application is concerned with the minimal representation of patterns based on a rectangular grid. Since a grid is a bicolored graph, these patterns can be also represented by bicolored graphs. See Harary et al. [22] for an application of enumeration techniques to architectural design. Determining the value of *permanent* of a matrix can be done by an algorithm for counting all maximum matchings in a bipartite graph corresponding to the matrix (see Nijenhuis and Wilf [38]). A technique for generating the maximum matchings in a general graph is used for analyzing some biomedical images (see Tanimoto [54]). For developments in the field of graph generation, see Read [42]. The problem of graph generation culminated in 1981 with the completion of the catalog of all graphs with 10 vertices. For an account of this cataloging, see Cameron et al. [5].

The enumeration by means of closed analytic formulas of acyclic transitive digraphs (partially ordered sets), finite topologies (transitive digraphs), and finite lattices are unsolved problems (see Harary and Palmer [23] and also Birkhoff [3]). Thus, in order to count these graphs with a given number of vertices p, we must currently rely on generative methods. For acyclic transitive digraphs and finite topologies, Evans, Harary, and Lynn [12], using a computer, counted the numbers of labeled cases on up to 7 vertices. For the unlabeled case, graphs on up to 7 vertices have been generated and recorded in a book by Sloane [51]. Culberson and Rawlins [7] recently generated the acyclic transitive digraphs up to 11 vertices. Lastly, Kyuno [33] and Kyuno and Ito [34] generated the finite lattices up to 10 vertices.

In the above described research, Evans et al. and Kyuno and Ito used an expensive isomorphism test to eliminate duplicates. By using an orderly algorithm, we can avoid

the repeated use of an isomorphism test (see Read [41]).

A $k-$colored graph is represented by exactly the same canonical form as can be used for acyclic digraphs. Thus, we can obtain an orderly algorithm for generating $k-$colored graphs from an orderly algorithm for generating acyclic digraphs. For labeled k-colored graph enumeration, see Read[40] and for bicolored and bicolorable graph enumeration, see Harary[17], [20].

1.5 Overview

The plan of this dissertation is as follows. In Chapter 2, various definitions and notation which are used in later chapters are defined. In Chapter 3, the algorithm for generating a binary reflected code by Bitner Ehrlich and Reingold is introduced. Then, the existence of a Gray code of lower ideals of a forest poset is given. Finally, the algorithm for generating such lower ideals in a Gray code manner is given and some methods of lower ideals generated by the algorithm are also given. In Chapter 4, the previous algorithm is improved to an algorithm with theoretically optimal time complexity. In Chapter 5, we describe orderly algorithms for generating acyclic digraphs, acyclic transitive digraphs (posets), finite lattices, and finite topologies. In Chapter 6, we give orderly algorithms for generating $k-$colored graphs. We also present some properties by which we may improve the efficiency of orderly algorithms in general. In the last chapter, we summarize our results and describe open problems related to the topics treated in this dissertation.

Chapter 2

Mathematical Preliminaries

Graph-theoretical terms used in this dissertation may be found in Harary [20], except that in this dissertation, *points* are called *vertices* and *lines* are called *edges* and similarly, poset-theoretical terms may be found in Stanley [52]. For the convenience of the reader, however, some of the central terms are recalled in this chapter. We consider a set of elements P and a binary relation $\leq$ defined on P. In this dissertation, the number of elements in P is always finite. By an abuse of notation, we denote this as $P = (P, \leq)$. We say that x and y are *comparable* if either $x \leq y$ or $y \leq x$; otherwise, they are *incomparable*. We write $x < y$ if $x \leq y$ but $x \neq y$. This P can be represented as a graph as follows. Let each element x in P be represented by a vertex. Let two vertices x and y be connected by an arc from y to x if the corresponding two elements x and y are such that $x \leq y$. Then, the graph constructed in such a way is called a *digraph*, denoted also by P, for convenience.

A digraph P is called a *transitive digraph* if P satisfies the condition: for x and y in P, $x \leq y$ and $y \leq z$ imply $x \leq z$ (transitivity). Refer to Figure 2.1 for an example of a transitive digraph.

A sequence of distinct elements in P, $C = x_1 < x_2 < \ldots < x_n$ is called a *chain* or *directed path* in the corresponding graph if any consecutive two elements of C are comparable with respect to the binary relation in P. The *length* of the chain C is denoted by $l(C) = \mid C \mid -1$. If x_1 and x_n are equal in the chain, the chain is called a *directed cycle*.

A digraph P which has no directed cycles is called an *acyclic digraph.*

A *source* is a vertex with no incoming arcs. If x is a vertex of a digraph P having no directed cycles, we define the *height* $h(x)$ as follows. $h(x) = sup\{l(C) : C\ chain \in \mathcal{C}(x)\}$, where $\mathcal{C}(x) = \{y \in P : \exists\ chain\ from\ y\ to\ x\}$. In particular, if x is a source, $h(x) = 0$. Similarly, $h(P) = sup\{h(x) : x \in P\}$ is the *height* of P. If the underlying digraph is clear, we simply write h (without a parameter) instead of $h(P)$. A *layer* is a set of elements in P which have the same height. See Figure 2.2.

A digraph P is called a *partially ordered set (poset)* if the binary relation satisfies the following two additional conditions as well as transitivity: (1) $x \leq x$ for all $x \in P$ (reflexivity), and (2) $x \leq y$ and $y \leq x$ imply $x = y$ (antisymmetry). A graph representing a partially ordered set is called an *acyclic transitive digraph.* An element x of a poset P is *maximal (minimal)* if $x \leq y$ ($x \geq y$) implies $x = y$ for all $y \in P$; x is *greatest (least)* if $y \leq x$ ($y \geq x$) for all $y \in P$.

We generally denote a poset by $\mathcal{P}$, and its underlying graph by P. We denote the set of elements in $\mathcal{P}$ by $S(\mathcal{P})$ and relation that defines $\mathcal{P}$ by $R(\mathcal{P}) \subseteq S(\mathcal{P}) \times S(\mathcal{P})$. If $x, y \in \mathcal{P}$, then we say y *covers* x if $x < y$ and no element $z \in P$ satisfies $x < z < y$. We also say that x is a *child* of y and y is a *parent* of x if $x < y$. An *ideal* I of a poset $\mathcal{P}$ is a subset of the elements of $\mathcal{P}$ such that $x \in I$ and $y \leq x$ implies that $y \in I$.

An *antichain*, A, is a subset of the elements of $\mathcal{P}$ in which every two elements are

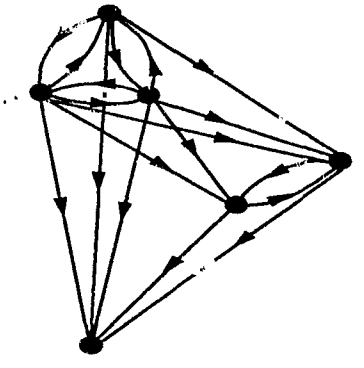

Figure 2.1: An example of a transitive digraph

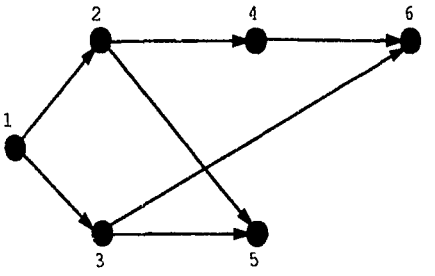

Figure 2.2: An example of acyclic digraph whose vertices are ordered by distance from sources

incomparable; that is, if $x, y \in A$ with $x \neq y$, then neither $x < y$ nor $y > x$ holds. Observe that the set of maximal elements M in an ideal I forms an antichain and M fixes I in the sense that $I = \{x | \exists m \in M, x \leq m\}$. Thus, there is a one-to-one correspondence between the ideals and the antichains.

The poset that is an antichain of n elements is denoted $\mathcal{A}_n$. For two sets $\mathcal{A}$ and $\mathcal{B}$, the Hamming distance between the two sets $\mathcal{A}$ and $\mathcal{B}$ is the number of elements in which they differ, that is, $H(\mathcal{A}, \mathcal{B}) = |(\mathcal{A} - \mathcal{B}) \cup (\mathcal{B} - \mathcal{A})|$.

A poset can be represented by a *Hasse diagram* in which distinct elements are represented by distinct points, in which x is placed above y whenever $x > y$, and x and y are joined by an arc from x to y whenever x covers y. Any finite poset is determined up to isomorphism by its diagram.

If the Hasse diagram of a poset $\mathcal{P}$ is a tree rooted at a unique minimal element, we simply say that the poset is called a *tree poset* and the Hasse diagram is termed a tree. If the Hasse diagram of a poset is a disjoint union of trees, we term the poset a *forest poset*. We may treat this tree poset as an *ordered tree* with the minimal element being the root. In general, there are many ordered trees corresponding to a given poset. We will pick up one of them as a representative of the poset. If a poset is a tree poset, a tree is an *oriented ordered tree*, where the direction of arcs are oriented towards the root, the minimal element. By ordered tree we mean its subtrees are ordered. An *ordered forest* is similarly defined. Nodes in an ordered forest are numbered in preorder.

Children of a common parent are called *siblings*. *Rooted-subtrees* are subtrees whose minimal element is the root of a tree poset. Then, we can easily see that there exists a one-to-one correspondence between the set of lower ideals of a tree poset $\mathcal{T}$ and the set of *rooted-subtrees*. By the definition of an ideal, the empty subtree is also an ideal of a tree. Each ideal of a tree is represented by a codeword of a bit-string where "0" means the corresponding element is not in the ideal and "1" means the corresponding element is in the ideal. In this paper, subtree means rooted-subtree unless otherwise stated.

A *forest poset* is one whose Hasse diagram is an ordered forest. The roots of the trees of the forest are the minimal elements of the poset. By $x \prec y$ or $y \succ x$ we mean that the element x is the predecessor of y or y is the successor of x in the *preordering* of the elements in a forest poset $\mathcal{F}$.

In Ruskey [49] the ideals of a tree poset are called *r-subtrees*. The set of ideals of a poset $\mathcal{P}$, ordered by set inclusion, form a distributive lattice $J(\mathcal{P})$. It is well known that for every finite distributive lattice L, there is a unique poset $\mathcal{P}$ for which $J(\mathcal{P})$ is isomorphic to L. For more information and the definition on a distributed lattice, see Stanley [52]. In a slight abuse of notation, we also use $J(\mathcal{P})$ to denote the Hasse diagram of the lattice of ideals of $\mathcal{P}$. In this context we refer to $J(\mathcal{P})$ as the *ideal graph* of $\mathcal{P}$. We are interested in finding Hamilton cycles and paths in the underlying undirected graph of Hasse diagrams of distributive lattices (such a cycle or path yields a Gray code of the ideals, the object of our algorithms). For more information on posets and ideals, see Stanley [52].

Note that $J(\mathcal{P})$ is connected and bipartite. It is connected since every distributive lattice has a maximum and a minimum element. There is a bipartition of the vertices of $J(\mathcal{P})$ into two sets, $Even(\mathcal{P})$ and $Odd(\mathcal{P})$, depending on whether the ideal represented by a vertex has an even or odd number of elements. The *parity difference* $d(\mathcal{P})$ of a poset $\mathcal{P}$ is defined to be the difference $|Even(\mathcal{P})| - |Odd(\mathcal{P})|$.

There are two standard ways of combining posets to get a new poset. The *direct sum* of two disjoint posets $\mathcal{P}$ and $\mathcal{Q}$ is the poset $\mathcal{P} + \mathcal{Q}$ on the union $S(\mathcal{P}) \cup S(\mathcal{Q})$ such that $x \leq y$ in $\mathcal{P} + \mathcal{Q}$ if and only if either $x, y \in \mathcal{P}$ and $x \leq y$ in $\mathcal{P}$, or $x, y \in \mathcal{Q}$ and $x \leq y$ in $\mathcal{Q}$. The *ordinal sum* of two disjoint posets $\mathcal{P}$ and $\mathcal{Q}$ is the poset $\mathcal{P} \oplus \mathcal{Q}$ on the union

$S(\mathcal{P}) \cup S(\mathcal{Q})$ such that $x \leq y$ in $\mathcal{P} \oplus \mathcal{Q}$ if $x, y \in \mathcal{P}$ and $x \leq y$ in $\mathcal{P}$, or $x, y \in \mathcal{Q}$ and $x \leq y$ in $\mathcal{Q}$, or if $x \in \mathcal{P}$ and $y \in \mathcal{Q}$. Note that a tree poset can be built up by applying the operations of direct sum of two forest posets and the ordinal sum of a forest poset with a single element.

A graph $G = (V(G), E(G))$ consists of a set of *vertices* $V(G)$ and a subset of unordered pairs of vertices called *edges* $E(G) \subseteq V(G) \times V(g)$. The *product* of the graphs G and H is the graph $G \times H$ with vertex set $V(G) \times V(H)$, in which $[v, w]$ is adjacent to $[v', w']$ if and only if $v = v'$ and $[w, w'] \in E(H)$ or $w = w'$ and $[v, v'] \in E(G)$. We can easily see that

$$J(\mathcal{P}_1 + \mathcal{P}_2) \cong J(\mathcal{P}_1) \times J(\mathcal{P}_2) \tag{2.1}$$

where $\cong$ denotes graph isomorphism.

We are interested in efficient algorithms for generating combinatorial objects. Suppose that the objects are represented by sequences of n elements and that the total number of objects is N. An algorithm for generating those objects is said to run in *constant average time* if the total amount of computation (excluding output) is $O(N)$. If the total amount of computation is $O(nN)$, then the algorithm is said to run in *linear average time.*

A constant average time algorithm is said to be *loopless* (or constant worst-case time) if (a) the amount of computation used in going from one object to the next is $O(1)$; (b) the amount of computation needed to produce the first object is $O(n)$; and (c) the amount of computation needed to decide whether an object is the last one is $O(1)$. Up to a constant factor, no algorithm can be faster than a loopless algorithm. Similarly, a linear average time algorithm is called *linear worst-case time* if (a) and (c) above can be done in $O(n)$.

A *lattice* $L = (L, \vee, \wedge)$ is a poset L such that for every two elements x, y in L, the poset $\{z \in L : z \geq x, z \geq y\}$ has a least element $x \vee y \in L$, the *join* of x and y, and the poset $\{z \in L : z \leq x, z \leq y\}$ has a greatest element $x \wedge y \in L$, the *meet* of x and y. We call this lattice property *the unique bound condition.* If only either a greatest or a least element is defined for any two elements x and y, the lattice is called a *semilattice.* For simplicity, the Hasse diagram of a lattice is also called a lattice.

From the definition of a lattice, we will note the following property.

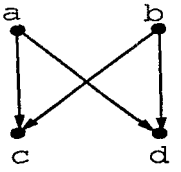

Figure 2.3: L_4: Four elements not satisfying *the unique bound condition*

PROPOSITION **2.1** *If a poset P is a lattice, then P does not contain the poset L_4 of figure 2.3 as a subposet.*

□

If there is such a set of four elements in P, we say that the four elements violate *the unique bound condition.*

A digraph is *strongly connected*, or *strong*, if any two vertices are mutually reachable by a directed path. A *strong component* of a digraph G is a maximal strong subgraph. Let $S_1, S_2, \ldots, S_m$ be the strong components of G. The *condensation* G^* of a digraph G is a digraph that has the strong components of G as its vertices, with an arc from S_i to S_j whenever there is at least one arc in G from a vertex of S_i to a vertex in S_j. The condensation G^* of any digraph G has no cycles, and hence is an acyclic digraph. Two vertices u and v of the digraph G are *similar* if there is some automorphism α of G, for which $\alpha(u) = v$. Two edges $x_1 = (u_1, v_1)$ and $x_2 = (u_2, v_2)$ are *similar* if there is an automorphism α of G such that $\alpha((u_1, v_1)) = (u_2, v_2)$. A digraph is *symmetric* if every pair of vertices is similar and every pair of edges is similar.

We state two lemmas without proofs. For the proofs, see Harary [12](page 200).

LEMMA **2.1** *Every strong component of a transitive digraph is complete and symmetric.*

LEMMA **2.2** *The condensation of a transitive digraph with m strong components is itself an acyclic transitive digraph on m vertices.*

For an object ω of a linearly ordered combinatorial family, the *rank* of ω is the integer

r such that ω is the rth member of the family. The *lexical order* of two vectors v and w is defined as follows. If $v = (v_1, v_2, \ldots, v_n)$ and $w = (w_1, w_2, \ldots, w_m)$, then v is lexically larger than w (denoted by either $v \succ w$ or $w \prec v$) if and only if there is some $k < m, n$ such that $v_i = w_i$ for all $i < k$, and $v_k > w_k$ for some $k \leq m, n$, or $n > m$ and for $1 \leq i \leq m$, $v_i = w_i$. If v could be equal to w, we write $v \succeq w$ or $w \preceq v$.

For a given vector of size n, $v = (e_1, e_2, \ldots, e_n)$, a *prefix* v^i is defined to be the *subvector* composed of the leading i elements of v; that is, $v^i = (e_1, e_2, \ldots, e_i)$, $1 \leq i \leq n$.

A *permutation matrix* S is a square matrix which has precisely one "1" in each row and precisely one "1" in each column. The permutation matrix S is obtained from the identity matrix by interchanging the kth row with the rth row such that $k \leq r$ and $c_{rk} \neq 0$. Let M be a square matrix. Then, $M' = S \times M$, where "$\times$" is the usual matrix multiplication, is the square matrix obtained by interchanging the kth row and the rth row in the matrix M and $M'' = M \times S$ is the square matrix obtained by interchanging the kth column and the rth column in the matrix M. If M is an adjacency matrix for a graph G, then $M' = S \times M \times S$ is the adjacency matrix for the graph G obtained by interchanging the labelings of the vertices k and r. The multiplication of two permutation matrices $S' = S_1 \times S_2$ is obviously also some permutation matrix.

A k-colored graph is a graph $G = (X_1, X_2, ..., X_k, V, E)$ where X_i, $i = 1, \ldots, k$ are mutually disjoint subsets of the vertices V, with $\cup_{i=1}^{k} X_i = V$ and E is the set of edges, which is a subset of unordered pairs of vertices (x_i, x_j), $x_i \in X_i$ and $x_j \in X_j$, $1 \leq i < j \leq k$. We consider in this paper only the k-colored graphs for which each of the X_i, $i = 1, \ldots, k$ is not empty, i.e., $|X_i| = m_i > 0$.

Chapter 3

Forest Poset ideals

3.1 Introduction

Among all the Gray codes the one most widely know is called the Binary Reflected Gray Code (BRGC). Algorithm E due to Ehrlich [11] for generating the BRGC is described in Figure 3.1. This algorithm runs in $O(n)$ time from one codeword to the next. The algorithm can be viewed as analogous to the well known Johnson-Trotter algorithm [27][55] for generating permutations. In the Johnson-Trotter algorithm, each element has an associated priority, and a status. If the status of an element is *movable*, it is moved(swapped with the adjacent element), otherwise, the element of the next highest priority is moved. Once an element of some priority is moved, all the elements with higher priorities become movable. The process continues until all the elements are not movable.

In the case of our algorithms, the priority of the elements of a forest poset is defined according to the preorder of the elements in the forest poset (the root element has the highest priority). All the elements are considered according to this priority as in the Johnson-Trotter algorithm. In addition, all the elements have predecessor or successor relationships in the poset. Unlike the Johnson-Trotter algorithm, only the elements in the current ideal or having a predecessor in the current ideal are considered. Moreover, no elements having a successor in the ideal can be an candidate to be removed from the current ideal.

Algorithm E has been improved by Bitner, Ehrlich and Reingold to the one which runs in $O(1)$ time from one codeword to the next. We refer to their algorithm as Algorithm BER.

We adopt a different approach to improve Algorithm E. Algorithm B, our version of Algorithm BER(to be explained in Chapter 4), is used as a subprocedure in the algorithm for generating ideals of a forest poset which is the main result of the first part of this dissertation.

3.2 The Ehrlich and BER Algorithms

Now, we introduce Algorithm E by Ehrlich [11]. In connection with our Algorithm P for generating forest ideals, we describe Algorithm E in terms of poset elements. Algorithm E works on n unrelated poset elements (i.e., an antichain) $\mathcal{A}_n$. The algorithm lists all the possible subsets S of $\mathcal{A}_n$. We assume the elements are labeled in increasing order from one to n and arranged accordingly with the smallest on your left. The smaller the label is, the higher the priority.

We consider that each element of $\mathcal{A}_n$ can take either *free* status or *stuck* status.

If the status of the element is *free*, the element may be either removed from the current subset S (the element is *deletable*) or added to S (the element is *augmentable*); otherwise, the status of the element is *stuck* and it can be neither removed from S nor added to S. The set $\mathcal{A}_n$ is *changeable* if there exists an element which is either deletable or augmentable.

Initially, all the elements are *free*. The algorithm runs as long as there exists a free element in S (step (4)). We choose the leftmost *free* element v from S (step (5)). Then, all the elements to the left of v become *free* and v itself becomes *stuck* (steps (6) and (7)). Depending on whether or not v is in S, we add v to or delete v from S (step (8)). After the updated S is output (step (9)), we return to step (4).

```
(1)  forall v ∈ S do status(v) ← free;
(2)  S ← ∅;
(3)  Output(S);
(4)  while S is changeable do begin
(5)      v ← min{v ∈ S | v is changeable};
(6)      forall w < v do status(w) ← free;
(7)      status(v) ← stuck;
(8)      if v ∈ S then S ← S − {v} else S ← S ∪ {v};
(9)      Output(S);
(10) end;
```

Figure 3.1: Algorithm E

3.3 The Existence of a Gray Code of Lower Ideals of a Forest Poset

A proof of the existence of a Gray code of lower ideals of a forest poset is given by induction on the number of elements in the poset. If a Gray code has the property that the first codeword and the last codeword differ by a single element, it is called a *cyclic Gray code.* Necessary and sufficient conditions for a Gray code of lower ideals of a forest poset to be cyclic are also given below.

First, we give two easy lemmas. We show that there is a Hamiltonian path in $J(\mathcal{P})$ if $\mathcal{P}$ is a forest poset. This will follow as a consequence of more general results about the ordinal and direct sums of posets. A necessary condition for the existence of a Hamiltonian path in $J(\mathcal{P})$ is that the parity difference $|d(\mathcal{P})| \leq 1$. Let us first determine $d(\mathcal{P})$ if $\mathcal{P}$ is a forest poset.

Let $i(\mathcal{P})$ denote the number of elements in $J(\mathcal{P})$ (the number of ideals of $\mathcal{P}$). It is easy to show that $i(\mathcal{P}+\mathcal{Q}) = i(\mathcal{P}) \times i(\mathcal{Q})$ and that $i(\mathcal{P} \oplus \mathcal{Q}) = -1 + i(\mathcal{P}) + i(\mathcal{Q})$. Similarly, we have

$$d(\mathcal{P}+\mathcal{Q}) = d(\mathcal{P})d(\mathcal{Q}) \tag{3.1}$$

$$d(\mathcal{P}\oplus\mathcal{Q}) = \begin{cases} -1+d(\mathcal{P})+d(\mathcal{Q}) & \text{if } |\mathcal{P}| \text{ is even} \\ +1+d(\mathcal{P})-d(\mathcal{Q}) & \text{if } |\mathcal{P}| \text{ is odd} \end{cases} \tag{3.2}$$

For forests we apply (3.2) with $\mathcal{P} = \mathcal{A}_1$. Since $d(\mathcal{A}_1) = 0$, the parity difference for forests is 0 or 1. However, that $d(\mathcal{P}) = 0$ is not a sufficient condition for $J(\mathcal{P})$ to have a Hamiltonian path. Such an example is illustrated in Figure 3.2.

LEMMA **3.1** *Let $\mathcal{P}$ and $\mathcal{Q}$ be non-empty posets both of whose ideal graphs have Hamiltonian paths starting at $\emptyset$. Then $J(\mathcal{P}+\mathcal{Q})$ has a Hamiltonian path starting at $\emptyset$. Furthermore, if $|J(\mathcal{P})|$ (or $|J(\mathcal{Q})|$) is even then $J(\mathcal{P}+\mathcal{Q})$ has a Hamiltonian cycle.*

PROOF: Let H be a Hamiltonian path in $J(\mathcal{P})$ and H' be a Hamiltonian path in $J(\mathcal{Q})$. Then $H \times H'$ is a spanning subgraph of $J(\mathcal{P}) \times J(\mathcal{Q})$. Hence, by (2.1) it is again isomorphic to a spanning subgraph of $J(\mathcal{P}+\mathcal{Q})$. Since H and H' are paths, $H \times H'$ is an $|H|$ by $|H'|$ grid graph. The ideal $\emptyset$ of $J(\mathcal{P}+\mathcal{Q})$ corresponds to the corner vertex $(\emptyset,\emptyset)$ of $H \times H'$. It is easy to construct a Hamiltonian path in a grid graph, starting at a corner. Furthermore, unless both $|H|$ and $|H'|$ are odd, a Hamiltonian cycle can be found easily in $H \times H'$. □

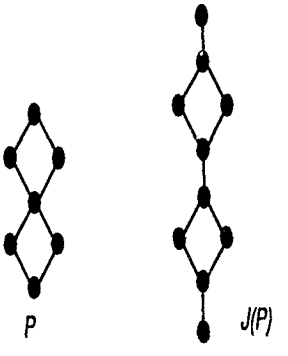

Figure 3.2: A poset $\mathcal{P}$ for which $d(\mathcal{P}) = 0$ but $J(\mathcal{P})$ has no Hamiltonian path.

LEMMA **3.2** *Let $\mathcal{P}$ be a poset for which $J(\mathcal{P})$ has a Hamiltonian path starting at $\emptyset$ and ending at $S(\mathcal{P})$, and let $\mathcal{Q}$ be a poset for which $J(\mathcal{Q})$ has a Hamiltonian path starting at $\emptyset$. Then there is a Hamiltonian path in $J(\mathcal{P} \oplus \mathcal{Q})$ starting at $\emptyset$.*

PROOF: Let $a_1, a_2, \ldots, a_N$ be a Hamiltonian path in $J(\mathcal{P})$ where $a_1 = \emptyset$ and $a_N = S(\mathcal{P})$. Let $b_1, b_2, \ldots, b_M$ be a Hamiltonian path in $J(\mathcal{Q})$ where $b_1 = \emptyset$. Then the path

$$a_1, a_2, \ldots, a_N \cup b_1, a_N \cup b_2, \ldots, a_N \cup b_M$$

is a Hamiltonian path in $J(\mathcal{P} \oplus \mathcal{Q})$ starting at $\emptyset$. □

Note that the condition of Lemma 3.2 that the Hamiltonian path in $J(\mathcal{P})$ ends at $S(\mathcal{P})$ cannot be relaxed since $S(\mathcal{P})$ is a cut vertex of $J(\mathcal{P} \oplus \mathcal{Q})$ (if $\mathcal{P}$ and $\mathcal{Q}$ are both non-empty).

As a consequence of Lemmas 3.1 and 3.2 we have the following theorem. The proof is immediate by induction on the number of nodes in the forest, noting that any forest poset $\mathcal{F}$ can be constructed by repeated applications of either $\mathcal{T}_1 + \mathcal{T}_2$ or $\mathcal{T}_1 \oplus \mathcal{T}_2$ where T_j for $j = 1, 2$ are disjoint subtrees of $\mathcal{F}$ such that $i(T_j) < i(\mathcal{F})$ for $j = 1, 2$.

THEOREM **3.1** *If $\mathcal{P}$ is a forest poset, then there is a Hamiltonian path in $J(\mathcal{P})$.*

We can also characterize those forests posets whose ideal graphs have Hamiltonian cycles.

THEOREM **3.2** *Let $\mathcal{F}$ be a forest poset consisting of trees $T_1, T_2, \cdots, T_k$. There is a Hamiltonian cycle in $J(\mathcal{F})$ if and only if $k \geq 2$ and there is at least one tree T_i for which $|J(T_i)|$ is even.*

PROOF: If $k = 1$, then $\emptyset$ is a pendant vertex in $J(\mathcal{F})$. If all $|J(T_i)|$ are odd then $i(\mathcal{F})$ is odd, and since $J(\mathcal{F})$ is bipartite it cannot have a Hamiltonian cycle. On the other hand, if $k \geq 2$ and there is at least one tree T_i for which $|J(T_i)|$ is even, then by Lemma 3.1 there is a Hamiltonian cycle in $J(\mathcal{F})$. □

3.4 Algorithm P

Algorithm P generates all lower ideals of a forest poset without repetition. It determines successive ideals by traversing the previous ideal in effect in preorder. The transition from one ideal to the next takes $O(n)$ time in the worst case, where n is the number of elements of $\mathcal{P}$. Experimentally, however, the algorithm has been found to exhibit on the average constant transition time from one ideal to the next. We offer the conjecture (see Chapter 7) that Algorithm P has constant average transition time, but the problem appears to be difficult.

Theorem 3.1 is constructive. Note, however, that the algorithm corresponding to the proof of Theorem 3.1 may yield different Hamiltonian paths in $J(\mathcal{P})$ for a given forest poset $\mathcal{P}$ depending on how $\mathcal{P}$ is decomposed into smaller tree posets. In this section we will describe an algorithm that finds such a particular Hamiltonian path systematically. The main idea is to use a decomposition based on a depth-first search. In Chapter 4 we will show how to improve this algorithm so that it is loopless.

In Chapters 3 and 4 we deal with a fixed forest poset $\mathcal{F}$ with n elements. The elements of a forest poset are referred to as *nodes* (We will reserve the term *vertices* for discussion of the ideal graph). The parent of a node v is denoted by $par(v)$. We extend the notions of *status*, and *changeable* defined on an element in an antichain to those defined on an element of a poset. Each node v has a status, $status(v)$, which takes on one of two values, *free* or *stuck*.

DEFINITION: Let F be a rooted subforest of a forest poset $\mathcal{F}$, and v a node for which $status(v) = free$.

- If $v \notin F$, then v is *augmentable* if v is a root or $par(v)$ is in F.
- If $v \in F$ then, v is *deletable* if none of the children of v is in F and the status of each child is *stuck*.

A node that is augmentable or deletable is said to be *changeable*. If there is a changeable node in F, then F is *changeable*; if v is not in F, we say that v is *null* with respect

```
(1)  forall v ∈ F do status(v) ← free;
(2)  F ← ∅;
(3)  Output(F);
(4)  while F is changeable do begin
(5)       v ← min{v ∈ FA | v is changeable};
(6)       forall  w < v, w ∈ FA do  status(w) ← free;
(7)       status(v) ← stuck;
(8)       if  v ∈ F   then  F ← F − {v};
                            forall x ∈ Children(v) do status(x) ← free;
                       else  F ← F ∪ {v};
(9)       Output(F);
(10) end;
```

Figure 3.3: Algorithm P

to F (In the sequel, where F is understood, we may simply say that v is *null*). For a rooted subforest F we define the associated set of nodes: $FA = \{x \in S(\mathcal{P}) \mid x \in F$, or $par(x) \in F\}$.

We assume that the forest has been given a *preorder* labeling (that is, the nodes have been labeled in the order encountered in a depth-first search). We use this labeling when referring to the nodes of $\mathcal{F}$. A preorder labeling is obtained by recursively labeling a root and then its subtrees, from left to right. The labels are $1, 2, \ldots, n$.

The algorithm performs a preorder traversal of the forest, changing stuck nodes to free, until the first changeable node is encountered. Then, the node is added to or deleted from the tree depending on whether or not v is in F. It is clear that the running time of Algorithm P is $O(nN)$, where $N = i(\mathcal{F})$. An example of the list of ideals produced by Algorithm P is illustrated in the appendix.

The proof of the validity of Algorithm P is parallel to the proof of the existence of a Gray code sequence of ideals in a forest. Let F' be the final ideal output by Algorithm P.

Then, by Algorithm P′ we mean Algorithm P with line (2) replaced with $F \leftarrow F'$, where for all the nodes $v \in F'$, $status(v) = free$.

LEMMA **3.3** *Algorithm P produces a Hamiltonian path in $J(\mathcal{F})$ starting at $\emptyset$. If F' is the ending vertex (the last ideal) on this Hamiltonian path, then Algorithm P′ produces the same Hamiltonian path, but in opposite order, starting from F'.*

PROOF: We prove our claim by induction on the number of nodes, k, in the input forest $\mathcal{F}$. For $k = 1$, Algorithm P starts with $\emptyset$ and the node is augmented and output. At that time the $status(v) = stuck$ and the algorithm halts. Algorithm P′ works oppositely.

For $k > 1$, suppose that the forest consists of m trees $T_1, T_2, \ldots, T_m$. We assume that our claim holds for a forest of size $< k$. For a forest of size k, there are two cases to consider, depending on whether $m = 1$ or $m > 1$.

If $m = 1$, then Algorithm P first outputs the empty ideal and then the ideal consisting only of the root (node 1). A succeeding ideal contains the root and some additional nodes. Thus, except for at the beginning of the first iteration, the root is not changeable. Hence, Algorithm P, after the first iteration, performs as it would on the forest $\mathcal{F} - \{1\}$, except that the root is included in the ideal generated by the induction hypothesis and halts with F'.

On the other hand, for the last ideal F' generated by Algorithm P, the root is in F'. Algorithm P′ will execute exactly as it would on $\mathcal{F} - \{1\}$, except that the root $\{1\}$ is included in all of the generated ideal, until the ideal of $\mathcal{F}$ that contains only the root, when the root becomes changeable. But then, since the root is free and its children are all null and stuck if they exist, the root is deleted and the empty ideal will be output, and the algorithm will terminate.

If $m > 1$ then we regard $\mathcal{F}$ as consisting of a pair of non-empty subforests $\mathcal{F}_L = T_1, T_2, \ldots, T_p$ and $\mathcal{F}_R = T_{p+1}, \ldots, T_m$, for some p such that $1 \leq p < m$.

By the induction hypothesis, Algorithm P will first generate all ideals of $\mathcal{F}_L$ ending at some ideal F'_L containing no changeable nodes. The next change will be in $\mathcal{F}_L$, and all nodes in F'_L are set free. Thus, by the claim for Algorithm P', the ideals of $\mathcal{F}_L$ will now be generated as constituents of the ideals of $\mathcal{F}_L$ in the opposite order. The last ideal

of $\mathcal{F}_L$ that is generated is thus the empty ideal with unchangeable status. Now again, a change is made in F_R and all the roots in $\mathcal{F}_L$ are set free. The same argument continues until both F_L and F_R become unchangeable, terminating the algorithm.

The symmetric argument holds for Algorithm P' when $k > 1$. □

Ruskey [50] pointed out that if we traverse poset elements in reverse-preorder, we will get an alternative algorithm. Note that if Algorithm P is applied to the antichain poset $\mathcal{A}_n$, then it generates a Binary Reflected Gray Code. In this case, Algorithm P reduces to Algorithm E of Section 3.1 (Algorithm 6 of Ehrlich [11]).

3.5 The Ideals Generated by Algorithm P

Let us discuss the general properties of the list of ideals generated by Algorithm P. *Subcodes* are induced from codewords by a subset of elements in the underlying poset.

LEMMA **3.4** *In the list of codewords induced by any subset of elements (not necessarily an antichain) in a forest poset $\mathcal{F}$ generated by Algorithm P, the Hamming distance between the consecutive subcodes is either 1 or 0.*

Proof: Suppose there exists two consecutive subcodes whose Hamming distance is greater than one. Then the original list of codewords does not form a Gray Code, contradicting Lemma 3.3. □

As we did in the proof of Lemma 3.3, let F' be the extreme ideal output by Algorithm P. Recall that Algorithm P′ is Algorithm P with line (2) replaced with $F \leftarrow F'$, where for all the nodes $v \in F'$, $status(v) = free$. First we show that some nodes in a forest poest such as the rightmost child of the root in the rightmost tree are always included in the extreme ideal F'. We define the *rightmost path* in a forest poset $\mathcal{F}$ to be the path in the rightmost tree of $\mathcal{F}$, extending from the root to the first leaf in the preordering of the tree *from the right*. See Figure 3.4

LEMMA **3.5** *F' includes the nodes along the rightmost path in a forest poset.*

Proof: Since the number of ideals in $J(P)$ is finite, there must be a first time when the leaf node v in the rightmost path (which is unique) is included in an ideal F. In Algorithm P, after the v is included, all the nodes along this path will be unchangeable since each one has at least one child in F. The status of v becomes *stuck*. When the status of v is checked next time, the current ideal F is unchangeable, that is, the current ideal $F = F'$ and the algorithm terminates. □

We further prove the following key lemma. This lemma is later used to obtain the loopless algorithm. This lemma is basically a direct consequence of Algorithm P.

LEMMA **3.6** *For a given subideal F'_v, once the rightmost child of v is deleted and its status becomes stuck, it remains so until the subideals induced by the subtree rooted at v becomes empty (or a singleton $\{v\}$).*

Proof: Note first that if the rightmost child of v is one of the nodes in the rightmost path, it cannot be deleted from an ideal (this case occurs if v itself is in the right most path). So we assume that the rightmost child of v is not in the rightmost path.

Consider the subforest induced by a subtree rooted at v. Since the ideal of $\mathcal{F}$ is empty at the beginning of Algorithm P, the subideal in the subtree is also empty. If the rightmost child of v is not in the rightmost path, there exists a node $r \in \mathcal{F}$ such that $x \prec r$ for all x in the subtree. Let us choose r such that its index in the preorder numbering of nodes is minimum. The node r is either deleted or augmented after the subideal in the subtree

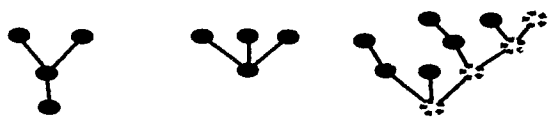

Figure 3.4: A rightmost path in a forest poset

becomes unchangeable (i.e., F'_v). When r is changed, by Algorithm P', the subideal of the subtree rooted at v is regenerated to the empty subideal (a singleton $\{v\}$). In this process, consider the first time that a subideal of the subtree rooted at the rightmost child of v becomes empty (hence the rightmost child is *null* and *stuck*). This rightmost child can be augmented the next time only after a subideal of all subtrees rooted at the rest of children becomes empty. Hence, the assertion follows. □

Lemma 3.6 implies that if we restrict our attention to the subforest of $\mathcal{F}$ rooted at a node v, then for the sequence of ideals generated by Algorithm P, the leftmost child of v is initially augmented and the entire subtree of v will be unchangeable and then the status of nodes in the subtree will be reset to *free*. For Algorithm P', the generating sequence of subideals is reversed. Towards the end of this reversed subsequence of subideals, the rightmost child is deleted and becomes *stuck*, and then, the second rightmost child is deleted and becomes *stuck* while the rightmost node remains *stuck* and *null*, and so on until the leftmost sibling is deleted. One might expect that the subcode induced by a particular branch may form a Binary Reflected Gray code. This does not hold and the reader can easily find a counter example.

One might guess that F', the ideal generated when Algorithm P terminates, must be a certain configuration. It depends on the forest structure. We will discuss one further property of F' beyond Lemma 3.5. Let $p_1, p_2, \ldots, p_m$ be nodes in the rightmost path (p_1 is the root of the rightmost tree in $\mathcal{F}$ and p_m is a leaf node) and $\{t_i\}$ be a set of children nodes of p_i respectively (t_m is trivially empty). Let $T_1, T_2, \ldots, T_m$ be a set of subforests $\mathcal{F}$ rooted at $t_1, t_2, \ldots, t_m$, respectively. Let T_0 denote a set of trees in $\mathcal{F}$ other than the rightmost tree. See Figure 3.5

LEMMA 3.7 *When Algorithm P terminates, all nodes in T_{m-1} are not in F'.*

Proof: If the subtree T_{m-1} is already empty, the assertion trivially holds. Suppose T_{m-1} is not empty. In the list of ideals generated by Algorithm P, consider the first time a subideal of T_{m-1} becomes unchangeable. Since this is the first time that it occurred, the

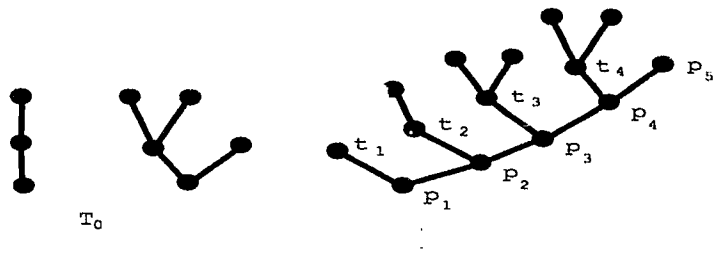

Figure 3.5: A property of the final ideal F

subideal is not empty. After this, the leaf node of the rightmost path will be augmented and its status will be *stuck*. Then, all the subideals induced by $T_0, T_1, \ldots, T_{m-1}$ will be regenerated in backwards order until it becomes unchangeable by the algorithm. In this process, a subideal of T_{m-1} becomes empty and unchangeable. Consider the time that the last node in T_{m-1} was deleted in the subideal of T_{m-1}. This is possible only when a subideal of $T_0, T_1, \ldots, T_{m-2}$ is unchangeable. This implies that when the last node of the subideal of T_{m-1} is deleted, all the nodes in the ideal are unchangeable and the algorithm must terminate. □

Other than the above, we can say only that the subideals induced by T_i from F' are either empty or non-trivial, but unchangeable by themselves.

Chapter 4

Loopless Generation of Forest Poset Ideals

4.1 Motivation

Our Algorithm L described in this chapter also generates all lower ideals of a forest poset without repetition. The algorithm is an improvement over Algorithm P in which it determines the next ideal from the previous in $O(1)$ time.

Algorithm P traverses an ideal in preorder, searching for a changeable node. Observe that in Algorithm P, a non-leaf node of the most recently generated ideal is never changeable. However, in order to run Algorithm P correctly, it is sufficient to check only the changeable nodes. In other words, those non-leaf nodes are irrelevant to Algorithm P; hence, traversing these nodes is unnecessary. In Algorithm L, irrelevant nodes are not scanned. Those relevant nodes which might be augmented or deleted are bound together and called the *useful node set.*

We introduce a data structure for maintaining a list of the useful node set and a simple algorithm to find a free node in the list. This algorithm is later shown to be equivalent, when applied to the antichain poset $\mathcal{A}_n$, to the loopless algorithm for a Gray Code of fixed size by Bitner, Ehrlich, and Reingold [4].

The validity of Algorithm L is proved by induction on the number of elements just

as for Algorithm P. We show that Algorithm L simulates each step of Algorithm P. The delay time complexity $O(1)$ is attained by a careful implementation of the useful node set data structure. We use Algorithm B for generating BRGC as a subroutine.

4.2 The Binary Reflected Gray Code

A loopless algorithm for generating BRGC using links is introduced. This algorithm (Algorithm B) has turned out to be essentially equivalent to that by Bitner, Ehrlich and Reingold [4].

First, we consider the case of a fixed size Gray code. Recall that there exists a natural one-to-one correspondence between the codewords of the Binary Reflected Gray Code and the binary counting sequence for binary strings of length n, which is next described (For details, see Reingold et al. [46]). In the Binary Reflected Gray Code (BRGC), the next bit change is made at the position where a carry stops when we add 1 to the corresponding binary string. Let us consider implementing the above idea directly using pointers. The procedure in pseudocode is presented in Figure 4.2. The $n+1$ nodes of the data structure correspond to the n bits of the binary counter (See Figure 4.1).

Initially, each node in the list consists of two pointers, *run* and *next* and a data field q. The nodes are ordered from the least significant digit to the most significant digit. There is a link for each run of 1's from the least significant digit to the digit immediately to the right of the most significant digit of each run of 1's. (Hence, we need one dummy node to the right of the node for the most significant digit.) The 1's between the second and the last of each run of 1's are replaced by a selfloop pointer. By this rule, the original 0's are also represented by a selfloop pointer.

Although the *next* pointer is unnecessary for a fixed size binary string, this data structure becomes useful when the new loopless algorithm for generating the ideals of a forest poset is introduced. Refer to Figure 4.3.

This data structure for runs of 1's can be created and updated as follows. Initially, there are n selfloops on the list. If a 1 is added to the binary string, we find the location

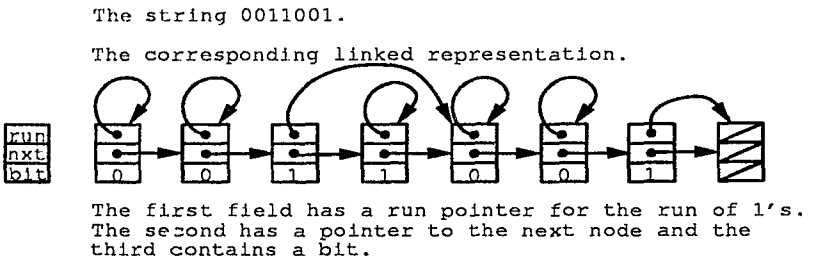

Figure 4.1: The data structure for the binary counter algorithm

where the carry operation stops. This will be position q and is pointed to by the *run* pointer of the least significant digit (step (2)). Let r be next to q (step (3)). We invert the binary bit at position q to obtain the next BRGC codeword (step (4)). Then, we reset the *run* pointer for the first run of 1's, which caused the carry (step (5)).

If there is no run of 1's after q, the *run* pointer of node q points to the next node to indicate a run of a single 1. If there is a run of 1's after q, we combine these two runs. (steps (6) and (7)).

Termination occurs when a 1 is added to the least significant digit and the rest of the digits are already a single run of 1's. We detect whether or not the pointer for the run of 1's of the least significant digit points to the dummy node (step (8)). We note that all of the operations described above can be done in constant time by simple link manipulation.

The idea of this algorithm is used as one of the essential parts of the loopless algorithm for forest poset ideals introduced later. The development of this algorithm and the loopless algorithm (Algorithm 5.7(c)) by Bitner et al. [4] are quite different. However, an equivalence between the two algorithms is shown step by step in the comments of Figure 4.2. The following lemma summarizes the properties of the algorithm. The validity of the algorithm is readily seen.

LEMMA 4.1 *Algorithm B is a loopless algorithm for generating the Binary Reflected Gray Code.*

□

(* x.y means y is field of x and a pointer *) { correspondence to the BER algorithm}

q.on_off: boolean

(1) **repeat**

(2) $q \leftarrow head.run;$ $\{i \leftarrow \tau_0\}$

(3) $r \leftarrow q.next;$

(4) $q.on\_off \leftarrow \overline{q.on\_off};$ $\{g_i \leftarrow \overline{g_i}\}$

(5) $head.run \leftarrow head;$ $\{\tau_0 \leftarrow 1\}$

(6) $q.run \leftarrow r.run;$ $\{\tau_{i-1} \leftarrow \tau_i\}$

(7) $r.run \leftarrow r;$ $\{t_i \leftarrow i+1\}$

(8) **until** $q = tail;$

Figure 4.2: Algorithm B

4.3 The Algorithm

Algorithm L mimics Algorithm P in an efficient manner. We create and maintain a set of *useful nodes* U in Algorithm L. First, we will define a *useful node set* U_P with respect to an ideal of $\mathcal{F}$ for Algorithm P. Later we will show that U_P is equal to U and that it is sufficient to maintain the status of only the nodes in U in order to determine the next changeable node. For an example of the useful node set, see Figure 4.3.

DEFINITION 4.1 $U_P = V_1 \cup V_2 \cup V_3 \cup V_4 \cup V_5$, *where*

$V_1 = \{v \notin F \mid v \in Roots(\mathcal{F})\}$

$V_2 = \{v \notin F \mid par(v) \in F, par(v) \notin Leaf(F)\}$

$V_3 = \{v \notin F \mid par(v) \in Leaf(F), status(par(v)) = stuck\}$

$V_4 = \{v \in F \mid v \in Leaf(\mathcal{F}), status(v) = stuck\}$

$V_5 = \{v \in F \mid v \in Leaf(F), status(v) = free\}.$

For a given forest ideal F, it is easily seen that the set of useful nodes U_P forms an antichain with respect to $\mathcal{F}$. Assume that $status(v) = free$ and v is in V_1, V_2, or V_3 so that v is deletable. If $status(v) = free$ and v is in V_5, then v is augmentable. (We never

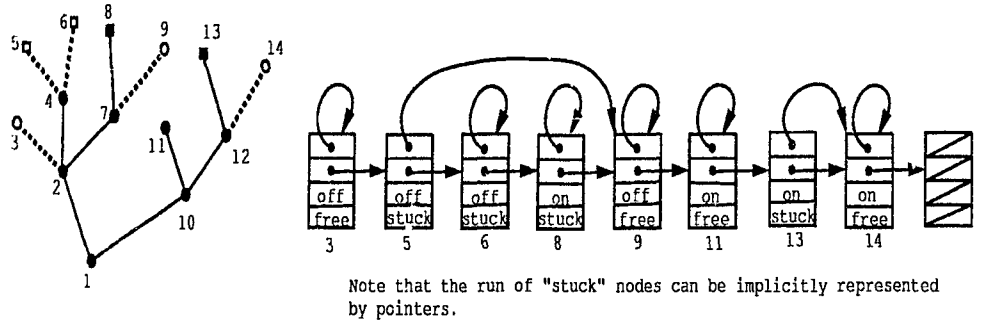

Figure 4.3: The useful node list

have the case V_4.) We now present the algorithm in pseudocode (See Figure 4.4). The Boolean variable $Penult(v)$ is true if and only if v is a leftmost sibling and all siblings of v are stuck and null. The subideals of the subtree rooted at v are generated and regenerated in reflected order in the process of generating the ideals of $\mathcal{F}$. In the regeneration process of subideals, the subideal just before the empty subideal is the subideal consisting of a leftmost child of v (This motivates the term "penultimate" for the Boolean variable). Thus, for a given node v, that $Penult(v) = true$ implies that one cyclic sequence of generation and regeneration of subideals was completed for the subtree rooted at $par(v)$. In the regeneration process, the extremes are either the empty subideal or F_v', where F_v' is the final subideal output when Algorithm P is applied to the subtree rooted at v. The notation $Children(v)$ has the usual meaning.

Initially, U consists of a set of roots in $\mathcal{F}$, which are all *free* (line (1)). Algorithm L runs as long as there exists one *free* node in U (line (5)). If there is no *free* node, we will go out of the while loop and terminate the algorithm; otherwise, we update the set U. We observe that if there is a free node in U, then the current ideal F is changeable, and conversely, if F is unchangeable, then there is no free node in U. Hence, asking whether or not there is a free node in U is equivalent to checking whether or not the current ideal F is changeable. This check appears in step (4) in Algorithm P. Let v be the changeable node with the least index in the preordering of the nodes (line (6)). This v, whether to be augmented or deleted, is assigned the status *stuck* and any node $x \prec v$ is assigned the

(1) $U \leftarrow$ { the roots of the trees of $\mathcal{F}$};
(2) **forall** $x \in U$ **do** $status(x) \leftarrow free$;
(3) $F \leftarrow \emptyset$;
(4) Output(F);
(5) **while** $\{x \in U \mid status(x) = free\} \neq \emptyset$ **do begin**
(6) $v \leftarrow \min\{x \in U \mid status(x) = free\}$;
(7) **forall** $\{x \in U \mid x \prec v\}$ **do** $status(x) \leftarrow free$;
(8) $status(v) \leftarrow stuck$;
(9) **if** $v \in F$ **then begin** (* delete v *)
(10) $F \leftarrow F \setminus \{v\}$;
(11) **if not** $Penult(v)$ **then** (* case 1.1 do nothing*)
(12) **else begin** (* case 1.2 *)
(13) $U \leftarrow U \setminus Siblings(v) \cup \{par(v)\}$;
(14) **forall** $x \in Siblings(v)$ **do** $status(x) \leftarrow free$;
(15) **end**
(16) **end else begin** (* augment v *)
(17) $F \leftarrow F \cup \{v\}$;
(18) **if** v is a leaf of $\mathcal{F}$ **then** (* case 2.1 do nothing *)
(19) **else begin** (* case 2.2 *)
(20) $U \leftarrow U \setminus \{v\} \cup Children(v)$;
(21) **end**
(22) **end**;
(23) Output(F);
(24) **end**;

Figure 4.4: Algorithm L

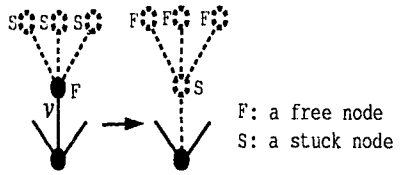

Figure 4.5: Case 1.1. Deletion in the case not covered by Case 1.2

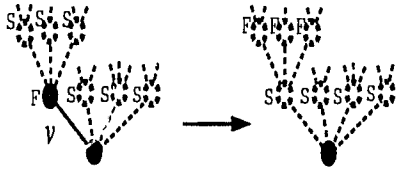

Figure 4.6: Case 1.2. Deletion of the first free node that is the first sibling while the rest of the siblings are null and stuck

status *free* (lines (7) and (8)). This prevents the algorithm from entering an infinite loop alternately augmenting and deleting v. In Algorithm P, the equivalent step appeared in steps (6) and (7). For this v, there are two basic cases (with some subcases): deleting v and augmenting v depending on whether or not v is in F (line (9)). This classification is inherited from step (8) of Algorithm P.

Case 1: If $v \in F$, delete a leaf node from F (line (10)). Here, two subcases can arise depending on whether or not v is penult. The node status for each subcase of Algorithm P is depicted in Figure 4.6 and Figure 4.5.

Case 1.1: In the first subcase, v is not penult; that is, v is not a leftmost sibling, or v is a leftmost sibling but there is a sibling of v which is in F (and also in U). In this case, the useful node set remains the same (line (11)).

Case 1.2: The second subcase is that v is penult. That v is penult implies that subideals with respect to the subtree of $par(v)$ are generated from the singleton subideal $par(v)$ to some non-trivial extreme subideal and regenerated backwards so that deleting v is justified as the final step of the generation and regeneration of the subideal of the subtree of $par(v)$.

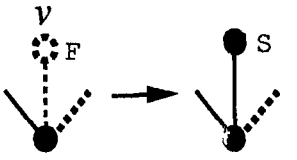

Figure 4.7: Case 2.1. Augmentation on a node having no children

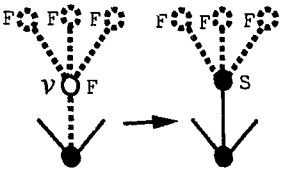

Figure 4.8: Case 2.2. Augmentation on a node having children

Note that after deleting v, we have the empty subideal of the subtree of $par(v)$, which is the initial extreme of the subideal of the subtree. Observe also that after deleting v, the next candidate node to be treated is one of the nodes that have been reset to *free*. Among all the nodes reset to *free* in line (7), only $par(v)$ is not yet in U, while v is reset to *stuck* and taken out of the ideal. Thus, we can update U by replacing all the children nodes, of the underlying branch, which are now all *stuck*, with the parent node (line (13)). (Note that the parent node $par(v)$ is the predecessor of v in the preorder.) Since the nodes outside of U are no longer relevant to determining the next changeable node, they are reset to free (line (14)). This operation makes Algorithm L different from Algorithm P.

Case 2: In the second case, $v \notin F$. Augment the node v to form a leaf in F (line (17)). Two subcases can arise depending on whether or not v has any children. The node status for each subcase of Algorithm P is depicted in Figure 4.7 and Figure 4.8.

Case 2.1: If v is a leaf of $\mathcal{F}$, the list U remains the same (line (18)).

Case 2.2: If v has children, they are not in F. So, we replace v by its children in U (line (20)). The status of the children is *free* but they are not in F. The reason why these children are brought into U is clarified by considering the corresponding procedure in Algorithm P. In Algorithm P, in the next while loop, these children would be possible

candidate nodes to be changed, since the *status* of their parent is *stuck*. Observe that the above operations do not affect the other stuck nodes since the free node v was replaced by new free nodes. After updating U, we return to line (5).

4.4 The Validity of Algorithm L

The validity of Algorithm L can be easily obtained by induction on the number of nodes in a poset as in the proof of Lemma 3.3. Here, we show the equivalence between Algorithm L and Algorithm P by showing that, for a given input, those algorithms produce the same ideal at each transition.

Let v_P be the node chosen for augmentation or deletion of the current ideal F at a given transition at each transition by Algorithm P and v be the node chosen at the corresponding transition by Algorithm L. Let U_P be the useful node set for this transition defined on $\mathcal{F}$ by Algorithm P and U be the corresponding useful node set in Algorithm L. For $X, Y \subseteq S(\mathcal{P})$, $X = Y$ denotes set equality (as usual) in both X and Y are equal, and we write $X \simeq Y$ to denote that not only $X = Y$ but also the status of the elements in both sets is the same. Let $A = S(\mathcal{F}) - F'$ where $F' = F \cup U$. Similarly, let $B = F - U$ (Note that both F' and B are ideals). Let $A_P = S(\mathcal{F}) - FA$ where FA is defined as in section 3.3; that is, $FA = \{x \in S(\mathcal{P}) \mid x \in F, \text{ or } par(x) \in F\}$.

Note that v_P is in U_P in Algorithm P. We show below that $U_P \simeq U$ and for all $x \in A \cup B$, $status(x) = free$ at each iteration of the while loop in Algorithm L, and for all $x \in A_P$, $status(x) = free$ in Algorithm P, by induction on the number of loops in Algorithms P and L.

LEMMA 4.2 *In Algorithm P, for $x \in U_P$, if the status of x is reset to free in step (6), then $x \in U_P$ for the next iteration.*

PROOF: If x satisfies either of the conditions V_3 or V_5, then $status(x) = free$. So we consider the rest of the cases. If x satisfies either of the conditions V_1 or V_2, then x stays in U_P regardless of its status. If x satisfies the condition V_4, then x satisfies condition V_5

after x is reset to $free$. □

LEMMA 4.3 *Algorithm L produces exactly the same sequence of ideals as Algorithm P given the same input.*

PROOF:

At the $K = 0$ th step, both U_P and U contain all of the $free$ root nodes. Obviously, the first root node (labeled one) is chosen in either algorithm. By the initialization for Algorithm L, for all elements $x \in A$, $status(x) = free$ and B is empty.

Suppose that after the K th loop, $U_P \simeq U$. Consider the $K+1$ st step. We know that the first $free$ node v_P is found in U_P; therefore, if $U_P \simeq U$, $v = v_P$ by Algorithm L, noting that v is the first free node in U.

If there is no free node in U_P and U, both algorithms terminate. So our claims hold. If there is a free node, the first node $v(= v_P)$ is either augmented or deleted in both algorithms.

Now, we show that after updating the useful node sets in both Algorithms P and L, our second claim holds.

Case 1: Suppose $v \in F$, then v is deleted from F. There are two subcases. The first subcase is that v is not penult and the second subcase is that v is penult.

Case 1.1: v is not penult. (Hence, there are at least two siblings.) In Algorithm L, U is unchanged. In Algorithm P, for an element x such that $x \in Siblings(v_P)$, $x \neq v_P$ $x \in U_P$ and $x \in F$, we have the situation that prior to the deletion of v_P from F condition V_5 is met by x, and after the deletion condition V_2 is met. Thus, U_P is unchanged.

Consider the case in which there is no such x. There are as many cases as the number of children of v in which v_P is the only child of $par(v_P)$ with respect to F. Consider the sublist of subideals induced by the subtree rooted at $par(v_P)$ from the list of ideals generated by Algorithm P. The first subideal appeared in this sublist is the empty subideal (or a singleton $\{par(v_P)\}$) and the last subideal is also empty. Let A_s denote the subideal having v_P as the only child with respect to F. Therefore, the subideal A_s which precedes

the subideal F_s and the subideal B_s which succeeds the subideal F_s in the sublist of subideals, A_s and B_s are not the empty subideals; A_s and B_s must have an element besides v_P and $par(v_P)$. Hence, the node v_P was never deleted during the transition from A_s to F_s and from F_s to B_s so that the resultant is the empty subideal in the sublist (status of v_P was not $free$). This case has never occurred. Therefore, $U_P = U$.

For both algorithms P and L, $status(v) = stuck$. In Algorithm P, for all $x \prec v$ such that $x \in FA$, $status(x) = free$. In particular, for all $x \prec v$ such that $x \in U_P$, $status(x) = free$. For all $x \succ v$ such that $x \in U_P$, $status(x)$ remain the same. As a result, the elements and their status in A_P remain the same.

In Algorithm L, by line (7) for all $x \prec v$ such that $x \in U_L$, $status(x) = free$. For all $x \succ v$ such that $x \in U_L$, $status(x)$ remain the same. Thus, $U_P \simeq U$. Obviously, A and B remain the same.

Case 1.2: Suppose v is penult. In Algorithm L, $par(v)$ replaces $Siblings(v)$ in U. In Algorithm P, v_P is deleted from F and $status(v)$ becomes $stuck$. Since $par(v) \in F$ but $status(par(v)) = free$, the elements of $Siblings(v)$ do not satisfy any of the case V_1, V_2 and V_3; thus, they are taken away from U_P. On the other hand, since $status(par(v)) = free$ and $par(v) \in Leaf(F)$, $par(v)$ satisfies the case V_5; hence, $par(v) \in U_P$. For all $x \prec v_P$ such that $x \in U_P$, x remains U_P after $status(x)$ is reset free by Lemma 4.2. Thus, $U_P = U$.

In Algorithm P, for all $x \prec v_P$ such that $x \in FA$, $status(x) = free$. In particular, for all $x \prec v_P$ such that $x \in U_P$, $status(x) = free$, and $status(par(v_P)) = free$ since $par(v_P) \prec v$. For all $x \succ v_P$ such that $x \in U_P$ but not any of $siblings(v_P)$, $status(x)$ remain the same. Note that the status of all $Siblings(v_P)$ except v_P itself is $stuck$ before they are replaced by $par(v)$. We are also sure that status of elements in A_P is $free$ since just before v_P is deleted, status of $Children(v_P)$ which are added to A_P, has been reset to $free$ in step (8).

In Algorithm L, for all $x \prec v$ such that $x \in U$, $status(x) = free$. We have $status(par(v)) = free$ since $par(v)$ was in B. Since no node was added to B, B has only $free$ nodes. For all $x \succ v$ such that $x \in U$ but $Siblings(v) \cap U = \emptyset$, $status(x)$ remains the same. Since the

status of all $Siblings(v)$ becomes $free$ in line (14), the updated A has only $free$ nodes.

Case 2: Suppose $v \notin F$, then F is augmented with v. There are two subcases depending on whether or not v is a leaf of $\mathcal{F}$.

Case 2.1: Suppose v is a leaf of $\mathcal{F}$. The claims of this case are proved similarly to Case 1.1 (non penult v is deleted).

Case 2.2: Suppose v has children. In Algorithm L, after F is augmented with v, v is replaced by $Children(v)$ in line (20). In Algorithm P, after F is augmented with v_P, $status(v_P)$ becomes $stuck$. Therefore, unless v_P is a leaf of $\mathcal{F}$, v_P is no longer in U_P, while the elements of $Children(v_P)$ satisfy condition V_3 of U_P. For all $x \prec v$ such that $x \in U_P$, x remains in U_P after $status(v)$ is reset $free$ by Lemma 4.2. Thus, $U_P = U$.

We now show that the status of the elements of U_P and U are equal. In Algorithm P, for all $x \prec v_P$ such that $x \in FA$, $status(x) = free$. In particular, for all $x \prec v_P$ such that $x \in U_P$, $status(x) = free$; and $par(v_P) = free$ since $par(v_P) \prec v$. For all $x \succ v_P$ such that $x \in U_P$, $status(x)$ remains the same. Note that the status of the elements of $Children(v_P)$ is $free$ since they were in A_P previously.

In Algorithm L, for all $x \prec v$ such that $x \in U$, $status(x) = free$. We have $status(par(v))$ $= free$ since $par(v)$ was in B previously. The status of the remaining nodes in B is obviously $free$. For all $x \succ v$ such that $x \in U$, $status(x)$ remains the same. The elements of $Children(v)$ which are newly entered in U are all $free$ since $Children(v)$ were previously in A, whose elements are all $free$ by the assumption.

Thus, we have shown that in any case our claims hold after the $k+1$ st loop. □

4.5 The Looplessness of Algorithm L

Our main concerns here are whether or not we can update a useful node set and pointers for runs of stuck nodes in constant time and can also terminate the algorithm in constant time.

The useful node set U in the previous section is now linearly linked. The order of the

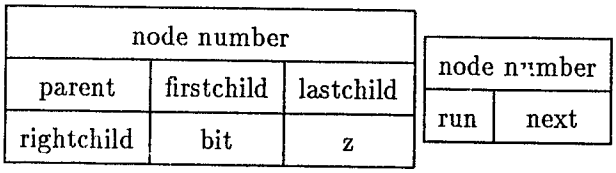

Figure 4.9: Data structures for a node in a forest poset and a node in the useful node list

nodes in the list is inherited from the preorder ordering of the nodes in the forest poset $\mathcal{F}$. The set of nodes in the useful node list forms a maximal antichain in $\mathcal{F}$. Observe that the next changeable node is the leftmost free node, either augmentable or deletable, in the linked list. We also create a link from the first node to the last node of each run of stuck nodes as in Algorithm B; hence, the pointer of the first node points to the node immediately to the right of the last node. In the algorithm, we will assume the data structure in Figure 4.9. This data structure is the combination of that for a forest poset and that for binary counter algorithm. Refer to Figure 4.1.

Considering the status *free* as the bit value 0 and the status *stuck* as the bit value 1, the run of stuck nodes is analogous to the run of ones in Algorithm B. To attain a loopless algorithm, we introduce the *z value* associated with each node in $\mathcal{F}$. Let $RS(v) = \{x \in Siblings(v) \mid v \leq x\}$. In other words, $RS(v)$ is the set of nodes to the right of v among the siblings of v. A pointer *rightchild*(v) points to the sibling node to the right of v.

DEFINITION 4.2 *Let F be an ideal in $\mathcal{F}$. Suppose v is a node such that v is a leaf of F and $status(v) = free$. Then we define $z(v) = 1$ if rightchild(v) $= NIL$ or for all $x \in RS(v)$, $x \notin F$ and $status(x) = stuck$; otherwise, $z(v)$ is defined to be 0.*

In a given branch, if $z(v) = 1$, then the status of all the siblings to the right of v is stuck and not in an ideal F.

By the definition of the z value, if z(the leftmost child of $par(v)$)=1, then Case 1.1 holds (v is penult). That is, the first child is *free* and belongs to the current ideal F, and all of the rest of the siblings of v (if they exist) are *stuck* and not in F. From the description of Algorithm L in the previous section, we will show that every step can be done in constant time.

LEMMA 4.4 *Algorithm L is loopless.*

Proof: The first free node v can be found in a useful node list U by a link as in Algorithm B in constant time. We show that updating U can be accomplished constant time by link manipulation.

There are two cases when deletion occurs. The first case is that v is penult (case 1.1). This involves testing whether or not v is a leftmost child, which can be done by checking if the leftmost child of $par(v)$ is equal to v, and testing whether or not $z(v)$ is one. These tests can be obviously done in constant time. Replacing the *stuck* children by their parent can be done in constant time by changing the *next* pointer of the node before v in U to point to $par(v)$ and the *next* pointer of $par(v)$ to point to the node after the rightmost sibling of v in U.

The second case is that v is not penult (case 1.2). The algorithm deletes v from the current ideal F (but v remains in U), and updates the status of v to *stuck* and the *run* pointer of v in U. These updates take only constant time by link operations.

Now, consider how the z value of v is updated. By Lemma 3.6, siblings of v become *stuck* and *null* from the right to the left towards the end of the cyclic sequence of the generation of the subideals of the subtree rooted at $par(v)$. Therefore, if v is not penult and deleted, there must be some sibling $x \in F$ immediately to the left of v. Thus, we set $z(x) = 1$. This updating takes only constant time.

When augmentation occurs, v remains in U if it is a leaf node in $\mathcal{F}$ (case 2.1); otherwise, v is replaced by its children (case 2.2). The first case certainly takes constant time since only v is added to F, while U remains unchanged. In the second case, since the siblings of v are linked together in advance, we can replace v by its children by changing the *next* pointer of the node before v in U to point to the first child and the *next* pointer of the last child to point to the node after v in U. This operation takes only constant time by link manipulation.

Finally, we consider the time to terminate the algorithm. If F is unchangeable, U consists entirely of *stuck* nodes. Thus, as in Algorithm B, the termination can be detected by a simple link operation in constant time. Therefore, all the operations involved with

producing the next ideal can be done in constant time. □

From Lemmas 4.3 and 4.4, we conclude the following.

THEOREM 4.1 *Algorithm L produces exactly the same sequence of ideals as Algorithm P produces given the same input. Moreover, Algorithm L runs with constant transition time from one generated ideal to the next.*

□

4.6 Application

There is an algorithm by Steiner for generating ideals in a (general) poset [53]. A second approach can be found by using Algorithm P or Algorithm L as a subprocedure. First, we find a spanning forest in a poset (If a poset is a forest, first create one pseudo element and take the ordinal sum of the forest poset with the new element. The result is a tree poset with the new element as the minimal element.) We then apply the algorithms discussed to the spanning forest. Since not all the leaves of a given subtree of the spanning forest are members of an ideal of a poset, when we generate a subtree of the spanning forest, we have to check whether each leaf is neither a descendant nor an ancestor of other leaves.

At every enumeration of a subtree, exactly one node is different from the previous one, so if a new node is added or a new leaf node is revealed after deleting an existing leaf node, we check to see if the new node or the discovered leaf node has any relation to the rest of the nodes. This can be done in time $O(n)$. Constructing a spanning tree in the extended poset takes $O(n)$ time. Thus, if we use our loopless algorithm for ideal generation on the spanning tree, the total time complexity would be $O(Nn)$, where N is a number of ideals in the spanning tree.

Chapter 5

Acyclic Digraph Generation

5.1 Introduction

In this chapter we are concerned with the generation of unlabeled acyclic digraphs. As far as the numbers of labeled and unlabeled acyclic digraphs are concerned, Robinson [47] obtained the formulas to count these digraphs. However, the formula for unlabeled acyclic digraph is strictly speaking not a closed formula (The numbers of the acyclic digraphs are calculated recursively with some complication). Since graphical forms of posets, finite lattices and finite topologies are essentially acyclic digraphs, first we shall give a technique for generating acyclic digraphs.

Let $\mathcal{S}_p$ be the set of adjacency matrices $\mathcal{M}$ of size $p \times p$. Let $\mathcal{G}$ be a given permutation group of degree p. Two matrices $\mathcal{M}_1$ and $\mathcal{M}_2$ in $\mathcal{S}_p$ are said to be isomorphic if there exists a permutation in $\mathcal{G}$ which, acting on the indices of $\mathcal{M}_1$, transforms it into $\mathcal{M}_2$. We will present an orderly algorithm for generating the lists $\mathcal{L}_p$, which contains a single representative from each isomorphic class of unlabeled acyclic digraphs of $\mathcal{S}_p$. We then present orderly algorithms for generating acyclic transitive digraphs, finite lattices, and transitive digraphs.

5.2 Acyclic Digraph Representation

In order to employ orderly algorithms, we must find a canonical form for the underlying graphs. We first need the following fundamental properties of an acyclic digraph.

Let an acyclic digraph of size p be layered according to the heights of its vertices.

PROPOSITION **5.1** *If a directed graph P is acyclic, then each vertex is uniquely placed on one of the disjoint layers according to its height, where all sources constitute the first layer, and within a layer no vertices are adjacent. Furthermore, all arc directions are from the i-th to $i+k$-th layer, for some $k > 0$.*

Proof. We first prove by contradiction that there are no arcs between vertices in the same layer. Suppose there exists an arc (x, y) within some layer. Then, by definition, $h(x) = h(y)$. But, since the arc (x, y) is present, clearly $h(y) > h(x)$. This proves that there can be no such arc.

Now we show that all arcs are from the lower levels to the upper levels. If there exists an arc (x, y) from the jth layer to the ith layer, $i < j$, then $h(y) = j < i = h(x)$. This is a contradiction. □

Hence, we arrive at the following representation of acyclic digraphs. Let an acyclic digraph of size p be layered according to the heights of vertices and represented by an adjacency matrix $\mathcal{M}$ of size $p \times p$. We view the vertices as ordered by their index in the adjacency matrix. In defining the canonical form we will choose an ordering which accommodates the heights of the vertices.

Let X_i denote the set of vertices in the ith layer and l_i be the number of vertices in X_i, $|X_i|=l_i$. Consider a *composition* of p to be a sequence of integers $l_1, l_2, ...l_D$ such that $p = l_1 + l_2 + \ldots + l_D$.

Let M_i be the submatrix induced from $\mathcal{M}$ by the rows corresponding to the vertices in $X_1 \cup X_2 \cup \ldots \cup X_{i-1}$ and the columns corresponding to the vertices in X_i. Note that entries in $\mathcal{M}$ whose row and column indices correspond to vertices in X_i are all zero. We also note that all entries $a_{i,j}$, $i \geq j$, in $\mathcal{M}$ are zeros.

Thus, a list of submatrices M=$\{M_1, M_2, \ldots, M_{D-1}\}$ represents an acyclic digraph of height D. For an example, see Figure 5.1. For the rest of this paper, we assume a graph is represented in this way by a list of submatrices M.

The vector v *associated* with the list of submatrices M is the sequence of integers obtained by considering each of the columns of the submatrices M in the natural order as a binary number, which we call *the value of the column.* The lexical order of matrices is defined by the lexical order of associated vectors. See Figure 5.1.

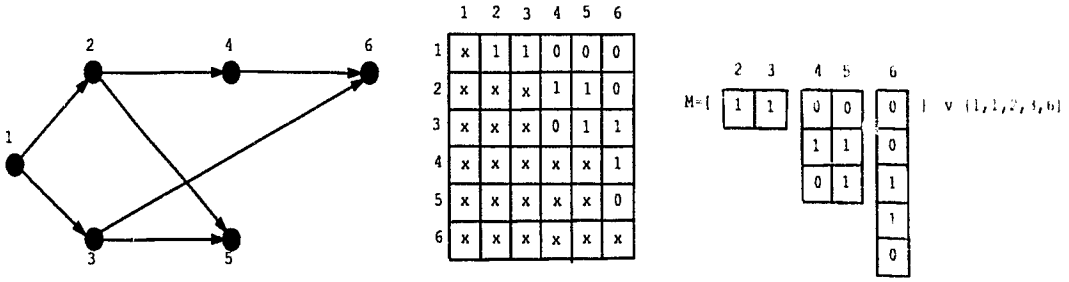

Figure 5.1: A layered acyclic digraph and its matrix form.

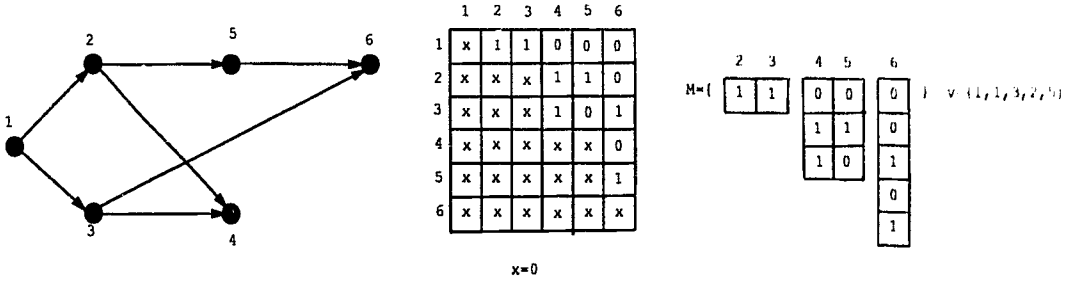

Figure 5.2: The canonical acyclic digraph and its code

The canonical form of an acyclic digraph is the acyclic digraph where the vertices are ordered by their heights in such a way that the resulting matrix M defined above is lexically the largest possible. By a *canonical acyclic digraph* we mean an acyclic digraph in canonical form. The *code* of a graph G is either the canonical matrix M or equivalently the associated vector v. In Figure 5.2, after exchanging the labels of vertices 4 and 5, we obtain the vector $v = \{1, 1, 3, 2, 5\}$. This new labeling of vertices, in fact, yields the largest

associated vector which is, therefore, the *code* of the isomorphism class of G.

5.3 Augmentation of an Acyclic Digraph

In this section, we will define the augmentation of an acyclic digraph which is represented in the form described in the previous section. First, we recall the following obvious property of an acyclic digraph.

PROPOSITION **5.2** *Let G be an acyclic digraph on p vertices. Then, for any vertex v of G, G - v is also an acyclic digraph.*

□

From Proposition 5.1 and Proposition 5.2, we immediately obtain the following.

LEMMA **5.1** *Let $G_{p,h}$ be a canonical acyclic digraph of height h and p vertices. If we delete the vertex v of largest index in the last layer, then G - v is either a canonical acyclic digraph of p-1 vertices and height h, or a canonical acyclic digraph of p-1 vertices and height h-1.*

Proof. From Proposition 5.2, $G - v$ is an acyclic digraph. The height of $G - v$ is h if the last layer of G contains more than one vertex, and $h - 1$ otherwise. Canonicity of the $G - v$ is shown as follows. Suppose after the deletion of the vertex, hence after the deletion of the corresponding column and row in the underlying matrix, the resultant matrix M' is not canonical. Then, there exists a permutation matrix S on $\mathcal{M}'$ for $G - v$ such that after the permutation, the resultant matrix $M''(= S \times M' \times S)$ is lexically maximal. Apply the same permutation on the first $p - 1$ vertices of the original matrix $\mathcal{M}$, then the resultant matrix is larger than the original. This leads to a contradiction. □

Hence, we have two types of augmentation. A *type I augmentation* is such that a vertex to be added forms a new last layer, and is connected by at least one arc from a vertex in

the old last layer, and the height of the digraph is increased by 1. A *type II augmentation* is such that a vertex is added to the last layer of a canonical acyclic digraph together with at least one arc from $h-1$-st level to this vertex. In a type II augmentation, the height of the digraph remains the same.

Let us describe one general property of the canonical matrix of a general graph, The canonical matrix of a graph is represented by an upper triangular matrix obtained by excluding diagonal elements and the lower triangular part of the adjacency matrix $\mathcal{M}$ of the underlying graph.

In the following lemma 5.2 and lemma 5.3, v_i represents the ith column vector of some submatrix M of $\mathcal{M}$.

LEMMA **5.2** *If a matrix* M *for a graph G is canonical, then for* $i < j$, v_j^i *is lexically smaller than or equal to* v_i.

Proof. Suppose $\mathcal{M}$ is the canonical matrix for the graph but there exist i and j such that v_i is lexically smaller than v_j^i. We exchange the ith and jth columns and rows simultaneously in $\mathcal{M}$. Although this permutation affects the values of column vectors v_k, $k > i$, since column vector v_i comprises more significant digits than those of v_k, $k > i$, the overall resultant matrix M′ is lexically larger than M, leading to a contradiction. □

By applying Lemma 5.2 to the acyclic digraph of h layers, we obtain the following. Let v_{M_k} denote the subvector of v corresponding to the submatrix M_k.

LEMMA **5.3** *If a matrix* M=$\{M_1,M_2,...,M_{h-1}\}$ *for the acyclic digraph of h layers is canonical, then in the associated vector* $v = (v_{M_1}, v_{M_2}, \ldots, v_{M_{h-1}})$, *the integer subsequence* v_{M_k} *is monotone non-increasing.*

Proof. The assertion follows noting that to test canonicity, it is sufficient to permute labeings of vertices only within the same layer, and the sizes of column vectors $\{c_i\} \in v_{M_k}$ for the kth layer are equal. □

Regarding isolated vertices, we have the following lemma:

LEMMA **5.4** *If an acyclic digraph of h layers is canonical and has isolated vertices, they are present in only the first layer X_1. Further, those isolated vertices have largest possible labelings in X_1.*

PROOF: Suppose a graph is canonical and an isolated vertex is X_i, $i > 1$, then by relabeling the isolated vertex so that they are in X_1, we obtain a lexically larger matrix. Then, if an isolated vertex does not have the largest possible index in X_1, the vertex can be relabeled by the largest possible index and we obtain a lexically larger matrix. □

From Lemmas 5.1 - 5.4 we obtain the following result.

THEOREM **5.1** *Let $ACY(l_1, l_2, l_3, \ldots, l_h)$ be a canonical acyclic digraph, not necessarily connected, with height h and $l_1 + l_2 + \ldots + l_h = p$ vertices. The acyclic digraph $ACY(1)$ consists of a single vertex. Then $ACY(l_1, l_2, l_3, \ldots, l_h)$ can be obtained by augmentation either from $ACY(l_1', l_2, l_3, \ldots, l_{h-1})$ with $l_1 - l_1'$ isolated vertices in the first layer if $l_h = 1$ or $ACY(l_1'', l_2, l_3, \ldots, l_h - 1)$ with $l_1 - l_1''$ isolated vertices in the first layer if $l_h > 1$. An augmenting vertex is connected by arcs from at least one vertex on the $(h-1)$−st layer, and from any additional isolated vertices in the first layer.*

5.4 Generating Acyclic Digraphs.

The number of nonisomorphic acyclic digraphs grows exponentially as the number of vertices increases. Since a digraph having isolated vertices is structurally almost the same as that without those isolated vertices, it is not necessary to store all digraphs; it is more efficient to store only those without isolated vertices. Let $A_{p,h}$ denote a canonical acyclic digraph with p vertices and height h. Let $L_{p,h}$ be a list of canonical acyclic digraphs $A_{p,h}$ that are not necessarily connected but that have no isolated vertices. Note that $p \geq h$ always holds.

$A_{p,h}$ can be obtained in two kinds of ways; either by a type I augmentation from a canonical acyclic digraph $A_{i,h-1}$ together with the addition of $p - 1 - i$ isolated vertices

to the first layer for, or by a type II augmentation from a canonical acyclic digraph $A_{i,h}$ together with the addition of $p-1-i$ isolated vertices to the first layer.

Thus we observe that to produce the list $L_{p,h}$, it is sufficient to have the set of lists $\{L_{i,j} \mid i \geq 2, j \leq i \leq j+(p-l), i \neq p, j \neq h\}$, and $\{L_{i,h} \mid l \leq i \leq l+(p-l)-1\}$. For example, to produce the list $L_{4,3}$, we need first generate only the lists $L_{1,1}$, $L_{2,1}$, $L_{2,2}$, $L_{3,2}$, and $L_{3,3}$, but not the lists $L_{3,1}$, $L_{4,1}$, and $L_{4,2}$. Refer to Figure 5.3.

	1	2	3	4	*	*	j	*
1	$L_{1,1}$							
2	$L_{2,1}$	$L_{2,2}$						
3	$L_{3,1}$	$L_{3,2}$	$L_{3,3}$					
4	$L_{4,1}$	$L_{4,2}$	$L_{4,3}$	$L_{4,4}$				
*	*	*	*	*	*			
*	*	*	*	*	*	*		
i	$L_{i,1}$	*	*	*	*	*	$L_{i,j}$	
*	*	*	*	*	*	*	*	*

Figure 5.3: The conceptual scheme of a set of lists $L_{i,j}$ ordered by the number of vertices and height.

In order to make sure that the algorithm works correctly, we have to know that each canonical acyclic digraph can be obtained by augmentation from a canonical acyclic digraph. We note that two non-isomorphic acyclic digraphs may be augmented to an isomorphic acyclic digraph. See Figure 5.4. If two non-isomorphic acyclic digraphs are canonical, however, they cannot be augmented to the same canonical acyclic digraph.

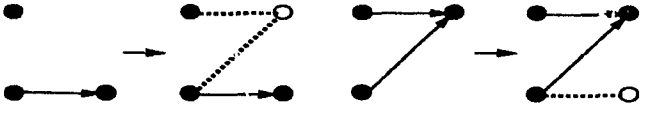

Figure 5.4: Two non-isomorphic acyclic digraphs that augment to isomorphic digraphs

We summarize this fact in the following lemma.

LEMMA **5.5** *Canonical acyclic digraphs from different lists cannot be augmented to isomorphic canonical acyclic digraphs.*

Proof. Suppose, for some $x > 0$ two non-isomorphic canonical acyclic digraphs with the indicated parameters $A_{p-x,h-1}$ and $A_{p-x,h}$, $x > 0$, are obtained by a type II and type I augmentations, respectively, to the same canonical acyclic digraph $A_{p,h}$. Let v', v'', and v be the associated vectors of these digraphs. If we delete the last vertex in v, then the result must be equal to v' and v'' by our assumption. But the heights of the digraphs for v' and v'' are different; that is, $v' \neq v''$, leading to a contradiction.

Suppose type I augmentations are made to non-isomorphic canonical acyclic digraphs $A_{p-X,h}$ and $A_{p-Y,h}$, $X \neq Y$ and X,$Y > 0$, yielding the same canonical acyclic digraph $A_{p,h}$. Appropriate isolated vertices are thus added to the first layer before the augmentation. Let v', v'', and v be the associated vectors of these digraphs. If we delete the last vertex in v, then the result must be equal to v' and v'' by our assumption. However, different numbers of isolated vertices are added to $A_{p-X,h}$ and $A_{p-Y,h}$. Thus, $v' \neq v''$, a contradiction.

A similar argument holds for the case of two type II augmentations. □

Theorem 5.1 above together with Lemma 5.5 gives the following orderly algorithm (See Figure 5.5 and Figure 5.3). If we simply wish to have a particular list $L_{p,h}$, we generate the lists level by level. The symbol $\Leftarrow_I$ or $\Leftarrow_{II}$ indicates augmentation of type I or type II, respectively, of the graphs in the list on the right hand side. If the digraph is canonical after the augmentation, store it in the list on the left hand side. In the algorithm, a canonicity test after the augmentation is assumed.

For example, suppose we want to generate $L_{4,3}$. See Figure 5.5. Initially we get $L_{1,1}$ and $L_{2,1}$ each of which contains one and two isolated vertices respectively. This can be done by the first *for loop* (initialization step). In the second stage, we get $L_{2,2}$ from $L_{1,1}$ by type I augmentation, and then, $L_{3,2}$ from $L_{2,1}$ and $L_{1,1}$ by type I augmentation, and from $L_{2,2}$ by type II augmentation. Similarly, in the third stage, we get $L_{3,3}$ from $L_{2,2}$ by type I augmentation, and then, $L_{4,3}$ from $L_{3,2}$ and $L_{2,2}$ by type I augmentation, and from

$L_{3,3}$ by type II augmentation. After obtaining $L_{4,3}$, the algorithm terminates. The second and third stage can be done by the nested *for loops* in the algorithm (general steps).

Note that we may also generate a set of lists vertex by vertex.

Algorithm ACY

/*generate a list of acyclic digraphs with p vertices and h levels*/

for $k = 1$ **to** $p - h + 1$

  Set $L_{k,1}$={k isolated vertices at level 1}

endfor

for $j = 2$ **to** h

  for $i = j$ **to** $j + p - h$

    for $k = i - 1$ **to** $j - 1$

      $L_{i,j} \Leftarrow_I L_{k,j-1}$+{$i - 1 - k$ isolated vertices at level 1}

    endfor

    if $j > i$ **then**

      for $k = i - 1$ **to** j

        $L_{i,j} \Leftarrow_{II} L_{k,j}$+{$i - 1 - k$ isolated vertices at level 1}

      endfor

    endif

  endfor

endfor

Figure 5.5: An orderly algorithm for generating acyclic digraphs.

5.5 Acyclic Transitive Digraphs

A list of acyclic transitive digraphs (equivalently, posets) can be extracted from the list of acyclic digraphs. Alternatively, when we augment an acyclic digraph, we check transitivity so that only acyclic transitive digraphs are kept in the list. This can be done as follows. Note that Lemma 5.5 can be strengthened to the following statement, proved similarly.

LEMMA **5.6** *A canonical acyclic transitive digraph can be obtained by augmentation from some canonical acyclic transitive digraph. Moreover, canonical acyclic transitive digraphs from different lists cannot be augmented to isomorphic canonical acyclic transitive digraphs.*

When we augment an acyclic digraph, we add a new column in **M**. The presence of 1's in the bth row in the augmenting column identifies the vertex b from which an arc comes to the augmenting vertex.

Find a column corresponding to this vertex b. Then, in the augmenting column, check if a 1 is present in the same row as the row in which a 1 is present in the column corresponding to the vertex b. This is equivalent to checking the transitivity of the augmented digraph. This takes $O(p^2)$ time, where p is the number of vertices in the acyclic digraph, since all of the arcs incident with each vertex b are scanned.

5.6 Finite Lattices and Semilattices

For finite lattices and semilattices, we have the following augmentation Lemma 5.7, the proof of which is again similar to the proof of Lemma 5.5 and left to the reader.

LEMMA **5.7** *A canonical finite lattice can be obtained by augmentation from some canonical finite lattice. Moreover, canonical finite lattices from different lists cannot be augmented to isomorphic canonical finite lattices.*

All finite lattices can be obtained by first generating all semi-lattices having a single source. From the property of the lattice, we immediately obtain the following well known fact (See Stanley, [52]).

PROPOSITION **5.3** *For a given semi-lattice graph with a single source if we add a new vertex, called the sink, forming a new last layer, and connect any vertex to the sink by arcs, then the resultant graph is a lattice. Furthermore, every finite lattice can be uniquely obtained in this fashion.*

Proof. Suppose the first part of the proposition does not hold. Then there exist four vertices a, b, c and d not satisfying the unique bound condition. By assumption the graph was a semi-lattice before adding the new vertex. Thus, the new vertex, is either c or d. Without loss of generality, let c be the sink. But by our assumption, c must be connected from the rest of vertices, in particular, from d, a contradiction.

From the existence of a unique greatest lower bound for any two vertices in the digraph of a lattice, the rest follows. □

Thus, we find a natural one-to-one correspondence between the semi-lattice of $n-1$ vertices and $l-1$ levels, and the lattice of n vertices and l levels. Hence, in order to count and generate lattices, it is sufficient to generate only semi-lattices. We can obtain a desired lattice by adding a sink node and connecting it to the rest of the nodes, by virtue of Proposition 5.3. The semi-lattices are obtained by the same algorithm as that for acyclic transitive digraphs with the following modification. Whenever we add an augmenting vertex, we check if the new vertex and the other vertices not connected with the new vertex have two incomparable ancestor vertices; that is, we check the unique bound condition, as well as the transitivity. We accept the augmenting vertex if it does not create two incomparable ancestors and it does preserve transitivity.

Since we have already described the transitivity test, we now illustrate how to check the unique bound condition. The definition of the unique bound condition suggests the following simple algorithm. For all incomparable pairs of nodes (c,d), check if their exist two nodes (a,b) such that a and b are not comparable to each other and $a \geq c,d$ and $b \geq c,d$. However, finding such a pair (a,b) is not sufficient to determine whether or not a given acyclic transitive digraph is a lattice. Such a case is illustrated by Figure 5.6.

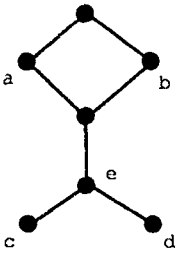

Figure 5.6: An example of checking the unique bound condition for a poset.

This poset has nodes a, b, c and d not satisfying the unique bound condition, but c and d have the least upper bound e. We note that $a \geq e$ and $b \geq e$ because of transitivity. Thus, whenever we find a common parent a of c and d, we must check whether or not a is a parent of any other parents (for example, e in Figure 5.6) of c and d. We maintain in a stack all the common parents of the augmenting node and the nodes not comparable with the augmenting node. Suppose a new parent node is found. We check whether or not the new parent node is an ancestor of any node in the stack. If such a relationship exists, we put the new parent on the stack; otherwise, the poset is not a lattice. In this way, we can test whether or not the given poset has four nodes not satisfying the unique bound condition.

The unique bound condition can be tested as follows. We find, first, a pair consisting of an augmenting node c and a node d not adjacent to c. For this pair, we check if there are no compatible upper bound nodes a and b.

Recall now the matrix representation of an acyclic digraph. We notice that, in an augmenting column, if row i has a "0" entry, then node i is not connected to the augmenting node. Accordingly, the ith column in the adjacent matrix corresponding to the node i and the augmenting column are compared row by row.

If there is a "1" in the jth row of both columns, node j is found to be a common parent of node i and the augmenting node. Put the node j in the stack, provided nothing is in the stack now. Suppose we find another such j' and we check whether or not j and j' are comparable. If they are not comparable, we have found four nodes not satisfying the unique bound condition and conclude that the matrix is not a lattice. Otherwise, we put

j' on the stack. We repeat the same process for all parents of node i and the augmenting node.

5.7 Finite Topologies

Finally, we consider finite topologies. A finite topology is equivalent to a transitive digraph. A transitive digraph is, in general, not acyclic. However, we will transform a transitive digraph to an acyclic transitive digraph with some labeling so that our orderly algorithm for generating acyclic digraphs can be applied.

A labeled transitive digraph can be obtained from a labeled acyclic transitive digraph. The technique of this transformation is shown in Harary et al. [12]. For the case of an unlabeled digraph, we adopt a similar but slightly different method.

Recall the properties of a transitive digraph described in Lemma 2.1 and Lemma 2.2. For a given unlabeled acyclic transitive digraph, we assign to each of the m vertices in the acyclic transitive digraph the numbers in the m-part composition of n without empty parts, $n \geq m$, and interpret each number as the size of each strong component in the transitive digraph. Then, we observe that there exists a one-to-one correspondence between unlabeled transitive digraphs with n vertices and m strong components and unlabeled acyclic transitive digraphs with m vertices labeled in such a manner. For example, see Figure 5.7.

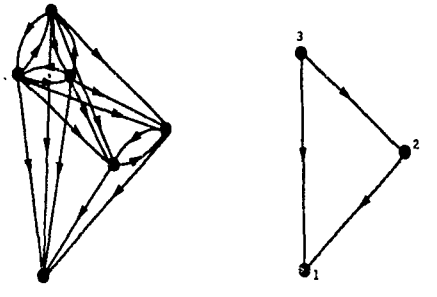

Figure 5.7: An example of a transdigraph and an equivalent representation by an acyclic transitive digraph with labeling

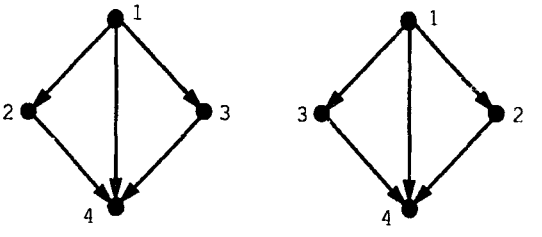

Figure 5.8: An assignment of two decompositions of the number 10 to the vertices such that the resulting digraphs are isomorphic

Thus, although a transitive digraph in general is not acyclic, we can represent a transitive digraph by assigning integer labels to its condensation. However, two distinct integer assignments to a given representation of a acyclic transitive digraph may yield representations of isomorphic transitive digraphs, unlike the labeled case; therefore, a canonicity test must be made on all automorphisms of the underlying acyclic transitive digraph. This is exemplified in figure 5.8.

Thus, we conduct a second canonicity test on the unlabeled acyclic transitive digraph labeled with the numbers mentioned above, going through a set of automorphisms which can be found by the first canonicity test on the acyclic transitive digraph. Note that the number of automorphisms is relatively small in acyclic digraphs (see Bender et al.[2]). This shows that a canonical transdigraph will result, by augmentation, from some canonical transdigraph.

Now if $n = m$, then the graphs obtained by the above labeling are in one-to-one correspondence with the acyclic transitive digraphs since all the vertices are merely labeled by 1. In particular, if there are $m(= n)$ isolated vertices in an acyclic transitive digraph, we obtain n isolated vertices as a transitive digraph.

If $n > m$, at least one label is greater than 1; that is, at least one strong component is of size greater than 1. In particular, if there are m isolated vertices, the number of distinct labelings of those vertices is equivalent to the number of partitions of n into m nonempty slots. This number can be obtained from the number of partitions of n into m nonempty slots where $n \geq m$ by subtracting the case that $n = m$. However, it is easily seen that

there is only one case in which the n is distributed into $m(=n)$ nonempty slots and this case was already treated above.

The code of a transitive digraph is represented by the code of the acyclic transitive digraph a, and its labeling b, where the elements of b are an integer sequence derived by the partition of the integer n. The lexical order of a pair of vectors (a, b) is easily extended from that of a; that is, (a, b) is lexically larger than (a', b') if a is lexically larger than a', or $a = a'$ and b is lexically larger than b'.

Note that the acyclic transitive digraphs of size n are isomorphic to the transitive digraphs of size n whose strong component all have size of exactly 1. Thus, the number of transitive digraphs of size n is the sum of the number of acyclic transitive digraphs of size n and the number of transitive digraphs of size n with at least one of whose strong components is greater than 1. If we know the number of acyclic transitive digraphs. this saves time in computing a census of the digraphs.

We have the following observation regarding a relation between the number of unlabeled transitive digraphs and that of unlabeled acyclic transitive digraphs. Let T_n be the number of an unlabeled transitive digraph on n vertices and G_n be that of unlabeled acyclic transitive digraphs on m vertices. Then we have the following inequality, where $P(n, m)$ is the number of partitions of n into m nonempty slots.

$$T_n \leq \sum_{m=1}^{n} P(n, m) \times G_m \tag{5.1}$$

This inequality can be compared to the formula below for the number of labeled transitive digraphs in Theorem 1 in [12], where T'_n is the labeled transitive digraphs on n vertices, G'_m is the number of labeled acyclic transitive digraphs and $P(n, m)$ has the same meaning as above.

$$T'_n = \sum_{m=1}^{n} P(n, m) \times G'_m \tag{5.2}$$

5.8 The Canonicity Test

In this section we show how to conduct a canonicity test for our representation of acyclic digraphs. The obvious way to do a canonicity test is to first permute all the vertices according to some permutation group of degree p (in this case, the permutation group is the *symmetric group* of degree p), and then compare the original matrix and the matrix after each permutation of the vertices. If the original matrix is lexically greater than any other matrices, the matrix is canonical; otherwise it is not. A more clever way is to take a permutation group as a Cartesian product of symmetric groups of degree l_i where $p = l_1 + l_2 + \ldots + l_h$ The second method is faster than the first method. However, we have to further improve the time complexity of the canonicity test. Such an improvement leads to the production of much larger graphs. We will now describe an improved method of canonicity test.

In the case of an acyclic digraph with height h, the digraph is represented by a list of submatrices $M = (M_1, M_2, .., M_{h-1})$. We first identify among all permutations of the sets of vertices X_1 and X_2 the permutation that maximizes submatrix M_1 induced by the sets X_1 and X_2 since M_1 contains more significant digits than any of the other matrices. This can be accomplished by permuting the rows corresponding to the elements of X_1 and sorting the columns corresponding to the elements of X_2. From the sorted columns of X_2 we can obtain all permutations maximizing M_1 as follows. In the sorted M_1, we identify the columns having the same values. These will be adjacent. Suppose we have l distinct column vector value. Let d_i, $i = 1 \ldots l$ be the number of columns having the ith value.

For a given fixed ordering of X_1, we find all possible permutations of X_2 that fix each group of columns with the same column value. There are $d_1! \cdot d_2! \cdot \ldots \cdot d_l!$ such permutations that maximize M_1.

Now, we want to find all the permutations on X_3 which maximize the submatrix M_2 induced by the sets $(X_1 \cup X_2)$ and X_3 under a given fixed permutation of $(X_1 \cup X_2)$. In general, we have to find all the permutations on X_j which maximize the submatrix M_{j-1} given a fixed ordering of $(X_1 \cup X_2 \cup \ldots \cup X_{j-1})$ unless $j = k$. The canonicity test can thus be organized as a search tree, the node of which correspond to permutations of the

vertex set of the digraph. This search tree can be traversed in a Breadth First Search manner if we, at each sorting step, obtain all the permutations maximizing the current submatrix and put them in a list called *Queue*. The search tree can be traversed in Depth First Search manner in which we find the next legitimate permutation from the previous permutation and put it on a stack. If a matrix is canonical, we have to explore all the nodes in the search tree in either method.

A canonicity test algorithm of the BFS form is summarized as follows. The algorithm takes matrix M and h, the number of levels, and returns "yes" if the matrix is canonical, otherwise "no". Note that the list *Queue* contains information consisting of a permutation and a level index.

The computational complexity of the canonicity test may be measured by its best and worst cases. First, we note that for a fixed permutation of $X_1 \cup X_2 \cup \ldots \cup X_{l-1}$, it is sufficient to sort the columns of M_h with respect to X_l for $l = 2 \ldots h$. Such a sorting can be done in $O(m_l log(m_l))$ time. Let $f(n)$ express the total time needed to sort all the columns at each level.

The best case occurs if each of $M_1, M_2, .., M_{h-2}$ contains only distinct column values. For each permutation of X_1, the outer *for* loop in the above algorithm dominates its overall complexity, which is $|X_1|!=m_1!$ time. Hence, the best case complexity of the canonicity test is $O(f(n) \cdot m_1!)$.

In the worst case, all of the columns in each M_i have identical values. Then all permutations on X_i maximize M_{i-1} for a fixed permutation on $X_2, X_3, \ldots, X_{i-1}$. The worst case complexity of the canonicity test is in this case $O(f(n) \cdot m_1! \cdot m_2! \ldots . m_{h-1}!)$.

In the case of a finite lattice, since there is only one source, that is, $l_1 = 1$, this can be further reduced to $O(l_2! \ldots ! l_{h-1}!)$. Note that despite this worst case time complexity, the time complexity amortized over all acyclic digraphs is much less than this, since the number of automorphisms of an acyclic digraph is on average a very small fixed number as noted in Bender et al., [2].

The actual implementation is as follows. Our basic strategy is backtracking by depth first search in the search space that is defined by the permutations found by the algorithm.

The information associated with each permutation in a stack is shown in Figure 5.10. The current permutation on a matrix is expressed as the ordered set of those permutations currently in the stack.

The parameters in a permutation vector are as follows: *loc* is the lowest index of columns with the same column value; *size* is the number of columns with the same column value; *limit* is (*size*)!; *order* is the order of a permutation among all the distinct permutations within the same layer; *current* is the rank of the current permutation in the Johnson-Trotter algorithm used as a subroutine to generate the (*size*)! permutations of the group of columns currently being considered; *level* is the index of the layer where the permutation resides.

Initially, the permutations of the first layer are stacked with $loc = 1$, *size*=size of the first layer, $limit = size!$, $order = 1$, $current = 1$, and $level = 1$. For this initial permutation of the first layer, we will find the initial permutations at each level. We use two pointers for the stack: *g_ptr* and *s_ptr*. The pointer *g_ptr* is used for pointing to the permutation of highest column index within the same layer, and *s_ptr* points to the current permutation. If there is only one permutation within a layer, *g_ptr* and *s_ptr* are the same in that layer. If there is more than one permutation within the same layer. *s_ptr* may be less than *g_ptr*.

The permutation defined by the Cartesian product of those permutations on the same layer is treated as one permutation associated with the underlying layer. Hence, when all the values of *current* of those permutations become *limit*, the permutation associated with the layer is completed.

Pointers *g_ptr* and *s_ptr* point to the last permutation of the highest column index after the initial stack setting described above. We get the information on the current rank pointed to by this *s_ptr* from the stack. We permute the vertices in the graph according to this current rank. Then, we compare the original matrix and the resultant matrix after permuting the vertices. At the very beginning, both are equal. Hence, the number of automorphisms is merely incremented by 1.

If *current* (the current rank) is not reached at *limit*, we will construct the permutation

whose rank is *current* in the Johnson-Trotter algorithm and reset *s_ptr* to *g_ptr*. Note that if the vertices to be permuted are on the ith level layer, only the column values of submatrices of the jth layer, $j > i$ are affected after the permutation. See Figure 5.1. Then, we compare the resultant matrix after permuting the vertices and the original matrix. If the original is still lexically larger than the newly permuted matrix, we go back to the beginning, meaning that we have to try another permutation of vertices to see whether or not the matrix is canonical; otherwise, we output that the matrix is not canonical and terminate the canonicity test (a permutation on vertices expressed by a set of permutation vectors in the stack makes the matrix larger than the original).

If *current* is reached at *limit*, two cases arise. The first case is that the *order* of the current permutation is greater than one, whether or not there is only one permutation within a layer. In this case, we will do the last permutation and decrement the pointers *g_ptr* and *s_ptr*. If *g_ptr* becomes zero, we will terminate the canonicity test outputting that the graph is canonical (since we did not encounter any permutation which makes the matrix lexically larger than the original). In the second case, there is more than one permutation and the *order* of the current permutation is greater than one, and we decrement *s_ptr* only while maintaining the value of *g_ptr*. The reason why we decrement only *s_ptr* is that a set of permutations of different column value within the same layer act as one permutation formed by the Cartesian product of these permutations and that any permutation affects only the column of a higher indexed layer. After conducting the permutation of vertices above and resetting the pointers, we will find an initial permutation for the level greater than the one pointed to by *g_ptr* (given the current permutation on the vertices up to the level of the permutation vector pointed by *g_ptr*).

g_ptr and *s_ptr* point to the last permutation of the highest column index after the initial stack setting described above. We get the information on the current rank pointed to by this *s_ptr* from the stack. We permute the matrix according to this current rank. Then, we compare the original matrix and the permuted matrix. At the very beginning, both are equal. Hence, the number of automorphisms is merely incremented by 1.

If *current* (the current rank) is not reached at *limit*, we will construct the permutation

whose rank is *current* in the Johnson-Trotter algorithm and reset *s_ptr* to *g_ptr*. Note that if the vertices to be permuted are on the ith level layer, only the column values of submatrices of the jth layer, $j > i$ are affected after the permutation. See Figure 5.1. Then, we compare the newly permuted matrix and the original. If the original is still lexically larger than the newly permuted matrix, we go back to the beginning, meaning that we have to try another permutation of vertices to see whether or not the matrix is canonical; otherwise, we output that the matrix is not canonical and terminate the canonicity test (a permutation on vertices expressed by a set of permutation vectors in the stack makes the matrix larger than the original).

If *current* is reached at *limit*, two cases arise. The first case is that the *order* of the current permutation is greater than one, whether or not there is only one permutation within a layer. In this case, we will do the last permutation and decrement the pointers *g_ptr* and *s_ptr*. If *g_ptr* becomes zero, we will terminate the canonicity test outputting that the graph is canonical (since we did not encounter any permutation which makes the matrix lexically larger than the original). In the second case, there is more than one permutation and the *order* of the current permutation is greater than one, and we decrement *s_ptr* only while maintaining the value of *g_ptr*. The reason why we decrement only *s_ptr* is that a set of permutations of different column value within the same layer act as one permutation formed by the Cartesian product of these permutations and that any permutation affects only the column of a higher indexed layer. After conducting the permutation above and resetting the pointers, we will find an initial permutation for the level greater than the one pointed to by *g_ptr* (given the current permutation on the vertices up to the level of the permutation vector pointed by *g_ptr*). Then, we go back to the beginning.

Although the worst case time complexity of this search is factorial, on average over all acyclic digraphs, the time complexity to do such a search is some small constant as noted before. As a result, we could generate digraphs with relatively large number of vertices (For example, the number of elements of the largest finite lattices is 13. For general graphs, Read generated graphs with 10 vertices and the number of distinct graphs with

10 vertices is 12,005,168 [5].)

In the next chapter, we consider the generation of $k-$colored graphs. These admit a representation essentially identical to the representation we have used in this chapter for acyclic digrpahs. Thus the above canonicity test can also be used for $k-$colored graphs.

Canonicity(M,h) /* $M = (M_1, M_2, ..., M_{h-1})$ */

1. $MTEMP = M$ /* reserve the original matrix */
2. **for** each permutation P_r on X_1
   - 2.1 $Q = (Pr, 1)$
   - 2.2 **while** $Q \neq \phi$
     - 2.2.1 $(P, i) = POP(Q)$
     - 2.2.2 **if** $i < h - 1$
       - **then** 1. permute X_i by P
         - 2. sort X_{i+1} such that M_i is max
         - 3. find all permutation $(P_1, P_2, .., P_h)$ on X_{i+1} such that M_i is max from the one found by 2
         - 4. $Q = \{(P_1, i+1), (P_2, i+1), \ldots, (P_h, i+1)\}$
       - **else**
         - 1. sort X_h so that M_h is max
         - 2. **if** $MTEMP < M$ **then** return("no")
       - **endif**
     - **endwhile**
   - **endfor**
3. **return**("yes")

Figure 5.9: The canonicity test

loc	*size*	*limit*	*order*	*current*	*level*

Figure 5.10: A permutation vector used for the canonicity test

Chapter 6

K-Colored Graph Generation

6.1 Introduction

In this chapter, we consider generation of bicolored graphs, generation of bicolorable graphs, and generation of $k-$colored graphs. These algorithms are fundamentally equivalent to the orderly algorithm for generating acyclic digraphs in Chapter 5, replacing the term "level" by "color class". Moreover, we also give some properties of the canonical matrix. Orderly algorithms in the previous chapter and this chapter are strictly speaking *weak*-orderly algorithms. In the last section, we give a simple technique to convert these weak-orderly algorithms to orderly algorithms in Read's sense, i.e., algorithms that produce the graphs in lexical order.

6.2 Properties of a Canonic Vector

A bicolored graph $G = (V, E)$, $X_1 \cup X_2 = V$, where X_1 and X_2 represent the color classes, and $X_1 \cap X_2 = \emptyset$ can be represented by a rectangular matrix in which the rows correspond to the vertices x_1 in X_1, and the columns correspond to the vertices x_2 in X_2. If the matrix is canonical, where canonicity of a bicolored graph is defined by the lexical order of its corresponding matrix, the associated vector forms a non-increasing sequence of integers. This can be easily proved as follows.

LEMMA **6.1** *The canonical vector of a bicolored graph always forms a non-increasing integer sequence.*

PROOF: Suppose the assertion does not hold. Then, for the canonical vector of some bicolored graph, there exists a pair of columns c_1 and c_2 such that $c_1 < c_2$ but the value of c_1 is greater than the value of c_2. We can then interchange columns c_1 and c_2 and get a lexically larger integer sequence. Therefore, the vector was not canonical, a contradiction. □

Before moving to $k-$colored graph generation, we show one property of the canonical matrix of a connected graph. The technique used in this lemma appears again in the proof for the case of connected bicolored graphs. We also use in this chapter the upper triangular matrix for representing general graphs as in Chapter 5. A particular property of the canonical matrix of general graphs was given in Lemma 5.2. From a reflection of our common sense that a connected graph can be extended from the next smallest graph recursively, a consequence of Lemma 5.2 is the following. By *augmentation* of a graph we mean that a vertex is added to a graph and in particular, in the matrix representation of a graph, an augmenting vertex, or equivalently, an augmenting column, is placed at the far right of the matrix.

LEMMA **6.2** *If $M(n)$ is the canonical matrix for a connected graph G on n vertices, then it must be obtained, by augmentation, from at least one canonical matrix $M(n-1)$ of a connected graph G' on $n-1$ vertices.*

PROOF: Suppose we have a graph $M(n-1)$ with two or more connected components. We may assume that the first component is not an isolated vertex, otherwise $M(n-1)$ is not canonical. Then we assume that a set of vertices $\{1,2,\ldots,i\}$, $1 < i < n-1$ constitutes the first component, that is, a set of column vectors $\{v_2, v_3, ..v_i\}$ of a subtriangular matrix constitutes a first component. Then, for j, $i < j \leq n-1$, we have prefix $v_j^{i-1} = 0$. Similarly, each other component forms an off-diagonal subtriangular matrix.

Suppose that after augmenting $M(n-1)$, we get an $M(n)$ whose corresponding graph is connected. Then, at least one "1" must appear in the elements of v_n corresponding to each component. But the "1" corresponding to the first component violates canonicity by Lemma 5.2.

Since the above argument holds even in the case that some of the components other than the first consist of a single vertex, the lemma follows. □

Note that a similar result holds for a digraph, a multigraph or a tournament graph.

6.3 K-Colored Graph Generation

To represent a k-colored graphs we consider the decomposition of p vertices into k ordered disjoint sets such that $p = m_1 + m_2 + ... + m_k$ where m_i represents the number of vertices colored with the ith color as in Read[49].

Let X_i denote the set of vertices of the ith color, with $|X_i|=m_i$. Let $\mathcal{M}$ be the adjacency matrix and let M_i be the submatrix induced from $\mathcal{M}$ by rows corresponding to the vertices in $X_1, X_2, \ldots, X_i$ and columns corresponding to the vertices in X_{i+1}. Then the list of submatrices $M = \{M_1, M_2, \ldots, M_{k-1}\}$ represents a k-colored graph. Let v_i denote the set of column vectors in M_i. Thus, the associated vector of M is represented by $(v_1, v_2, \ldots, v_{k-1})$. We immediately find that this $k-$colored graph representation is exactly same as that for an acyclic digraph of height k. In fact, any $k-$colored graph is isomorphic to an acyclic digraph of height k by choosing nodes in one color class as the first layer and the other as the last layer such that there exists at least a pair of vertices (x, y) where x is in the first layer and y is in the last layer, and the maximum distance is $k-1$, after appropriate orientation of edges in the graph, For any acyclic digraph of height k can be colored in k colors and the graph is isomorphic to a $k-$colored graph ignoring the orientation of each arc. Thus, the canonical matrix of a $k-$colored graph is defined analogously to that for an acyclic digraph by replacing the height k by the number of color classes k. See Section 5.2 concerning the representation of acyclic digraphs. In particular,

the *associated vector* of a matrix, the *canonicity* of a matrix, and the *code* of a graph are all analogously defined. Similarly, the augmentation of $k-$colored graphs is defined analogously to that for acyclic digraphs. Hence, we have two types of augmentation. A *type I augmentation* is such that a vertex to be added forms a new color class, and is connected by at least one arc from a vertex in the old last color class, and the number of color classes of the graph is increased by 1. A *type II augmentation* is such that a vertex is added to the last color class of a canonical $k-$colored graphs together with at least one edge from the $k-1$st color class to this vertex. In a type II augmentation, the number of color classes of the graph remains the same. In the matrix representation of a $k-$colored graph, an augmenting vertex, or equivalently, an augmenting column, whether it is type I or type II, is placed at the far right of the matrix.

Let $L_{M_1,M_2,...,M_{k-1}}$ denote the list of canonical matrices $M = \{M_1, M_2, \ldots, M_{k-1}\}$.

Some consequences of Lemma 6.1 applied to k-colored graph representation are stated in Lemmas 6.3 and 6.4 which follow. The proofs of these are same as for Lemmas 5.3 and 5.4 by identifying the height h as the number of color classes k. Thus, the proofs of these lemmas are omitted.

LEMMA **6.3** *If a matrix $K = \{M_1, M_2, .., M_{k-1}\}$ for a k-colored graph is canonical, then in the associated vector $(v_1, v_2, ..., v_{k-1})$, each integer sequence of v_i is monotone non-increasing.*

□

Regarding the isolated vertices, we obtain the following.

LEMMA **6.4** *If a k-colored graph is canonical, isolated vertices are present in only the first color class of vertices X_1. Further, those isolated vertices have the largest possible labeling in X_1.*

□

By the k-colored graph representation and Lemma 6.4, we obtain the following result.

LEMMA **6.5** *Let $K(m_1, m_2, .., m_k)$ be a canonical matrix for a k-colored graph G, where m_i represents the number of vertices colored by the i th color. Then any $K(m_1, m_2, ..., m_k)$ must be obtained by augmentation from at least one of $K(m_1', m_2, ..., m_k - 1)$'s with $m_1 - m_1'$ isolated vertices in the first color if $m_k > 1$ or one of $K(m_1', m_2, ..., m_{k-1})$'s with $m_1 - m_1'$ isolated vertices in the first color if $m_k = 1$, where an augmenting vertex is placed on the kth color and connected to at least one vertex in the $(k-1)$ st color and to all the isolated vertices in the first color if they exist. $K(m_1)$ implies that there are m_1 isolated vertices of the first color.*

PROOF: A proof is similar to that for Theorem 5.1 since the representation of a k-colored graph is the same as that of an acyclic digraph. □

In the case of bicolored graphs, we restrict our attention to connected graphs. Note that up to exchange of color classes, there exists a one-to-one correspondence between connected bicolored graphs and connected bicolorable graphs. We obtain the following Theorem 6.1,

THEOREM **6.1** *If $M(m, n)$ is a canonical matrix for a connected bicolored graph G, then it can be obtained by augmentation by either from a canonical $M(m, n-1)$ or from a canonical $M(i, n-1)$ with $m - i$ isolated vertices in X_1, that is, a matrix $M(i, n-1)$ with $m - i$ all-zero rows added to the bottom.*

PROOF: Suppose we have a graph $M(m, n-1)$ with two components. The rows and columns will be divided into two sets so that one of the components occupies the upper-left submatrix and the rest the lower-right submatrix. Otherwise the matrix would not be canonical. If neither component is an isolated vertex, then each of the submatrices must contain at least one 1.

Suppose that after augmenting $M(m, n-1)$, we get an $M(m, n)$ whose corresponding graph is connected. Then, the last column in $M(m, n)$ must have at least one 1 in each

of the sets of rows corresponding to the components. But the 1 corresponding to the first component violates the canonicity by Lemma 6.3. Hence it cannot be the case that both components are nontrivial.

It is readily verified that we reach the same conclusion if $M(m, n-1)$ has 3 or more components. Hence only one component of $M(m, n-1)$ can be nontrivial. The theorem follows. □

The proof technique in Theorem 6.1 seems not to be applicable to the case of more than two colors. Thus, all connected canonical bicolorable graphs can be obtained by augmentation from connected canonical bicolorable graphs of smaller size with the possible addition of a number of isolated vertices. In other words, $L_{m,n}$ may be generated by effectively augmenting elements in $L_{l,n-1}$, $l = 1, .., m$. Thus, assuming that each $L_{l,n-1}$ is lexically ordered, the generated lists are individually in lexical order, but not when taken altogether. Although sorting the resultant $L_{m,n}$ in lexical order can be done polynomially in the cardinality of $L_{m,n}$ and size n, it is unnecessary to sort them if we need only a list, since the canonicity test depends only on the matrix itself but not on the lexical ordering of the canonical graphs within the list.

By this theorem, we may generate effectively only the connected bicolorable graphs. The algorithm for generating k-colored graphs is presented in the next section.

6.4 Generating Bicolored and K-Colored Graphs

Let $L_{m_1,m_2,..,m_k}$ be a list of canonical matrices $K(m_1, m_2, .., m_k)$ of k-colored graphs, where m_i denotes the sizes of the color classes. First we produce a catalog of $(k-1)$-colored graphs and then it is from them that we obtain k-colored graphs. We assume that an augmenting column always exists at the far right of the matrix K; that is, an augmenting vertex is added to only the last color class X_k. By placing an augmenting column to the far right in the matrix, an implicit order for generating each list is introduced, and the orderly algorithm works properly.

We consider initially the case $k = 2$, i.e., a bicolored graph. We show first the following lemma.

LEMMA 6.6 *Any canonical matrix M(m,n) of a bicolored graph can be obtained by augmentation from a canonical matrix M(m,n−1).*

PROOF: Suppose the last column of $M(m,n)$ is deleted and the resultant matrix $M'(m,n-1)$ is not canonical, where the corresponding graph for M' is a graph G'. Then there exists a permutation relabeling vertices of G' which makes the corresponding matrix $M'(m,n-1)$ canonical.

In $M(m,n)$, apply the same permutation to the submatrix consisting of the first $n-1$ columns (which may be accomplished by permuting the rows and sorting the columns). The resultant matrix is the canonical matrix for G but its corresponding vector is larger than the one before. This contradicts the assumption that $M(m,n)$ for G is canonical. □

The above yields an orderly algorithm for bicolored graphs shown in Figure 6.1.

By Theorem 6.1, as we noted before, if we want to generate only the connected bicolored graphs, we may do so with Algorithm 2. We first obtain the following additional lemma.

LEMMA 6.7 . *Two distinct canonical matrices $M'(m',n-1)$ and $M''(m'',n-1)$, $m' \leq m''$, together with $m-m'$ and $m-m''$ isolated vertices, where $m > m'$ and $m > m''$, cannot be augmented to the same canonical $M(m,n)$.*

PROOF: Suppose $M(m,n)$ can be obtained by augmentation from $M'(m',n-1)$ and $M''(m'',n-1)$, $m' = m''$, both of which are in the same list $L_{m',n-1}$. As the list consists of non isomorphic graphs, their vector representations v' and v'' must be different; consequently, they cannot be augmented to the same matrix.

Suppose $M(m,n)$ can be obtained by augmentation from $M'(m',n-1)$ and $M''(m'',n-1)$ where $m' \neq m''$. Let v, v' and v'' denote the corresponding canonical vectors respectively. If we delete the last element in v, the resultant vector must be v' and v'' by assumption. However, as the number of isolated vertices in $M(m',n-1)$ and $M(m'',n-1)$

Algorithm 1 /*This algorithm finds all bicolored graphs on up to $m+n$ vertices, $m \neq n$.*/

1. Create a list $L_{m,1}$ consisting of canonical matrices $M(1,1)$, $M(2,1),..,M(m,1)$.
2. **For** $i = 2$ **to** n **do**
   - 2.1 **For** each $M(m,i-1) \in L_{m,i-1}$, augment it
     - **if** the augmented matrix is canonical
       - **then** add the matrix to a list $L_{m,i}$
       - **else** discard it.
     - **endif**
   - **endfor**

endfor

Figure 6.1: The algorithm for generating bicolored graphs

is different, v' cannot be equal to v'', a contradiction. □

The above Theorem6.1 together with Lemma 6.1, Lemma 6.6, and Lemma 6.7 give the following orderly algorithm described in Figure 6.4. Let $L_{m,n}$ be a not necessarily ordered list of canonical matrices $M(m,n)$ for connected bicolored graphs.

The validity of Algorithm 2 as an orderly algorithm follows from Theorem 6.1 together with Lemma 6.6 and Lemma 6.7.

Lemma 6.7 can be generalized to k-colored graphs as follows.

LEMMA **6.8** *Two distinct canonical matrices* $K'(m'_1, m_2, ..., m_k - 1)$ *and* $K''(m''_1, m_2, ..., m_k - 1)$, $m''_1 \leq m'_1$, *together with* $m_1 - m'_1$ *and* $m_1 - m''_1$ *isolated vertices respectively,* $m'_1 \leq m_1$ *and* $m''_1 \leq m_1$ *cannot be augmented to the same* $K(m_1, ..., m_k)$.

PROOF: **Case 1.** Let $m_k > 1$. if $K(m_1, ..., m_k)$ can be obtained by augmentation

Algorithm 2

/* This algorithm finds all bicolorable graphs with

up to $m + n$ vertices, where $m = |X_1|$ and $n = |X_2|, m \neq n$

1. Create the initial list.

   $L_{m,1} = \{M(m,1)\}, L_{m-1,1} = \{M(m-1,1)\}, \ldots, L_{1,1} = \{M(1,1)\}$.

2. **For** $i = 2$ **to** n **do**

   2.1 For each $M(l, i-1) \in L_{l,i-1}$ for $l = 1, ..., m-1$, add $m - l$ isolated vertices to X by adding $m - l$ all-zero rows to the bottom of $M(l, i-1)$. Call the resultant matrix $M'(l, i-1)$.

   2.2 Augment $M'(l, i-1)$ by copying the last column of it to the right of itself and subtracting 1 from the new last column, which ensures entries corresponding to isolated vertices become 1. The resultant matrix is called $M''(l, i)$.

   2.3 **Repeat** the following **until** the last column value becomes $2^{m-l} + 2^{m-l} - 1$.

   2.3.1 **If** the matrix is canonical

   then add it to $L_{m,i}$; **else** discard it.

   2.3.2 Subtract 2^{m-l} from the last column value.

   endfor

3. **For** each $M(m, i-1) \in L_{m,i-1}$ **do** /* no isolated vertices */

   3.1 Augment $M(m, n-1)$ by adding the last column of it to the right of the matrix.

   3.2 **Repeat** the following **until** the last column value of the resultant matrix becomes 1.

   3.2.1 **If** the matrix is canonical

   then add it to $L_{m,i}$; **else**, discard it.

   3.2.2 Subtract 1 from the last column value.

   endfor

Figure 6.2: The algorithm for generating connected bicolorable graphs

from $K'(m'_1, m_2, \ldots m_k - 1)$ and $K''(m''_1, m_2, \ldots, m_k - 1)$, $m'_1 = m''_1$, both of which are in the same list $L_{m'_1, m_2, \ldots, m_{k-1}}$. As the list consists of non-isomorphic graphs, their vector representations $(v_1, v_2, \ldots, v_{k-1})$ and $(v'_1, v'_2, \ldots, v'_{k-1})$ must be different; thus, they cannot be augmented to the same matrix.

Suppose $K(m_1, \ldots, m_k)$ is obtained by augmentation from $K'(m'_1, \ldots, m_k - 1)$ and $K''(m''_1, \ldots, m_k - 1)$, $m'_1 \neq m''_1$. Since $m'_1 \neq m''_1$, they belong to different lists and are not isomorphic. Let $(v_1, v_2, \ldots, v_{k-1})$, $(v'_1, v'_2, \ldots, v'_{k-1})$ and $(v''_1, v''_2, \ldots, v''_{k-1})$ denote the corresponding canonical vectors respectively. If we delete the last element in v_{k-1}, the resultant vector must be $(v'_1, v'_2, \ldots v'_{k-1})$ and $(v''_1, v''_2, \ldots, v''_{k-1})$. However, as the numbers of isolated vertices in $K'(m'_1, \ldots, m_k - 1)$ and $K''(m''_1, \ldots, m_{k-1})$ are different, $(v'_1, \ldots, v'_{k-1})$ cannot be equal to $(v''_1, \ldots, v''_{k-1})$, a contradiction.

Case 2. Let $m_k = 1$. Applying the above argument to two distinct canonical matrices $K(m_1, \ldots, m_{k-1})$ in the list $L_{m_1, \ldots, m_{k-1}}$, we obtain the same contradiction. □

An outline of the orderly algorithm for generating k-colored graphs with $m_1 > 0$, $m_2 > 0$, $m_k > 0$ vertices for each color class, respectively, is described in Figure 6.3.

Algorithm 3

/* Generate the catalog of k-colored graphs with up to m_1, m_2, $m_k > 0$ vertices in each color class */

0. Read $m_1, m_2, ..., m_k$.

1. Generate the bicolored graphs $K(l_1', l_2')$ with $l_1' \leq m_1$ and $l_2' \leq m_2$ vertices for each color class.

2. Set $l_3 = l_4 = ... = l_k = 0$.

3. Set $i = 3$.

4. **while**($i \leq k$)

4.1 **while**($l_i < m_i$)

4.1.1 **For** each graph $K(l_1', l_2, ..., l_i) \in L_i$ **do** the following, where $l_1' \leq l_1$.

Add $m_1 - l_1'$ isolated vertices.

Do an augmentation satisfying Lemma 6.3.

Do a canonicity test and

if it is canonical

then add it to the list $L_{l_1, l_2, ..., l_i+1}$

else discard it.

4.1.2 Set $l_i = l_i + 1$.

endwhile

4.2 Set $i = i + 1$.

endwhile

Figure 6.3: The algorithm for generating k-colored graphs

6.5 The Canonicity Test and the Canonic Matrix

Our representation of $k-$colored graphs is the same as for acyclic digraphs. Therefore, we can do a canonicity test as we did for acyclic digraphs. The only difference between acyclic digraphs and $k-$colored graphs is that each color class in a $k-$colored graph corresponds to a level in an acyclic digraph. Therefore, we replace the term "level" by "color" in the Algorithm Canonicity described in Figures 5.9 and 5.10, obtaining a canonicity test for $k-$colored graphs.

In Lemmas 5.2 and 6.3, we showed properties of the canonical matrix for general and $k-$colored graphs. We may immediately reject augmenting columns without a canonicity test if the augmented matrix does not satisfy the relevant properties. Thus, in particular, it is sufficient in considering a k-colored graph that we test only the matrices having an augmenting column value which is at most the value of the last column in the matrix to be augmented.

We observed through our computer experiments on the orderly algorithm for generating bicolorable graphs that some rejected augmenting columns at one stage reappeared in a canonical matrix at later stage and some did not. We obtained a sufficient condition for rejecting augmenting columns without a canonicity test.

Instead of showing the proof of this criterion for the bicolorable graph case, we prove it in terms of a graph represented in the upper triangle of the adjacency matrix $\mathcal{M}$ as we did for Lemma 5.2, which in turn implies that this condition can be applied to digraphs, tournaments and k-colored graphs as well as bicolorable graphs.

Let $M = (e_1, e_2, ..., e_n)$ be a canonical matrix and $M' = (e'_1, e'_2, ..., e'_n)$ be a matrix obtained by permuting rows and columns by permutation matrix P from the adjacency matrix M, that is, $M' = S \times M \times S$ (or simply SMS). Then, the *critical column vector index* is defined to be the smallest column vector index i such that $val(e'_i) > val(e_i)$, where $val(e)$ is the integer value of the column vector e. Suppose S and S' are permutations matrices such that SMS and $S'M'S'$ are, respectively, lexically maximal. Clearly, $SMS = S'M'S'$.

THEOREM **6.2** *If the ith augmenting column vector a_i was discarded by a canonicity test*

and the critical column vector index j associated with the permutation P for the canonicity test was less than i, then, the n th column vector, $n > i$, does not contain the discarded vector a_i as a prefix of it in the canonical matrix M for the underlying augmented graph.

PROOF: Suppose a_i reappears as the prefix of the nth column vector, $n > i$, and the matrix M is canonical, that is, $M = (e_1, e_2, \ldots, e_{i-1}, \ldots, e_{n-1}, a_i')$ is canonical, where a_i' contains a_i as a prefix. Let M' be a matrix obtained from M after exchanging the ith and nth columns as well as the ith and nth rows in M. Since a_i was discarded, there must exist a permutation matrix S_i effectively operating on the leading i columns and rows in M' such that $S_iM'S_i \succ M'$.

By the assumption the critical column vector index j is less than i and the leading $i-1$ column vectors in M and M' are the same. Hence $S_iM'S_i \succ M$. Note that the effect of the permutation S_i is that the value of the column vector $val(e_l)$, $i < l \leq n$, in M' may be affected but that this does not change the lexical order of the matrices $S_iM'S_i$, M' and M.

Since $S'M'S' \succeq S_iM'S_i$ and $SMS = S'M'S'$ by the definition of S' and S, we obtain $SMS \succ M$, contradicting the fact that M is the canonical matrix. □

If the matrix is for a bicolorable graph with color classes X_1 and X_2, that is, there are $|X_1|$ rows and $|X_2|$ columns, then we have the following Corollary 6.3 without considering prefixes. The proof is similar to the one above and omitted.

COROLLARY **6.3** *If the ith column a_i is discarded after a canonicity test and the critical column index j is less than i, then a_i cannot be in the nth column, $n > i$, in the canonical matrix M of the represented bicolorable graph.*

□

These criteria for rejecting augmenting columns without a canonicity test can be used in an implementation by associating another vector v' to each vector v representing a

graph. When we find a column vector value satisfying the condition of Theorem 6.2 during the canonicity test, we store the value in v'. Whenever we augment the vector v, we first refer the augmenting column value to v', which can be done in constant time. If the augmenting column vector has one entry in v' as a prefix, we ignore the augmenting vector and no further canonicity tests are performed. This requires more storage but if the canonicity test is computationally expensive, it is worth using such information.

6.6 The Orderly Algorithm Revisited

So far, we have talked only about orderly algorithms. However, we did not enforce the linear order of configurations within a list. The algorithms presented up to this point do not produce the objects in lexical order within a list. In this section we shall describe how we can modify our algorithms to enforce the lexical order within the generated lists. The results of this section apply to the generation of acyclic transitive digraphs, finite lattices, finite topologies, $k-$colored graphs. Thus, we will prove the result in general terms.

For simplicity, we assume each graph may contain isolated vertices and may be disconnected. Let $L_{\mathbf{p}}$ denote the list of graphs with parameters $\mathbf{p}$. We know that the list $L_{\mathbf{p}}$ is obtained from the list $L_{\mathbf{p_1}}$ by some augmentation and from the list $L_{\mathbf{p_2}}$ by another augmentation. In Algorithm 2, after all the configurations of $L_{l,i-1}$ are augmented, we start augmenting the configurations of $L_{l',i-1}$ where $l < l'$. We apply the technique of 2 way merge sort to our weak-orderly algorithm to obtain an orderly algorithm in the sense of Read [41]. We assume now that the lists $L_{\mathbf{p_1}}$ and $L_{\mathbf{p_2}}$ are ordered lexically. Let G_i^1 and G_j^2 denote graphs in the lists $L_{\mathbf{p_1}}$ and $L_{\mathbf{p_2}}$, respectively. Algorithm M which selects the next graph to be augmented from the two lists is described in Figure 6.4. For simplicity, G_i^j represents a canonical vector as well as a canonical graph.

We next argue the validity of Algorithm M.

LEMMA **6.9** *For any G_i^1 and any G_j^2, if G_i^1 is not a prefix of G_j^2 and vice-versa then graphs in the list $L_{\mathbf{p}}$ produced by applying algorithm M to the list $L_{\mathbf{p_1}} \cup L_{\mathbf{p_2}}$ are lexically ordered.*

Algorithm M

1. $i = 1; j = 1$
2. **while** $i \leq m$ **and** $j \leq n$
   2.1 **if** $G_i^a \prec G_j^b$ **then** augment G_i^a; $i = i + 1$
   else augment G_j^b; $j = j + 1$
   endif
   endwhile
3. **if** $i > m$ **then while** $j < n$ do
   augment G_j^b; $j = j + 1$ **endwhile**
   else while $i < m$ do
   augment G_i^a; $i = i + 1$; **endwhile**

Figure 6.4: Algorithm M

PROOF: First, we note that any G_i^1 cannot be isomorphic to G_j^2 since their defining parameters are different and neither of them can be a prefix of the other. By the validity of the merging algorithm and the definition of lexical order, the graph to be augmented is selected according to the lexical order defined on the sets $L_{\mathbf{p_1}} \cup L_{\mathbf{p_2}}$. Let assume that G_a and G_b ($G_a \prec G_b$) are two consecutive graphs selected by Algorithm M. Note that both of G_a and G_b are either from the same list or the distinct lists. Let G'_a be a graph obtained from G_a by an augmentation. Since G'_a has G_a as a prefix, $G'_a \prec G_b$. Since the order of augmentation conforms to the lexical order(if we choose the order in such a way), the graphs in the list $L_{\mathbf{p}}$ are lexically ordered. □

The condition that neither G_i^1 nor G_j^2 is a prefix of the rest cannot be relaxed; otherwise, we may have an augmented graph $G'_a \succ G_b$. In Algorithm 2, neither $M'(l, i-1)$ nor $M'(l', i-1)$ is a prefix of the other since the number of isolated vertices of them are

different.

Now, we can replace an appropriate portion of all the orderly algorithms (which are *weak*-orderly algorithms) presented so far by Algorithm M so that canonical graphs in a list are always lexically ordered. Hence, we conclude that there exist orderly algorithms (in Read's original sense) for generating acyclic digraphs, acyclic transitive digraphs, finite lattices, finite topologies and $\kappa-$colored graphs. The algorithm can be easily extended to more than two distinct lists.

Chapter 7

Conclusions and Future Research

7.1 Conclusions

7.1.1 Ideal Generation

We have presented efficient algorithms for generating all lower ideals of a forest poset. The algorithms are generalization of Erlich's algorithm for generating the Binary Reflected Gray Code. Erlich's algorithm is a Johnson-Trotter type of algorithm [27][55]. Our algorithm is also a Johnson-Trotter type of algorithm. The first algorithm traverses elements in a current ideal in effect in preorder and the time to obtain the next ideal from the current ideal is bounded by the number of elements in the forest poset. The second algorithm eliminates the traversals and the time to produce the next ideal is $O(1)$. Finally, we have shown that our algorithm can be used as a subprocedure to find ideals in a general poset.

7.1.2 Graph Generation

We have shown that there exist orderly algorithms for generating acyclic digraphs, posets, finite lattices and finite topologies. Although a straight forward graphical representation of finite topologies is not acyclic, we have shown that they are represented by acyclic transitive digraphs with some proper labeling.

Our representations of acyclic digraphs and $k-$colored graphs are essentially identical. Thus, we can easily obtain orderly algorithms for generating $k-$colored graphs. Based on an efficient canonical representation of k-colored graphs, we have demonstrated that k-colored graph generation is computationally similar to bicolorable graph generation, because permuting vertices in a proper subgraph of a graph is sufficient to determine whether or not the matrix of the graph is canonical.

We notice that as the canonical representation mechanism and the complexity of canonicity testing suggest, any computational improvement for bicolored graph generation implies a computational improvement for $k-$colored graph generation.

7.2 Future Research

7.2.1 Ideals Generation

We found that there exists a Gray Code-like sequence for the ideals of a forest poset $\mathcal{F}$. This sequence corresponds to a Hamiltonian Path in the ideal graph of $\mathcal{F}$.

This Gray Code-like sequence in a forest poset is a generalization of the fixed size Gray Code. Is there a class of posets for which the algorithms P and L generate a Gray code of ideals? More generally, is there any other family of graphs for which their subgraphs form a Gray Code-like sequence? It is known that we can generate the spanning trees of a graph in Gray Code sequence (see Cummins [8]).

We have run Algorithm P and Algorithm L with randomly generated rooted trees from size 10 to 40. For generating rooted trees, see Nijenhuis and Wilf [38]. For all cases, the ratio of run time between Algorithm P and Algorithm L is between 2 and 4.2. This indicates that perhaps Algorithm P runs in average constant time. An analytic proof has not yet been attempted. Algorithm L is essentially obtained by applying Ehrlich's algorithm for BRGC to the useful node list of a given forest poset.

7.2.2 Graph Generation

In developing our orderly algorithms, we ought to minimize the order of the set of permuted vertices in conducting the canonicity tests. If the permutation is an automorphism of the given graph, the order of the permutation used for a canonicity test is in fact minimum. (Note that finding automorphisms is computationally as hard as isomorphism testing.) There are a few programs which test isomorphism of graphs (for example, see MacKay[36] and Kocay[30]). These programs find a canonical certificate of the given graph. If two graphs have the same canonical certificate, they are isomorphic. Incorporating these isomorphism test algorithms into an orderly algorithm would be a natural sequel to our research.

We leave one open problem. There exists an orderly algorithm to generate effectively only the 2–connected bicolorable graphs, Koda [31]. In the algorithm given here, we generate all the acyclic digraphs having no isolated vertices. We can generate only the connected acyclic digraphs by throwing away the disconnected graphs while generating all the acyclic digraphs. Can we generate only the connected acyclic digraphs without actually generating disconnected ones? The technique used for bicolorable graphs seems not to work for this problem. In general, we may ask whether or not we can generate only k-connected graphs, where $k > 1$, without generating m-connected graphs, where $m < k$. We may also ask the same question for k-colored graphs.

Bibliography

[1] A. T. Balaban, *Chemical Applications of Graph Theory*, Academic Press, London, 1976.

[2] E.A. Bender, L.B. Richmond, R.W. Robinson and N.C. Wormald, The Asymptotic Number of Acyclic Digraphs I, *Combinatorica*, Vol. 6, No. 1, 1986, 15-22.

[3] G. Birkhoff, *Lattice Theory*, 3rd ed., *American Mathematical Society Colloquium Publication*, American Mathematical Society, Providence, R.I., 1967.

[4] J. Bitner, G. Ehrlich, and E. Reingold, Efficient Generation of the Binary Reflected Gray Code and its Applications, *Communications of the ACM*, Vol. 19, No. 9, 1976, 517-521.

[5] R. D. Cameron, C. J. Colbourn, R. C. Read, and N. C. Wormald, Cataloguing the Graphs on 10 vertices, *Journal of Graph Theory*, Vol 9, 1985, 551-562.

[6] C.J.Colbourn and R.C.Read, Orderly algorithms for generating restricted classes of graphs, *Journal of Graph Theory*, Vol. 3, No. 2, 1979, 187-195.

[7] J.C. Culberson and G. Rawlins, *On Counting Posets and the Structure of the Poset of Posets*, Technical Report TR 89-7, January 30, 1989, Dept. of Computer Science, The Univ. of Alberta, Edmonton, Alberta, Canada.

[8] R. L. Cummins, Hamiltonian Circuits in Tree Graphs, *I.E.E.E. Transactions on Computers*, Vol. 13, 1966, 82-90.

[9] N. Dershowitz, A Simplified Loop-free Algorithm for Generating Permutations, *BIT*, Vol. 15, 1975, 158-164.

[10] R.J. Douglas, Bounds on the Number of Hamilton Circuits in the n-cube, *Discrete Mathematics*, Vol. 17, 1977, 142-146.

[11] G. Ehrlich, Loopless Algorithms for Generating Permutations, Combinations, and Other Combinatorial Configurations, *Journal of ACM*, Vol. 20, No. 3, 1973, 500-513.

[12] J.W. Evans, F. Harary and M.S. Lynn, On the Computer Enumeration of Finite Topologies, *Communications of the ACM*, Vol. 10, No. 5, 1967, 295-297.

[13] M. Gardner, The Curious Properties of the Gray Code and How it Can be Used to Solve Puzzles, *Scientific American*, Vol. 227, 1972, 106-109.

[14] E.N. Gilbert, Gray Codes and Paths on the n-cube, *Bell System Technical Journal*, Vol. 37, 1958, 815-826.

[15] L. Goddyn, G. Lawrence and E. Nemeth, Gray Codes with Optimized Run Lengths, *Utilitas Mathematica*, Vol. 34, 1988, 179-192.

[16] F. Gray, *Pulse Code Communication*, U.S. Patent 2632058, March 17, 1953.

[17] F. Harary, On the number of bicolored graphs, *Pacific Journal of Mathematics*, Vol. 8, 1958, 743-755.

[18] F. Harary, Unsolved problems in the enumeration of graphs, *Publ. Math. Inst. Hung. Academy of Science*, Vol. 5, 1960, 63-95.

[19] F. Harary, Combinatorial problems in graphical enumeration, *Applied Combinatorial Mathematics* (E. F. Beckenbach ed.), Wiley, New York, 1964, 185-220.

[20] F. Harary, *Graph Theory*, Addison Wesley, Reading, Massachusetts, 1967.

[21] F. Harary, Graphical enumeration problems, Chapter 1 in *Graph Theory and Theoretical Physics* (F. Harary, ed), Academic Press, London, 1967, 1-41.

[22] F. Harary, L. March and R.W. Robinson, On enumerating certain design problems in terms of bicoloured graphs with no isolates, *Environment and Planning B*, Vol. 5, 1978, 31-43.

[23] F. Harary and E.M. Palmer, *Graphical Enumeration*, Academic Press, 1973

[24] F. Harary and G. Prins, Enumeration of Bicolourable Graphs, *Canadian Journal of Mathematics* Vol. 15, 1963, 237-248.

[25] B.R. Heap, The production of graphs by computer, in R.C. Read, ed., *Graph Theory and Computing*, Academic Press, 1972, 47-62.

[26] F.G. Heath, Origins of the Binary Code, *Scientific American*, Vol. 227, No. 2, August 1972, 76-83.

[27] S.M. Johnson, Generation of permutations by adjacent transposition, *Mathematics of Computation*, Vol. 17, 1963, 282-285.

[28] J.T. Joichi and D.E. White, Gray Codes in Graphs of Subsets, *Discrete Mathematics*, Vol. 31, 1980, 29-41.

[29] J.T. Joichi, D.E. White, and S.G. Williamson, Combinatorial Gray Codes, *SIAM Journal of Computing*, Vol. 9, No. 1, 130-141, 1980.

[30] W. Kocay, Groups & Graphs, A MacIntosh Application for Graph Theory, *The Journal of Combinatorial Mathematics and Combinatorial Computing*, Vol. 3, 1988, 195-206.

[31] Y. Koda, *Divide and Conquer Orderly Algorithm for generating Bicolorable Graphs*, Research Report CORR85-12, Department of Combinatorics and Optimization, University of Waterloo, 1985.

[32] Y. Koda, Orderly algorithms for generating K-colored graphs, submitted to *the Proceedings of the 23rd Southeastern International Conference on Combinatorics, Graph Theory, and Computing*, Bocca Raton, Feb., 1992.

[33] S. Kyuno, An Inductive Algorithm to Construct Finite Lattices, *Mathematics of Computation*, Vol. 33, No. 145, 1979, 409-421.

[34] S. Kyuno and A. Ito, *The Construction of Finite Lattices*, Tohoku-Gakuin Univ. Faculty of Engineering Research Report, Vol. 20, No. 2, 1986, 71-79.

[35] E. Lawler, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart and Winston, New York, New York, 1976.

[36] B. D. McKay, Practical graph isomorphism, *Congressus Numerantium*, Vol. 30, 1981, 45-87.

[37] R. H. Möring, Algorithmic aspects of comparability graphs and interval graphs, in I. Rival, editor, *Graphs and Order: The role of Graphs in the Theory of Ordered Sets and Applications*, NATO Advanced Study Institute Series C: Math. Phys. Sci., Vol. 147, 1984, 41-102.

[38] A. Nijenhuis and H. S. Wilf, *Combinatorial Algorithms*, Academic Press, London, 1978.

[39] G.S. Rao, H.S. Stone and T.C. Hu, Scheduling in a Distributed Processor System with limited Memory, *IEEE Trans. Computers*, Vol. C-28, 1979, 291-299.

[40] R.C. Read, The number of k-colored graphs on labeled nodes, *Canadian Journal of Mathematics* Vol. 12, 1960, 409-413.

[41] R.C. Read, Everyone a winner or how to avoid isomorphism search when cataloging combinatorial configurations, *Annals of Discrete Mathematics* Vol. 2, 1978, 107-120.

[42] R.C. Read, A survey of graph generation techniques, *Eighth Australian Combinatorics Conference*, Deakin University 1980, Lecture Notes in Mathematics 884, Spring-Verlag, 1981, 77-89.

[43] R.C. Read, A New system for the Designation of Chemical Compounds. 1. Theoretical Preliminaries and the Coding of Acyclic Compounds, *Journal of Chemical Information and Computer Science*, Vol. 23, 1983, 135-149.

[44] R.C. Read, A New system for the Designation of Chemical Compounds. 2. Coding of Cyclic Compounds, *Journal of Chemical Information and Computer Science*, Vol. 25, 1985, 116-128.

[45] R.C. Read and D. Corneil, The graph isomorphism disease, *Journal of Graph Theory*, Vol 1, 1977, 339-363.

[46] Reingold, Nievergelt, and Deo, *Combinatorial Algorithms*, Prentice Hall, Englewood, New Jersey, 1977.

[47] R.W. Robinson, Enumeration of acyclic digraphs, in *Combinatorial Mathematics and its Applications* (R.C. Bose et al. eds.), University of North Carolina, Chapel Hill, 1970, 391-399.

[48] R.W. Robinson and A. Nymeyer, *Unlabeled bicolored graphs*, Project report for Numerical Implementation of Graph Counting Algorithms, File No. F75/15164, Mathematics Department, Newcastle University, NSW, Australia, May 1978.

[49] F. Ruskey, Listing and Counting Subtrees of a tree, *SIAM J. Computing*. Vol. 10, No. 1, 1981, 141-150.

[50] F. Ruskey, Private communication.

[51] N.J.A. Sloane, *A Handbook of Integer Sequences*, Academic Press, London, 1973.

[52] R.P. Stanley, *Enumerative Combinatorics*, Vol. I, Wadsworth & Brooks, Belmont, California, 1986.

[53] G. Steiner, An Algorithm to generate the Ideals of a Partial Order, *Operations Research Letters*, Vol. 5, No. 6, 1986.

[54] S.L. Tanimoto, Analysis of biomedical images using maximal matching, *Proceedings of 1976 IEEE Conference on Decision and Control Adaptive Processes*, Clearwater Beach, Florida, Dec., 1976, 171-176.

[55] H.F. Trotter, Algorithm 115: Permutation, *Communication of ACM*, Vol. 5, No. 8, 1962, 434-435.

[56] H.S. Wilf, *Combinatorial Algorithms: An Update*, CBMS 55, SIAM, 1989.

[57] R. Wilson and L. Beineke, *Applications of Graph Theory*, Academic Press, London, 1979.

Appendix A

An Example of Ideal Generation for a Tree Poset

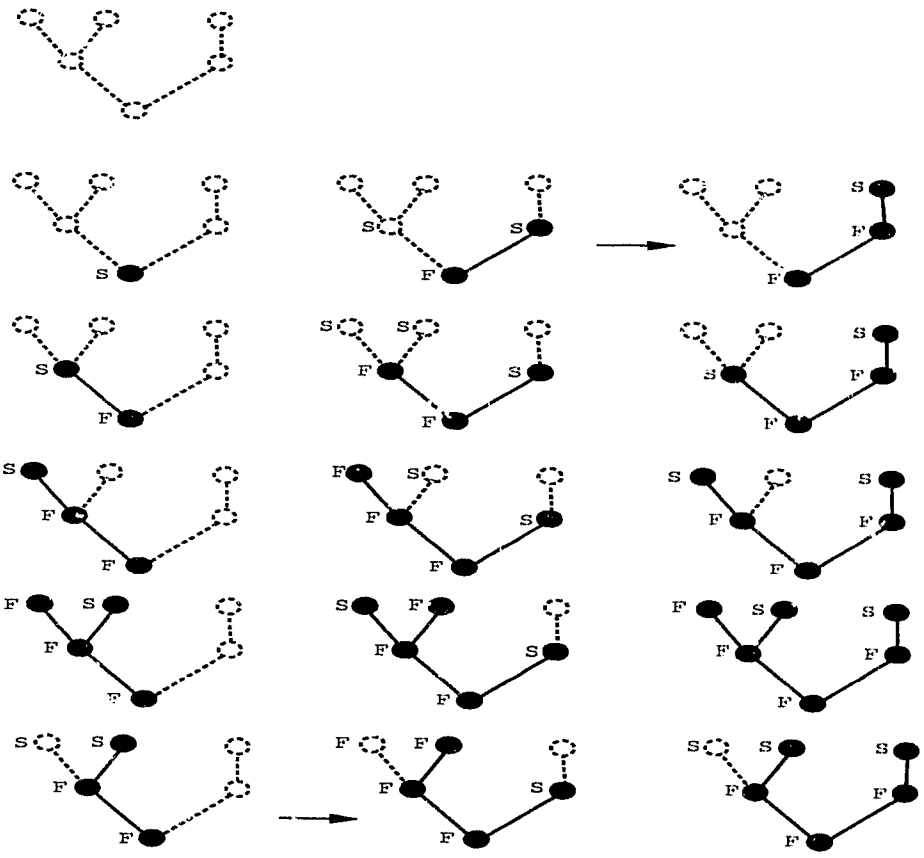

Figure A.1: The enumeration of ideals by Algorithm P

Appendix B

The BER Algorithm for Generating a Binary Reflected Gray Code

An $n-$bit Gray code $G(n)$ is the 2^n $n-$bit vector(codewords). Hence, the code is written in the form of a $2^n \times n$ binary matrix so that the ith codeword appears on the ith row of the matrix. Let $G(n)^R$ denote the matrix of $G(n)$ but the codewords are placed in the matrix in the reverse order. Then, the $n+1$bit Gray code $G(n+1)$ is expressed as follows.

$$G(n+1) = \begin{bmatrix} 0G(n) \\ 1G(n)^R \end{bmatrix}$$

In particular,

$$G(1) = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Let us number the columns of $G(n)$ from right to left. The transition sequence T_n of a Gray code $G(n)$ is the ordered list of the bit positions where a bit is inverted when we move from one codeword to the next. Then, the transition sequence T_n can be recursively defined by $T_1 = 1$, $T_{n+1} = T_n(n+1)T_n$, where $T_n^{reversed}$ is the sequence of T_n in the reverse order. Note that $T_n = T_n^{reversed}$ for all n. Thus, the transition sequence is equivalently

for $j = 0$ to $n+1$ **do**

$b_1 \leftarrow 0$

$i \leftarrow 0$

while $i < n+1$ **do**

output$(b_n, b_{n-1}, \ldots, b_1)$

$i \leftarrow \tau_0$

$b_i \leftarrow \overline{b_i}$

$\tau_0 \leftarrow 1$

$\tau_{i-1} \leftarrow \tau_i$

$\tau_i \leftarrow i+1$

Figure B.1: The BER algorithm for generating a Gray code

expressed as $T_1 = 1$ and $T_{n+1} = T_n(n+1)T_n$. If we have this transition sequence, we can readily generate a Gary code. An algorithm for generating the sequence T_n using a stack is depicted in Figure B.1. For more detail and the validity of the algorithm, see Bitner et al. [4]. We can easily observe that the stack content is created in restricted manner. When $j > 1$ is placed in the stack, we know later (after the number j is popped up), $j-1, j-2, \ldots, 1$ will be pushed on the stack. We use an array $(\tau_n, \tau_{n-1}, \ldots, \tau_0)$ to represent the content of the stack. The value of τ_0 points to the top element of the stack, and for $j \geq 1$, the value of τ_j points to the element below j on the stack if j is on stack. If j is not on the stack, the value of τ_j is reset to $j+1$. First we note that if $j+1$ is pushed on the stack, j is always on the above of $j+1$. If we pop up i, the elements $i-1, \ldots, 1$ will be pushed on the stack. Thus, when i is removed from the stack, we set $\tau_{i-1} \leftarrow \tau_i$ provided that τ_j was reset when j was popped up for all j, $i-1 \leq j \leq 1$. When $i \neq 1$ and the elements $i-1, \ldots, 1$ are pushed on the stack. Adding those elements can be done by the assignment $\tau_0 \leftarrow 1$.

Appendix C

Tables of Acyclic Digraphs, Posets, Finite Lattices and Finite Topologies

p	**	poset of size p	size of partition of p -1	Total
3	2	5	2	9
4	13	16	4	33
5	70	63	6	139
6	390	318	10	718
7	2476	2045	14	4535
8	18959	16999	21	35979
9	179823	183231	29	363083

Table C.1: The number of transitive digraphs.

** The number of transdigraphs which were obtained by assigning a number to each vertex of acyclic transdigraph and at least one of the vertices gets the number greater than one.

i \ j	1	2	3	4	5	6	7	Total
1	1							1
2	1	1						2
3	1	3	2					6
4	1	8	14	8				31
5	1	20	89	128	64			302
6	1	55	634	1934	2336	1024		5984
7	1	163	5668	36428	83648	84992	32768	243656

Table C.2: The number of unlabeled acyclic digraphs with i vertices and height j

	1	2	3	4	5	6	7	8	9	10	Total
1	1										1
2	1	1									2
3	1	3	1								5
4	1	8	6	1							16
5	1	20	31	10	1						63
6	1	55	162	84	15	1					318
7	1	163	940	734	185	21	1				2045
8	1	556	6372	7305	2380	356	28	1			16999
9	1	2222	52336	86683	35070	6259	623	36	1		183231
10	1	10765	534741	1261371	619489	125597	14258	1016	45	1	2567284

Table C.3: The number of unlabeled acyclic transitive digraphs(posets)

j / i	1	2	3	4	5	6	7	8	9	10	11	12	13	Total
1	1													1
2		1												1
3			1											1
4			1	1										2
5			1	3	1									5
6			1	7	6	1								15
7			1	15	26	10	1							53
8			1	33	103	69	15	1						222
9			1	71	408	426	150	21	1					1078
10			1	159	1638	2574	1307	286	28	1				5991
11			1	369	6853	15664	10889	3312	497	36	1			37622
12			1	896	30092	97740	89951	35898	7345	806	45	1		262775
13			1	2485	139676	630632	750364	379486	99745	14758	1239	55	1	2018442

Table C.4: The number of finite lattices with i elements and height j

Appendix D

Tables of Bicolored Graphs and Bicolorable Graphs

The numbers of bicolored graphs and the number of connected bicolorable graphs obtained by these algorithm s were compared with the values listed in a table by Robinson [48] and verified. The sizes of lists of bicolored and bicolorable graphs that were generated are summarized in Tables D.1 and D.2.

If the color classes are taken to be interchangeable then the entry for (1,1), (2,2), (3,3), (4,4) and (5,5) are $1, 2, 7, 36$

j / i	1	2	3	4	5	6	7	8	9	10
1	2	3	4	5	6	7	8	9	10	11
2	3	7	13	22	34	50	70	95	125	
3	4	13	36	87	190	386	734	1324		
4	5	22	87	317	1053	3250	9343			
5	6	34	190	386	5624	28576				

Table D.1: The number of unlabeled bicolored graphs with non-interchangeable color classes of orders i and j.

j / i	1	2	3	4	5	6	7	8	9	10
1	1	1	1	1	1	1	1	1	1	1
2	1	2	4	6	9	12	16	20	25	
3	1	4	13	34	76	155	290	510		
4	1	6	34	150	558	1824	5375			
5	1	9	76	558	3529	19687				

Table D.2: The number of unlabeled connected bicolored graphs with non-interchangeable color classes of orders i and j.