UNIVERSITÉ DU QUÉBEC À MONTRÉAL

EFFICIENT GENERATION OF THE IDEALS OF A POSET IN GRAY CODE ORDER

THÈSE

PRÉSENTÉE

COMME EXIGENCE PARTIELLE

DU DOCTORAT EN MATHÉMATIQUES

PAR

MOHAMED ABDO

MARS 2010

UNIVERSITÉ DU QUÉBEC À MONTRÉAL

EFFICIENT GENERATION OF THE IDEALS OF A POSET IN GRAY CODE ORDER

THESIS

PRESENTED

IN PARTIAL SATISFACTION OF THE REQUIREMENTS

OF THE DEGREE OF DOCTOR OF PHILOSOPHY IN MATHEMATICS

BY

MOHAMED ABDO

MARCH 2010

ACKNOWLEDGMENTS

I would like to thank Timothy Walsh, director of my research, for what he taught me, for his encouragement, for the time he spent answering my questions, for his valuable advice given to me when I was a teaching assistant for his course, and for his financial support. Without his supervision, this work would not have been completed. In addition, I thank him for the concerts to which he invited me when he was celebrating his birthday and elsewhere.

I would also like to thank Srečko Brlek for what he taught me and for his interest in students and especially for his valuable advice, encouragement and collaboration.

I thank my father Abdulhamid and my mother Naima who supported and constantly encouraged me to achieve my dreams in my studies.

I will never forget my brother Ahmad who passed away in November 2009. He was my real friend and support, we were as twins.

I'm glad that my wife was beside me while I was writing my thesis. I owe her a lot because she gives me the love and enthusiasm that I need.

Thanks to Manon Gauthier, who provides all the administrative advice and who shows much interest in students. Thanks to Gisèle Legault, who, with her knowledge of computers, allowed me to save much time. Thanks to Lise Tourigny, Jérôme Tremblay and all the staff in the Departments of Mathematics and Computer Science.

Finally, I am grateful to LaCIM, ISM, FARE, FQRNT and the Faculty of Sciences for their financial support during my studies for my M.Sc. and Ph.D. These scholarships were complementary to the financial support of my supervisor.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

RÉSUMÉ

Pruesse et Ruskey ont présenté un algorithme pour la génération de leur code Gray pour les idéaux d'un poset (ensemble partiellement ordonné) où deux idéaux adjacents diffèrent par un ou deux éléments. Leur algorithme fonctionne en temps amorti de $O(n)$ par idéal. Squire a présenté une récurrence pour les idéaux d'un poset qui lui a permis de trouver un algorithme pour générer ces idéaux en temps amorti de $O(\log n)$ par idéal, mais pas en code Gray. Nous utilisons la récurrence de Squire pour trouver un code Gray pour les idéaux d'un poset, où deux idéaux adjacents diffèrent par un ou deux éléments. Dans le pire des cas, notre algorithme a la même complexité que celle de l'algorithme de Pruesse et Ruskey et dans les autres cas, sa complexité est meilleure que celle de leur algorithme et se rapproche de celle de l'algorithme de Squire. Squire a donné une condition pour obtenir cette complexité. Nous avons trouvé une condition moins restrictive que la sienne. Cette condition nous a permis d'améliorer la complexité de notre algorithme.

Mots clés: poset, extension linéaire, cycle hamiltonien, code Gray, algorithme, complexité.

ABSTRACT

Pruesse and Ruskey presented an algorithm for generating their Gray code for the ideals of a poset, where two adjacent ideals differ by one or two elements. Their algorithm takes $O(n)$ amortized time per ideal, where n is the number of elements in the poset. Squire presented a recurrence for the ideals of a poset that enabled him to find an algorithm for generating these ideals in $O(\log n)$ amortized time per ideal, but not in Gray code order, where n is the number of elements in the poset. We use Squire's recurrence to find a Gray code for the ideals of a poset, where two adjacent ideals differ by one or two elements. In the worst case our algorithm has the same complexity as that of Pruesse and Ruskey and in the other cases its complexity is better and approaches that of Squire's algorithm. Squire gave a condition to obtain this complexity. We found a less restrictive condition than his. This condition enabled us to improve the complexity of our algorithm.

Key words: poset, linear extension, Hamiltonian cycle, Gray code, algorithm, complexity.

INTRODUCTION

"Humanity has long enjoyed making lists. All children delight in their new-found ability to count 1,2,3, etc., and it is a profound revelation that this process can be carried out indefinitely. The fascination of finding the next unknown prime or of listing the digits of π appeals to the general population, not just mathematicians. The desire to produce lists almost seems to be an innate part of our nature. Furthermore, the solution to many problems begins by listing the possibilities that can arise" (Ruskey, 2003).

It was not until 1960 that generation of such combinatorial lists would be feasible with the advent of the computer (Lehmer, 64). However, for such a listing to be possible, generation methods must be efficient. A common approach has been to try to generate such lists so that successive objects differ by some predefined small amount. A famous example is the binary reflected Gray code (Gilbert, 58; Gray, 53) which consists of listing all the binary numbers of the same size so that successive numbers differ in exactly one bit.

The origins of listing combinatorial objects so that successive objects differ only by small amount can be found in the early work of (Gray, 53; Wells, 61; Trotter, 62; Johnson, 63; Lehmer, 65; Chase, 70; Ehrlich, 73; Nijenhuis and Wilf, 78). However, the term combinatorial Gray code first appeared in (Joichi, White and Williamson, 80).
There are many combinatorial Gray codes such as:

1. listing all the permutations of $1, 2, \ldots, n$ so that successive permutations differ only by the swap of one pair of adjacent elements (Johnson, 63; Trotter, 62);

2. listing all the k-element subsets of an n-element set in such a way that successive sets differ by exactly one element (Bitner, Ehrlich and Reingold, 76; Buck and

Wiedemann, 84; Eades, Hickey, and Read, 84; Eades and McKay, 84; Nijenhuis and Wilf, 78; Ruskey, 88; Liu and Tang, 73);

3. listing all the partitions of an integer n so that in successive partitions, one part has increased by one and one part has decreased by one (Savage, 89);

4. listing the linear extensions of certain posets so that successive elements differ only by a transposition (Ruskey, 92; Pruesse and Ruskey, 91; Stachowiak, 92; West, 93);

5. listing all the binary trees of a given size so that consecutive trees differ only by a rotation at a single node (Lucas, 87; Lucas, Roelants and Ruskey, 93; Proskurowski and Ruskey, 90);

6. listing the well-formed parenthesis strings of a given length in such a way that successive strings differ by a transposition of two letters (Proskurowski and Ruskey, 90). Walsh is the first who gave a loop-free generation algorithm for this problem (Walsh, 98);

7. listing all the length-n involutions so that each involution is transformed into its successor via one or two transpositions or a rotation of three elements (Walsh, 2001).

8. listing all the ideals of a poset so that successive ideals differ in one or two elements (Abdo, 2009; Pruesse and Ruskey, 93; Koda and Ruskey, 93).

This thesis presents another Gray code for the ideals of a poset and an algorithm for generating these ideals that runs as least as fast as that of (Pruesse and Ruskey, 93) on all the posets on which we tested it and considerably faster on some of them. The definition of a poset and of an ideal, an outline of the thesis and a summary of the major results appear in the first two pages of Chapter 1.

One can find applications of the Gray code to circuit design (Robinson and Cohn, 81), signal processing (Ludman, 81), ordering of documents on shelves (Losee, 92), data compression (Richard, 86), statistical computing (Diaconis and Holmes, 94), combinatorial map theory (Cori, 75), graphics and image processing (Amalraj, Sundararajan and Dhar, 90), processor allocation in the hypercube (Chen and Shin, 90), hashing (Faloutsos, 88), computing the permanent (Nijenhuis and Wilf, 78), information storage and retrieval (Chang, Chen and Chen, 92), and puzzles, such as the Chinese Rings and Towers of Hanoi (Gardner, 72).

"Although many Gray code schemes seem to require strategies tailored to the problem at hand, a few general techniques and unifying structures have emerged. The paper (Joichi, White and Williamson, 80) considers families of combinatorial objects, whose size is defined by a recurrence of a particular form, and some general results are obtained about constructing Gray codes for these families. Ruskey shows in (Ruskey, 92) that certain Gray code listing problems can be viewed as special cases of the problem of listing the linear extensions of an associated poset so that successive extensions differ by a transposition. In the other direction, the discovery of a Gray code frequently gives new insight into the structure of the combinatorial class involved" (Savage, 97).

Walsh generalized Chase's method (Chase, 89) so that all the words in a suffix-partitioned list form an interval of consecutive words (Walsh, 95). Moreover, he gave sufficient conditions on a Gray code in order that Ehrlich's method (Ehrlich, 73) possesses a loop-free implementation and he generalized this method so that it works under less restrictive conditions (Walsh, 2000).

"So, the area of combinatorial Gray codes includes many questions of interest in combinatorics, graph theory, group theory, and computing, including some well-known open problems" (Savage, 97).

CHAPTER I

GRAY CODE

1.1 Introduction

A *poset* (partially ordered set) $\mathcal{P}$ is a set E of elements together with a reflexive, transitive, anti-symmetric relation $R(\mathcal{P})$ on E. An *up-set* of a poset $\mathcal{P}$ is a subset U of E such that if $x \in U$ and $y \geq x$, then $y \in U$. A *down-set* of a poset $\mathcal{P}$ is a subset D of E such that if $x \in D$ and $y \leq x$, then $y \in D$. The set of *ideals* of a poset is either the set of all its up-sets or the set of all its down-sets. Generating the ideals of a poset has several applications in optimization problems, including scheduling, reliability and line balancing problems (Steiner, 86). Schrage and Baker (Schrage and Baker, 78), Lawler (Lawler, 79), and Ball and Provan (Ball and Provan, 83) all presented algorithms that generate the set $\mathcal{U}(\mathcal{P})$ of up-sets of $\mathcal{P}$ in $O(n^2 \cdot u(\mathcal{P}))$ time, where $u(\mathcal{P})$ is the cardinality of $\mathcal{U}(\mathcal{P})$. Steiner (Steiner, 86) was the first to present an $O(n \cdot u(\mathcal{P}))$ generation algorithm. Squire (Squire) presented a recurrence for the ideals of a poset and used it to generate these ideals in $O(\log n \cdot u(\mathcal{P}))$ time. But none of these algorithms, including that of Squire, generates them in *Gray code order* - that is, so that two adjacent ideals in the list differ by a bounded number of elements. The advantage of Gray code order is that certain properties of the members of a list can be updated quickly if adjacent members of a list differ only slightly.

If the Hasse diagram of the poset $\mathcal{P}$ is a forest, the algorithms of Beyer and Ruskey (constant average time generation of subtrees of bounded size, 1989, unpublished manuscript), and Koda and Ruskey (Koda and Ruskey, 93) can generate its up-sets in Gray code order in $O(u(\mathcal{P}))$ time. Another class of posets for which an efficiently implementable Gray code exists is presented in (Knuth and Ruskey, 2003). However, for arbitrary posets, the best known algorithms, including that of Pruesse and Ruskey (Pruesse and Ruskey, 93), may still take $O(\Delta \cdot u(\mathcal{P}))$ time, where Δ is the maximum number of elements that cover any element of $\mathcal{P}$. Their algorithm generates a list of the ideals of a poset so that two ideals that are adjacent in the list differ by one or two elements.

We say that an element v *covers* an element t if $t < v$ and there is no element u such that $t < u < v$. When v covers t, we say that t is *covered by* v.

In this thesis an ideal will mean a down-set. We generalize Squire's recurrence and use our generalized recurrence to list the elements of $\mathcal{D}(\mathcal{P})$, the down-sets of $\mathcal{P}$, so that two ideals that are adjacent in the list differ by one or two elements, as does the algorithm of Pruesse and Ruskey (Pruesse and Ruskey, 93). A preliminary version of our algorithm, described in Section 1.3 and published in (Abdo, 2009), is similar to the one in (Pruesse and Ruskey, 93) except that it lists each ideal once instead of twice. The presentation here is simpler than the one in (Pruesse and Ruskey, 93) because it deals only with posets, whereas the one in (Pruesse and Ruskey, 93) is generalized to antimatroids. We were recently informed that the proof in (Pruesse and Ruskey, 93) was recast in terms of posets in Theorem 4.4 (Chow and Ruskey, 09), published in the same year as (Abdo, 2009). Both algorithms have the same time-complexity. However, we were able to improve the time-complexity of our algorithm so that it never ran more than a few percent slower than Squire's on any of the posets on which we tested it, whereas the Pruesse-Ruskey algorithm ran considerably slower on some of them.

The theory behind our algorithm is presented in Chapter 1, the algorithm itself in Chapter 2, the experimental comparison of our algorithm with the Pruesse-Ruskey algorithm and the Squire algorithm in Chapter 3 and the source code of all three algorithms in the Appendix A.

In our present algorithm we almost always choose the median element of a linear extension of $\mathcal{P}$. If the number of elements of $\mathcal{P}$ is n then the position of the median is $\lfloor\frac{n+1}{2}\rfloor$.

We introduce the following notation that will be used in equation (1.1) below. Let $\mathcal{P}$ be a poset on a set E, D a down-set of P and Y a subset of E such that $D \cap Y = \emptyset$ and $D \cup Y$ is a down-set of $\mathcal{P}$. Also, denote by $\mathcal{D}(\mathcal{P}, D, Y)$ the set $\{I \in \mathcal{D}(\mathcal{P}) : D \subseteq I \text{ and } I \subseteq D \cup Y\}$ of down-sets of $\mathcal{P}$ that contain the down-set D and whose other elements are chosen from Y. Finally, let x be some element of Y and $D[x]$ and $U[x]$ be, respectively, the elements of $\mathcal{P}$ that are $\leq x$ and $\geq x$. Note that $\mathcal{D}(\mathcal{P}) = \mathcal{D}(\mathcal{P}, \emptyset, E)$.

Squire's recurrence for the down-sets is given by equation (1.1):

$$\mathcal{D}(\mathcal{P}, D, Y) = \mathcal{D}(\mathcal{P}, D, Y \setminus U[x]) \cup \mathcal{D}(\mathcal{P}, D \cup D[x], Y \setminus D[x]). \tag{1.1}$$

This recurrence is solvable because $\#(Y \setminus U[x])$ and $\#(Y \setminus D[x])$ are both strictly less than $\#(Y)$, where $\#(S)$ means the cardinality of the set S, and $\mathcal{D}(\mathcal{P}, D, \emptyset) = D$.

This recurrence is called initially with $D = \emptyset$ and $Y = E$ to generate $\mathcal{D}(\mathcal{P})$. In what follows we call $\mathcal{D}(\mathcal{P}, D, Y \setminus U[x])$ the *first part* and $\mathcal{D}(\mathcal{P}, D \cup D[x], Y \setminus D[x])$ the *second part*. These two parts are disjoint. Usually we consider $\mathcal{D}(\mathcal{P}, D, Y \setminus D[x])$ as the second part and then we add $D[x]$ to each ideal of this part. Squire's method of generating the ideals of $\mathcal{P}$ in $O(\log n \cdot u(\mathcal{P}))$ time depends upon choosing for x the "median" element of Y in the following sense. An *extension* of a poset $\mathcal{P}$ on a set E is a poset $\mathcal{Q}$ on E such that $R(\mathcal{P}) \subseteq R(\mathcal{Q})$. An extension of $\mathcal{P}$ that is a total order is called a *linear extension* of $\mathcal{P}$. Before calling the recurrence (1.1), Squire constructs a linear extension of $\mathcal{P}$. The elements of Y are then listed in the order that is compatible with that of the linear extension, and the median element x_m in this list is chosen for x. We illustrate the use of (1.1) in the following example.

Example 1. *Let $\mathcal{P}$ be the poset whose Hasse diagram is on the left side (of the arrow) in Figure 1.1. In equation (1.1), let D be the empty down-set $\emptyset$ and Y be the set $\{1, 2, ..., 7\}$. The linear extension chosen for $\mathcal{P}$ is $1 < 2 < 3 < 4 < 5 < 6 < 7$ so that the median element x_m is 4. The diagram immediately to the right of the arrow*

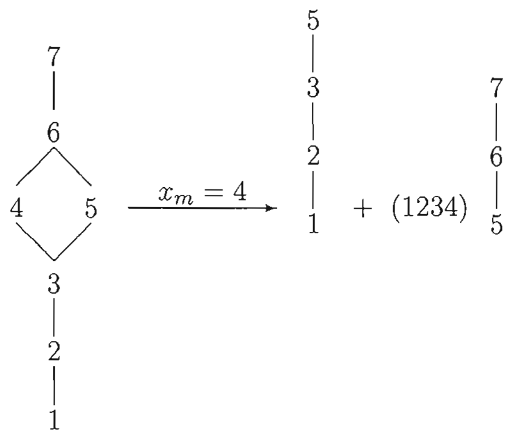


Figure 1.1. Poset of Example 1 is on the left side.

is the poset restricted to $Y \setminus U[4]$ and the rightmost diagram is the poset restricted to $Y \setminus D[4]$ with $D[4]$ (with the commas removed and the braces replaced by brackets) written to its left. The set of all the down-sets of $\mathcal{P}$ is the union of two disjoint sets (parts). The first part is the set of down-sets whose elements are restricted to the set $Y \setminus U[4] = \{1,2,3,5\}$ and the second part is the set of down-sets that contain all of the elements of $D[4] = \{1,2,3,4\}$ and whose other elements are restricted to the set $Y \setminus D[4] = \{5,6,7\}$. Each of these parts can then be partitioned into two parts and so on until all the parts contain at most two down-sets (corresponding to Y which contains at most one element).

1.2 Construction of a Hamiltonian cycle

A *Hamiltonian cycle* in a graph is a cycle that passes through each vertex exactly once. Given a poset $\mathcal{P}$, we call $G(\mathcal{P})$ the graph whose vertices are the down-sets of $\mathcal{P}$, where two vertices are *adjacent* (joined by an edge) if the corresponding down-sets differ by one element and $G^2(\mathcal{P})$ the graph with the same vertex set as $G(\mathcal{P})$ but which has an edge between every pair of vertices that are connected by a path of length at most 2 in $G(\mathcal{P})$. We use Squire's recurrence to construct a Hamiltonian cycle in $G^2(\mathcal{P})$, so that two adjacent down-sets, as well as the first and last ones, differ by one or two elements. In what follows, when we refer to a Hamiltonian cycle in $\mathcal{P}$, we mean a Hamiltonian cycle in $G^2(\mathcal{P})$.

In our construction, the basic step is to call a single vertex and a single edge (traversed in both directions) a Hamiltonian cycle, so that a single down-set or a pair of down-sets that differ by at most two elements constitutes a Hamiltonian cycle. For the induction step we assume that a Hamiltonian cycle has been found for the first part $\mathcal{D}(\mathcal{P}, D, Y \setminus U[x])$ of (1.1) and also for the second part $\mathcal{D}(\mathcal{P}, D \cup D[x], Y \setminus D[x])$ of (1.1) and we merge these two Hamiltonian cycles into a single one for $\mathcal{D}(\mathcal{P}, D, Y)$.

1.3 Hamiltonian cycle when the median is a minimal element

Theorem 1 below (Abdo, 2009) shows that a Hamiltonian cycle can be constructed for any poset. We call x a *maximal (minimal)* element of a poset $\mathcal{P}$ if no element of $\mathcal{P}$ is strictly greater (less) than x. When a poset has only one maximal (minimal) element, this element is called maximum (minimum). We note that any finite poset must contain at least one maximal element and one minimal element.

Theorem 1. *Let $\mathcal{P}$ be a poset on a set E and let x be a minimal element of $\mathcal{P}$. Then there is a Hamiltonian cycle of down-sets in $\mathcal{P}$ in which two adjacent down-sets differ by one or two elements and this cycle contains the edge $\{\emptyset, \{x\}\}$ consisting of the empty down-set and the singleton $\{x\}$.*

Proof. (By induction on $n = \#(E)$, the cardinality of E).

Basic step. If $n = 1$, then the poset consists of a single element x, which is necessarily a minimal element, and has two ideals $\emptyset$ and $\{x\}$, which differ by a single element. Then the edge $\{\emptyset, \{x\}\}$ is the required Hamiltonian cycle.

Induction step. Suppose that $n > 1$. We apply (1.1) with the initial values $D = \emptyset$ and $Y = E$ and with x a minimal element of Y. Since x is a minimal element of Y, every element of Y is either in $U[x]$ or else is incomparable with x. Then $\mathcal{P}$, restricted to Y, is the poset on the left side of Figure 1.2. Since $D[x] = \{x\}$, the poset of the first part

$$\mathcal{P} \quad \xrightarrow{x_m = x} \quad \mathcal{P} \setminus U[x] \quad + \quad (x) \quad \mathcal{P} \setminus \{x\}$$

Figure 1.2. Left side is the poset having x as a minimal element.

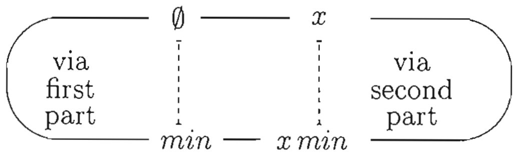


Figure 1.3. Cycle of Case 1 of Theorem 1.

of (1.1) (restricted to $Y \setminus U[x]$) consists of $\mathcal{P} \setminus U[x]$ and the poset of the second part (restricted to $Y \setminus D[x] = Y \setminus \{x\}$) is of the form shown to the right of the + sign in Figure 1.2. There are two cases to consider, depending upon whether or not $Y \setminus U[x]$ is empty.

Case 1: Suppose that $Y \setminus U[x] \neq \emptyset$. Let min be a minimal element of $Y \setminus U[x]$. Since $x \notin Y \setminus U[x]$, $\#(Y \setminus U[x]) < \#(E)$, so that by the induction hypothesis there is a Hamiltonian cycle of down-sets of the first part in which two adjacent down-sets differ by one or two elements and this cycle contains the edge $\{\emptyset, \{min\}\}$. The second part is restricted to the set $Y \setminus \{x\}$, which is not empty because $n > 1$ and whose cardinality $< \#(E)$ because it does not contain x. In addition, $Y \setminus \{x\}$ has min as minimal element. It follows from the induction hypothesis that this part too has a Hamiltonian cycle of down-sets in which two adjacent down-sets differ by one or two elements and that this cycle too contains the edge $\{\emptyset, \{min\}\}$. When x is added to each ideal of this second cycle, this cycle will contain the edge $\{\{x\}, \{x, min\}\}$. By removing the edges $\{\emptyset, \{min\}\}$ and $\{\{x\}, \{x, min\}\}$ and adding the edges $\{\emptyset, \{x\}\}$ and $\{\{min\}, \{x, min\}\}$ we obtain the required Hamiltonian cycle containing the edge $\{\emptyset, \{x\}\}$ (see Figure 1.3).

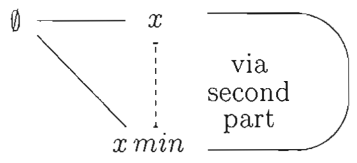


Figure 1.4. Cycle of Case 2 of Theorem 1.

Case 2: Suppose that $Y \setminus U[x] = \emptyset$. Let min be a minimal element of $Y \setminus D[x]$. Such an element exists because $Y \setminus \{x\} \neq \emptyset$ since $n > 1$. The first part of (1.1) consists of the single down-set $\emptyset$. By an argument similar to that in Case 1, the second part of (1.1) has a Hamiltonian cycle containing the edge $\{\emptyset, \{min\}\}$, which becomes $\{\{x\}, \{x, min\}\}$ when x is added to each down-set. By removing the edge $\{\{x\}, \{x, min\}\}$ and adding the edges $\{\emptyset, \{x\}\}$ and $\{\emptyset, \{x, min\}\}$ we obtain the required Hamiltonian cycle containing the edge $\{\emptyset, \{x\}\}$ (see Figure 1.4). $\square$

To simplify the notation, the set $\{x_1, x_2, \cdots, x_n\}$ will be denoted by $x_1 x_2 \cdots x_n$.

Remark 1. *We remove an edge from a cycle only when this cycle is not an edge or if it is an edge and each of its vertices will be joined by an edge with two other vertices.*

Example 2. *Consider the poset on the left side of the first line of Figure 1.5. By choosing* 1 *for the minimal element* x_m *and applying Theorem 1, Case 2, we obtain the first line of Figure 1.5. The first part is the empty poset which has the Hamiltonian cycle* $\emptyset$. *The second part is the poset* 2 *which has the Hamiltonian cycle* $\{\emptyset, 2\}$. *By adding the element* 1 *to each ideal of this last cycle we obtain the Hamiltonian cycle* $\{1, 12\}$. *Since we applied Theorem 1, Case 2, we add the edges* $\{\emptyset, 1\}$ *and* $\{\emptyset, 12\}$ *but according to Remark 1, we do not remove the edge* $\{1, 12\}$. *Thus, we obtain the required Hamiltonian cycle* $\{\emptyset, 1, 12\}$ *which is on the second line of Figure 1.5. We note that this cycle contains the edge* $\{\emptyset, 1\}$.

Example 3. *Consider the poset on the left side of the first line of Figure 1.6. By choosing* 3 *for the minimal element* x_m *and applying Theorem 1, Case 1, we obtain the*

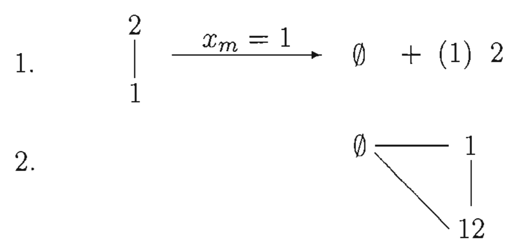


Figure 1.5. Example illustrating the use of Theorem 1, Case 2.

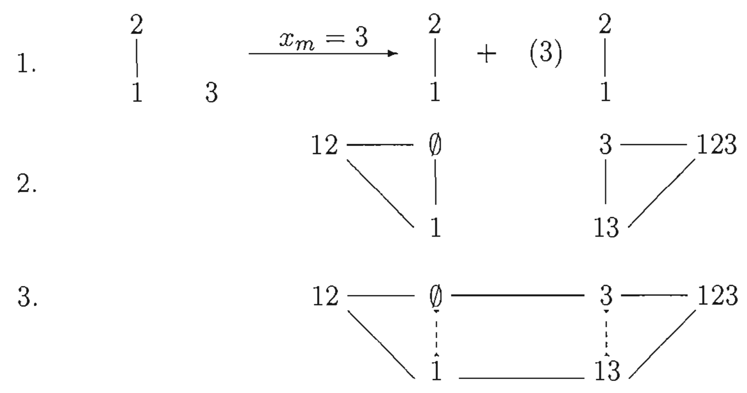


Figure 1.6. Example illustrating the use of Theorem 1, Case 1.

first line of Figure 1.6. Each of the first and the second parts is the poset $\{1, 2\}$ *where* $1 < 2$ *which, according to Example 2, has the Hamiltonian cycle* $\{\emptyset, 1, 12\}$. *By adding the element* 3 *to each ideal of the cycle of the second part we obtain the Hamiltonian cycle* $\{3, 13, 123\}$. *The cycle of each part is on the second line of Figure 1.6. Since we applied Theorem 1, Case 1, we add the edges* $\{\emptyset, 3\}$ *and* $\{1, 13\}$ *and remove the edges* $\{\emptyset, 1\}$ *and* $\{3, 13\}$. *Thus, we obtain the required Hamiltonian cycle on the third line of Figure 1.6. We note that this cycle contains the edge* $\{\emptyset, 3\}$.

By limiting ourselves to this theorem, we find an algorithm for generating the ideals of a poset in Gray code order whose complexity is identical to the algorithm of Pruesse and Ruskey (Pruesse and Ruskey, 93), but which generates each ideal only once instead of twice.

We note that the cycle generated by our algorithm is not identical to the one generated by that of (Pruesse and Ruskey, 93). For example, when executed on the poset $\{1, 2, 3, 4\}$ with $1 < 2$ and $2 < 3$, the Pruesse-Ruskey algorithm generates the cycle $\emptyset, 1, 123, 12, 1234, 124, 14, 4$, whereas our algorithm generates the cycle $\emptyset, 1, 123, 12, 124, 1234, 14, 4$.

1.4 Hamiltonian cycle when there is exactly one element less than the median

We recall that Squire's method of generating the down-sets of $\mathcal{P}$ in $O(\log n \cdot u(\mathcal{P}))$ time depends upon choosing for x the "median" element of a linear extension of Y. In our construction it is not always possible to choose the median element for x; however, we choose the element of Y that is as close as possible to the median element. In this way we improve the computational complexity of the generation algorithm.

To this end we have generalized (1.1) and used this generalization to design several other constructions analogous to that of Theorem 1, where x is chosen to be the median element, or close to it, rather than a minimal element. For certain posets, it turns out that the median is a minimal element; this case is treated by Theorem 1.

The other two cases - where there is exactly one, or more than one, element less than the median - are treated by Theorem 2 and Theorem 3 respectively, which are presented in the rest of Chapter 1. These three theorems, taken together, enabled us to improve the performance of the generation algorithm.

Theorem 2. *Let E be the underlying set of a poset $\mathcal{P}$ of which the median x covers only one element x_1, which is minimal. Then there is a Hamiltonian cycle of down-sets in $\mathcal{P}$ in which two adjacent down-sets differ by one or two elements and this cycle contains the edge $\{\emptyset, x_1x\}$.*

Proof. Let n be the cardinality of E ($n = \#(E)$). We consider, as we did in Theo-

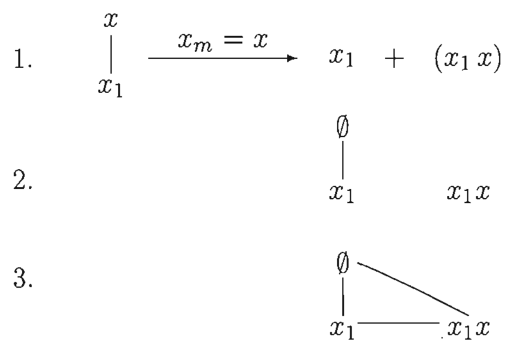


Figure 1.7. First case of Theorem 2.

rem 1, Squire's recurrence (1.1) for ideals, which we restate here:

$$\mathcal{D}(\mathcal{P}, D, Y) = \mathcal{D}(\mathcal{P}, D, Y \setminus U[x]) \cup \mathcal{D}(\mathcal{P}, D \cup D[x], Y \setminus D[x]),$$

where x is the median. We distinguish two cases.

1. If $n = 2$, then the poset is on the left side of the first line of Figure 1.7. We apply equation (1.1) with the initial values $D = \emptyset$ and $Y = E$ and with x as median element of Y. We obtain the first line in Figure 1.7. The Hamiltonian cycles for the first and second parts are on the second line of the same figure. By adding the edges $\{x_1\,x,\ \emptyset\}$ and $\{x_1\,x,\ x_1\}$ we obtain the required Hamiltonian cycle that is on the third line of the same figure and which contains the edge $\{\emptyset, x_1x\}$.

2. If $n > 2$, then by applying equation (1.1) with the initial values $D = \emptyset$ and $Y = E$ and with x as median element of Y, we obtain the first line of Figure 1.8. Since x_1 is a minimal element in $\mathcal{P}$ and x covers only x_1, x_1 is also a minimal element in $\mathcal{P} \setminus U[x]$. Applying Theorem 1 to $\mathcal{P} \setminus U[x]$ with x_1 as median element of $Y \setminus U[x]$, we obtain a Hamiltonian cycle containing the edge $\{\emptyset, x_1\}$. On the other hand, the poset in the second part of line 1 in Figure 1.8 is not empty since $n > 2$ and $D[x]$=2. It follows that this poset has a minimal element *min* and by applying Theorem 1 with *min* as median element of $Y \setminus \{x1, x\}$ we obtain a Hamiltonian cycle containing the edge $\{x_1\,x, x_1\,x\,min\}$. Finally by removing the edges $\{\emptyset,\ x_1\}$

1. $\mathcal{P} \xrightarrow{x_m = x} \mathcal{P} \setminus U[x] \ + \ (x_1\, x) \ \ \mathcal{P} \setminus \{x_1, x\}$

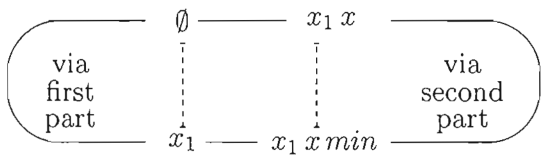
2. 

Figure 1.8. Second case of Theorem 2.

and $\{x_1\, x,\ x_1\, x\, min\}$ and adding the edges $\{\emptyset,\ x_1\, x\}$ and $\{x_1,\ x_1\, x\, min\}$ we obtain the required Hamiltonian cycle that is on the second line of Figure 1.8 and which contains the edge $\{\emptyset, x_1 x\}$. □

Example 4. *Consider the poset on the left side of the first line of Figure 1.9. By choosing* 2 *as median element* x_m *and applying Theorem 2, we obtain the first line of Figure 1.9. The poset of the first part has the Hamiltonian cycle* $\{\emptyset, 1, 12, 2\}$ *(by applying Theorem 1 to this poset). The second part is the poset* 3 *which has the Hamiltonian cycle* $\{\emptyset, 3\}$*. By adding the set* 12 *to each ideal of this last cycle we obtain the Hamiltonian cycle* $\{12, 123\}$*. The cycle of each part is on the second line of Figure 1.9. Since we applied Theorem 2, we add the edges* $\{\emptyset, 12\}$ *and* $\{1, 123\}$ *and remove the edge* $\{\emptyset, 1\}$ *but according to Remark 1, we do not remove the edge* $\{12, 123\}$*. Thus, we obtain the required Hamiltonian cycle on the third line of Figure 1.9.*

Remark 2. *Any minimal element in a poset may be chosen as median and consequently it may be a* neighbor *of* $\emptyset$ *(adjacent to* $\emptyset$*) when we apply Theorem 1. Sometimes we want to keep this choice for subsequent step; to this end we overline this minimal element (we draw a line above it).*

The purpose of overlining an element instead of applying Theorem 1 directly is to allow the choice of the median to be made as often as possible. These overlines are

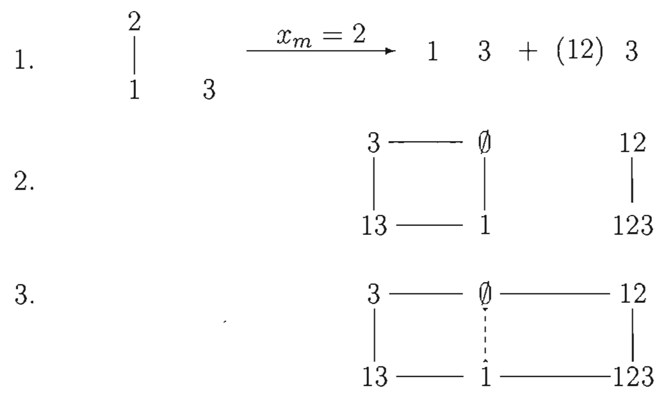


Figure 1.9. Example illustrating the use of Theorem 2.

transferable only to the first part in the next step of applying equation (1.1) since only for the first part do we have $\emptyset$. In Squire's equation (1.1), we choose x as median x_m of the linear extension of a poset $\mathcal{P}$. When this median is a minimal element, or when it covers only one element, which is minimal, we apply Theorem 1 or Theorem 2, respectively; otherwise we apply Theorem 3, which we present below. The number of overlined elements must not exceed 2 since $\emptyset$ cannot be a neighbor of more than two elements in a cycle. Therefore, when we have two overlined elements, we can no longer choose a median element; the method for dealing with this situation is explained below in Lemma 1 and Lemma 2. Before presenting these lemmas, we generalize Squire's recurrence for two non-comparable elements instead of one element.

1.5 Generalization of Squire's recurrence

Proposition 1. *We have*

$$\begin{aligned}\mathcal{D}(\mathcal{P}, D, Y) \;&=\; \mathcal{D}(\mathcal{P}, D, Y \setminus U[x,y]) \;\cup\; \mathcal{D}(\mathcal{P}, D \cup D[x], Y \setminus (D[x] \cup U[y])) \;\cup \\ &\mathcal{D}(\mathcal{P}, D \cup D[y], Y \setminus (D[y] \cup U[x])) \;\cup\; \mathcal{D}(\mathcal{P}, D \cup D[x,y], Y \setminus D[x,y]), \qquad (1.2)\end{aligned}$$

where x and y are non-comparable elements of Y and $U[x,y] = U[x] \cup U[y]$, $D[x,y] = D[x] \cup D[y]$. The meaning of the other symbols is the same as in the beginning of the thesis.

Proof. According to Squire, we have, for some x in $\mathcal{P}$, Equation (1.1):

$$\mathcal{D}(\mathcal{P}, D, Y) = \mathcal{D}(\mathcal{P}, D, Y \setminus U[x]) \cup \mathcal{D}(\mathcal{P}, D \cup D[x], Y \setminus D[x]).$$

Since x and y are non-comparable, $y \in Y \setminus U[x]$ and $y \in Y \setminus D[x]$. Consequently, $y \in Y \setminus U[x]$, so that

$$\mathcal{D}(\mathcal{P}, D, Y \setminus U[x]) = \mathcal{D}(\mathcal{P}, D, Y \setminus U[x, y]) \quad \cup \quad \mathcal{D}(\mathcal{P}, D \cup D[y], Y \setminus (D[y] \cup U[x]))$$

and $y \in Y \setminus D[x]$, so that

$$\mathcal{D}(\mathcal{P}, D{\cup}D[x], Y{\setminus}D[x]) = \mathcal{D}(\mathcal{P}, D{\cup}D[x], Y{\setminus}(D[x]{\cup}U[y])) \;\cup\; \mathcal{D}(\mathcal{P}, D{\cup}D[x, y], Y{\setminus}D[x, y])$$

By substituting these last two equalities into Equation (1.1) we find the recurrence announced above. □

1.6 Hamiltonian cycle using our generalization of Squire's recurrence

Lemma 1. *If a poset $\mathcal{P}$ has only two minimal elements, then there is a Hamiltonian cycle of down-sets in $\mathcal{P}$ in which two down-sets differ by one or two elements and such that $\emptyset$ is adjacent to each of these two minimal elements.*

Proof. Let $\mathcal{P}$ be a poset that has only two minimal elements x and y. By applying the generalized recurrence of Proposition 1, Equation (1.2), we obtain the first line of Figure 1.10. On the right side of the arrow we have 4 parts, the first of which is $\emptyset$.

The first part is empty since the poset has only two minimal elements. Let min_1 be a minimal element of the second part. In this case, it is also a minimal element of the fourth part since $Y \setminus (U[y] \cup \{x\}) \subseteq Y \setminus \{x, y\}$. Let min_2 be a minimal element of the third part. According to Theorem 1, there is a Hamiltonian cycle of the down-sets of the second part containing the edge $\{\emptyset,\ min_1\}$. Since x is included in every down-set of this cycle, this cycle contains the edge $\{x,\ x\, min_1\}$. By an argument similar to the previous one, there are two cycles of down-sets for the third and fourth parts containing, respectively, the edges $\{y,\ y\, min_2\}$ and $\{xy,\ x\, y\, min_1\}$. These cycles are on the second

1. $\mathcal{P} \xrightarrow{x,y} \emptyset + (x)\ \ \mathcal{P}\setminus(U[y]\cup\{x\}) + (y)\ \ \mathcal{P}\setminus(U[x]\cup\{y\})\ + (xy)\ \ \mathcal{P}\setminus\{x,y\}$

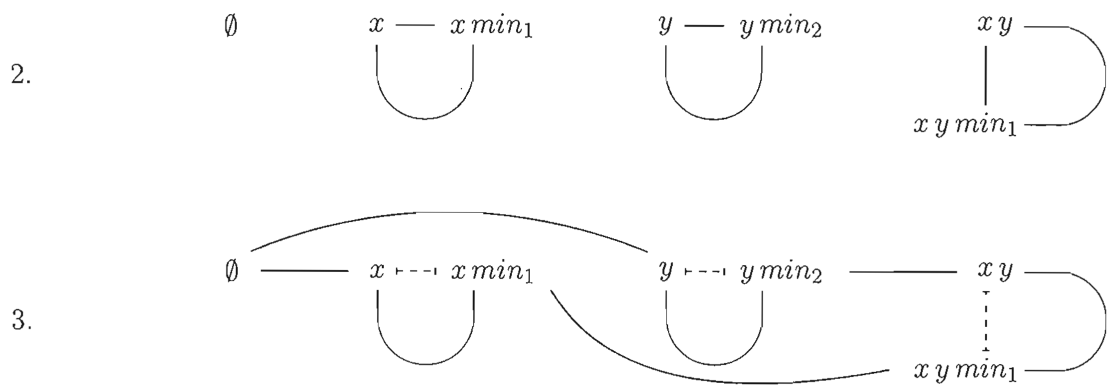


Figure 1.10. Illustration of Lemma 1.

line of Figure 1.10. On the third line we removed some edges and added others to obtain the required Hamiltonian cycle. The dashed lines are the removed edges. We can easily verify that when one or more among the second, third and/or fourth parts are empty the corresponding cycles are reduced to one element and we always have a Hamiltonian cycle of down-sets in which two adjacent down-sets differ by one or two elements, with $\emptyset$ being adjacent to each of these two minimal elements. $\square$

Example 5. *Consider the poset on the left side of the first line of Figure 1.11. By choosing* 1 *and* 3 *for the minimal elements (the elements* x *and* y*) and applying Lemma 1, we obtain the first line of Figure 1.11. The first part is the empty poset whose underlying set is* $\emptyset$*; consequently it has the Hamiltonian cycle* $\emptyset$*. The second part is the empty poset whose underlying set is* 2*; therefore it has the Hamiltonian cycle* $\{\emptyset, 2\}$*. By adding the element* 1 *to each ideal of this last cycle we obtain the Hamiltonian cycle* $\{1, 12\}$*. The third part is the empty poset whose underlying set is* 4*; therefore it has the Hamiltonian cycle* $\{\emptyset, 4\}$*. By adding the element* 3 *to each ideal of this last cycle we obtain the Hamiltonian cycle* $\{3, 34\}$*. The fourth part is the empty poset whose under-*

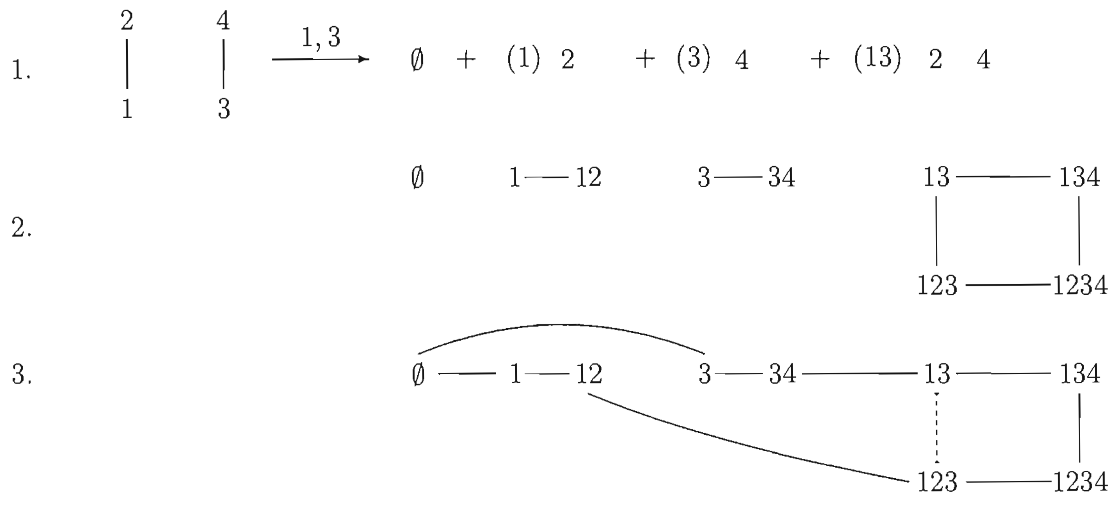


Figure 1.11. Example illustrating the use of Lemma 1.

lying set is $\{2, 4\}$*; therefore it has the Hamiltonian cycle* $\{\emptyset, 2, 24, 4\}$*. By adding the set* 13 *to each ideal of this last cycle we obtain the Hamiltonian cycle* $\{13, 123, 1234, 134\}$*. The Hamiltonian cycle of each part is on the second line of Figure 1.11.*

Since we applied Lemma 1, we add the edges $\{\emptyset, 1\}$, $\{\emptyset, 3\}$, $\{12, 123\}$ *and* $\{13, 34\}$*, and remove the edge* $\{13, 123\}$*, but according to Remark 1, we do not remove the edges* $\{1, 12\}$ *and* $\{3, 34\}$*. Thus, we obtain the required Hamiltonian cycle on the third line of Figure 1.11, where* $\emptyset$ *is adjacent to each of the ideals* 1 *and* 3.

Lemma 2. *If a poset* $\mathcal{P}$ *has at least two minimal elements, then there is a Hamiltonian cycle of down-sets in* $\mathcal{P}$ *in which two down-sets differ by one or two elements and such that* $\emptyset$ *is adjacent to each of these minimal elements. In addition, if two of these minimal elements are overlined, then each of the new posets obtained has only one overlined minimal element (it may have other minimal elements that are not overlined).*

Proof. (By induction on n, the number of minimal elements in $\mathcal{P}$).

Let x and y be two minimal elements in $\mathcal{P}$. By applying the generalized recurrence

1. $\mathcal{P} \xrightarrow{x,y} \mathcal{P}\setminus U[x,y] \quad + (x) \quad \mathcal{P}\setminus (U[y]\cup\{x\}) + (y) \quad \mathcal{P}\setminus (U[x]\cup\{y\}) \quad + (xy) \quad \mathcal{P}\setminus\{x,y\}$

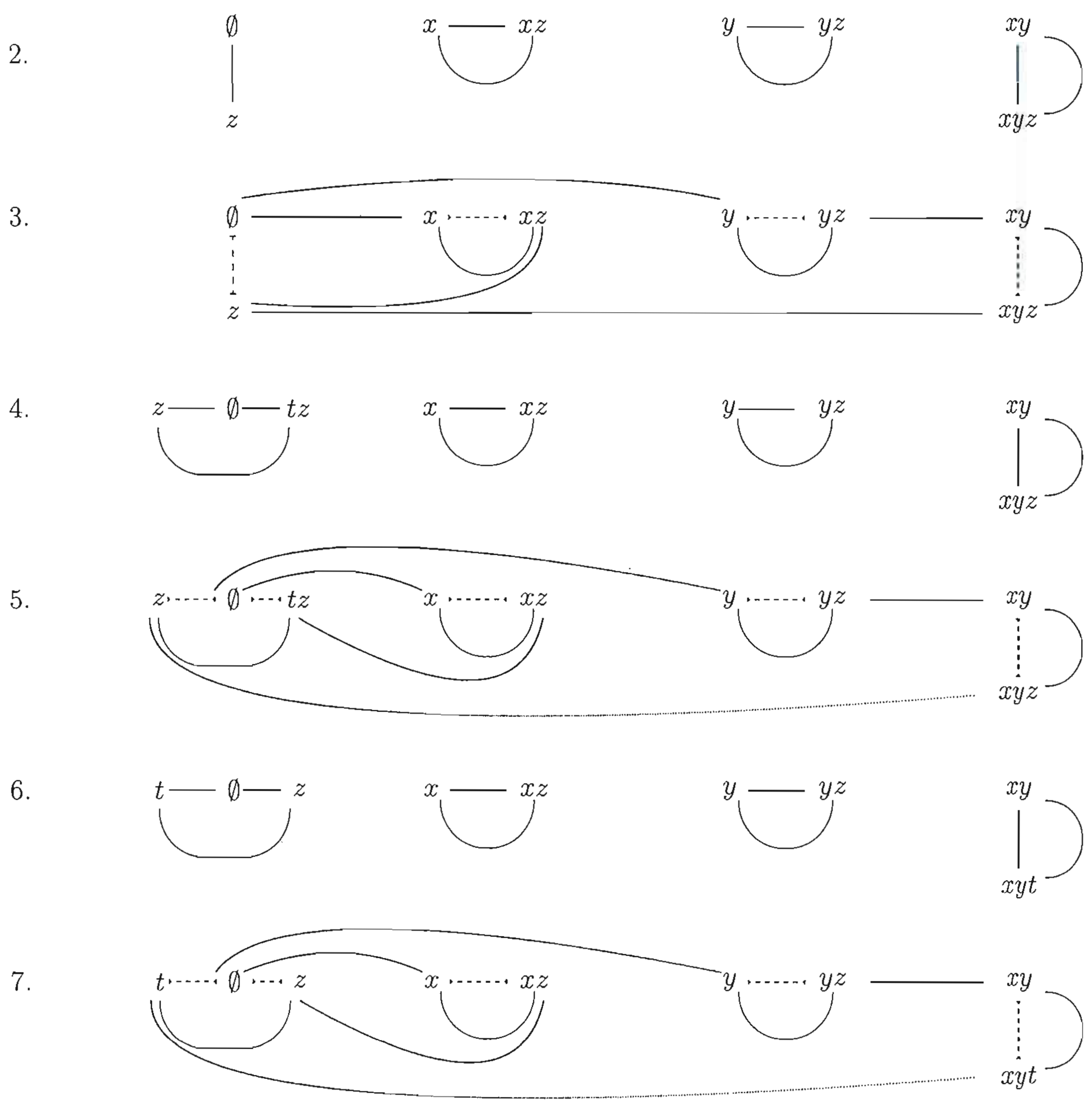


Figure 1.12. Illustration of Lemma 2. Lines 2-3 cover Case 1, lines 4-5 cover Case 2 and lines 6-7 cover Case 3.

(Equation 1.2), with initial values $D = \emptyset$ and $Y = E$ and the minimal elements x and y, we obtain the first line in Figure 1.12. We note that the first part is included in each of the second, third and fourth parts.

Consequently, if the first part has a minimal element, it will be also a minimal element of each of the other three parts.

Basic step. If $n = 2$, then the result follows from the proof of Lemma 1. Moreover, as we require min_1 to be adjacent to $\emptyset$ for the second and fourth parts and that min_2 be adjacent to $\emptyset$ for the third part (see Figure 1.10), each of these three parts will have an overlined minimal element according to Remark 2.

Induction step. If $n > 2$, then we have 3 cases.

1. The first part restricted to $Y \setminus U[x, y]$ contains a unique element z. Then there is a Hamiltonian cycle for this part that contains the edge $\{\emptyset, z\}$. Since z is a minimal element in each of the second, third and fourth parts, according to Theorem 1 these parts have the cycles indicated in line 2 of Figure 1.12. We note that one or more of the last three cycles may be reduced to one edge if the number of elements of the corresponding poset is restricted to one element. Consequently, the required Hamiltonian cycle of $\mathcal{P}$ is shown in the third line of Figure 1.12.

2. The first part restricted to $Y \setminus U[x, y]$ contains more than one element but exactly one minimal element z. This minimal element certainly has a cover t. Applying Theorem 1, Case 2, we obtain the first Hamiltonian cycle on the fourth line in Figure 1.12. The other Hamiltonian cycles on the same line are also obtained by applying Theorem 1 with z as median element. Consequently, the cycle of $\mathcal{P}$ is shown on the fifth line of the same figure.

3. The first part restricted to $Y \setminus U[x, y]$ has at least two minimal elements, say t and z. By the induction hypothesis there is a Hamiltonian cycle with $\emptyset$ adjacent to t and z as shown on the sixth line. The other Hamiltonian cycles on the same line are obtained as in Case 2 except that the median for the fourth part is t instead

of z. Finally, the cycle of $\mathcal{P}$ is shown on the seventh line of the same figure.

Analogously to the basic step, we can show that each of the second, third, fourth and possibly the first will have an overlined minimal. □

Figure 1.12 does not include overlines because it shows a general poset, and the overlines are a function of the Hasse diagram of a particular poset.

A situation is called *blocked* when we require $\emptyset$ to be adjacent to two elements, i.e. when we have 2 minimal elements, each of which is overlined. In such a situation, we use Lemma 2 to unblock this situation. Then we obtain new posets, each of which has one overlined minimal element. It follows that each time we use Lemma 2 (our generalization of Squire's recurrence) we return to the choice of median 3 or 4 times (depending upon the first part) where we apply Theorem 1, 2 or 3 as needed to the new posets.

Example 6. *Consider the poset on the left side of the first line of Figure 1.13. By choosing* 2 *and* 4 *for the minimal elements* x *and* y *and applying Lemma 2, we obtain the first line of Figure 1.13. The first part is the empty poset whose underlying set is* $\{1\}$*; consequently it has the Hamiltonian cycle* $\{\emptyset, 1\}$*. The second part is the empty poset whose underlying set is* $\{1, 3\}$*; therefore it has the Hamiltonian cycle* $\{\emptyset, 1, 13, 3\}$*. By adding the element* 2 *to each ideal of this last cycle we obtain the Hamiltonian cycle* $\{2, 12, 123, 23\}$*. The third part is the empty poset whose underlying set is* $\{1, 5\}$*; therefore it has the Hamiltonian cycle* $\{\emptyset, 1, 15, 5\}$*. By adding the element* 4 *to each ideal of this last cycle we obtain the Hamiltonian cycle* $\{4, 14, 145, 45\}$*. The fourth part is the empty poset whose underlying set is* $\{1, 3, 4\}$*; therefore by applying Theorem 1 and by taking* 1 *for the minimal element we can show that this poset possesses a Hamiltonian cycle having the edge* $\{\emptyset, 1\}$*. By adding the set* 24 *to each ideal of this last cycle we obtain a Hamiltonian cycle having the edge* $\{24, 124\}$*. The Hamiltonian cycle of each part is on the second line of Figure 1.13.*

Since we applied Lemma 2, we add the edges $\{\emptyset, 2\}$, $\{\emptyset, 4\}$, $\{1, 12\}$, $\{1, 124\}$ *and* $\{14, 24\}$,

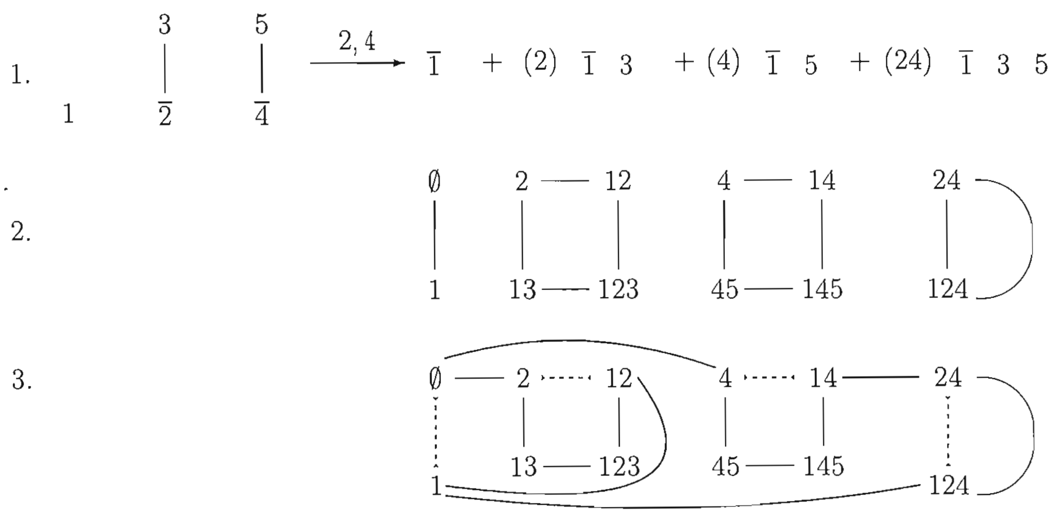


Figure 1.13. Example illustrating the use of Lemma 2.

and remove the edges $\{2, 12\}$, $\{4, 14\}$ *and* $\{24, 124\}$. *We finally remove, according to Remark 1, the edge* $\{\emptyset, 1\}$. *Thus we obtain the required Hamiltonian cycle on the third line of Figure 1.13, where* $\emptyset$ *is adjacent to each of the ideals* 2 *and* 4.

1.7 Hamiltonian cycle when there are at least two elements less than the median

The following Lemma 3 is a necessary step for the proof of Theorem 3.

Lemma 3. *Let* $\mathcal{P}$ *be the poset* $\mathcal{P} = \mathcal{Q} + \mathcal{R}$ *(disjoint union of two posets). Let* $\{x_1, x_2, \cdots, x_n\}$ *be the underlying set of* $\mathcal{Q}$*, where* $n \geq 2$*, and* x_n *be maximal. Then there is a Hamiltonian cycle of down-sets in* $\mathcal{P}$ *in which two adjacent down-sets differ by one or two elements and this cycle contains the edge* $\{x_1x_2\cdots x_{n-1}, x_1x_2\cdots x_n\}$.

Proof. (By induction on n).

Basic step. If $n = 2$, then we have 2 cases.

1. If x_2 is isolated, then both x_1 and x_2 are minimal elements. By applying equation

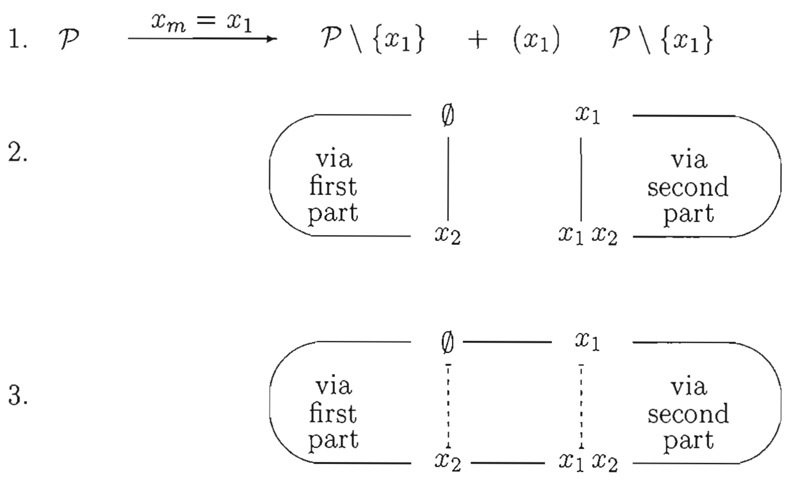


Figure 1.14. First case of the basic step in Lemma 3.

(1.1) with x_1 as median we obtain the first line of Figure 1.14. In this case, $\mathcal{P}\setminus\{x_1\}$ has a minimal element x_2. According to Theorem 1, the cycles of the first and second parts contain the edges $\{\emptyset, x_2\}$ and $\{x_1, x_1x_2\}$, respectively, that are on the second line of Figure 1.14. Also according to Theorem 1, Case 1, applied to $\mathcal{P}$ with x_1 as median, we add the edges $\{x_1, \emptyset\}$ and $\{x_1x_2, x_2\}$ and remove the edges $\{\emptyset, x_2\}$ and $\{x_1, x_1x_2\}$. Then we obtain the required Hamiltonian cycle on the third line of the same figure. We note that this cycle contains the edge $\{x_1, x_1x_2\}$.

2. If $x_1 < x_2$, then by applying equation (1.1) with x_1 as median we obtain the first line of Figure 1.15. We distinguish 2 cases.

   (a) If $\mathcal{P}\setminus U[x_1] = \emptyset$, then $\mathcal{P}\setminus\{x_1\} = \{x_2\}$. In this case the cycle of the first part is $\emptyset$ and the cycle of the second part contains the edge $\{x_1, x_1x_2\}$. According to Theorem 1, Case 2, applied to $\mathcal{P}$ with x_1 as median, we add the edges $\{\emptyset, x_1\}$ and $\{\emptyset, x_1x_2\}$ and, according to Remark 1, we do not remove the edge $\{x_1, x_1x_2\}$. Then we obtain the required Hamiltonian cycle on the second line of Figure 1.15. We note that this cycle contains the edge $\{x_1, x_1x_2\}$.

   (b) If $\mathcal{P}\setminus U[x_1] \neq \emptyset$, then it has at least one minimal element: we choose one of

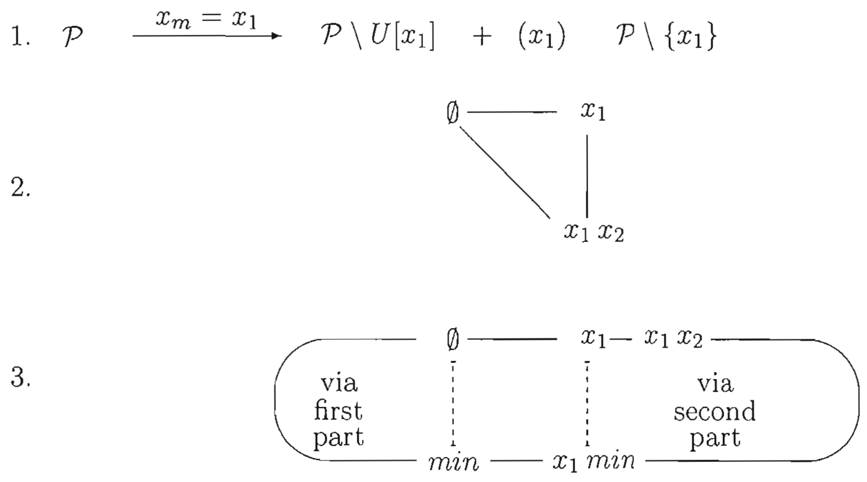

1. $\mathcal{P} \xrightarrow{x_m = x_1} \mathcal{P}\setminus U[x_1] \quad + \quad (x_1) \quad \mathcal{P}\setminus\{x_1\}$
2.
$\emptyset$
x_1
$x_1 x_2$
3.
via first part
$\emptyset$
x_1
$x_1 x_2$
via second part
min
$x_1 min$

Figure 1.15. Second case of the basic step in Lemma 3.

them and call it *min*. According to Theorem 1 with *min* as median, there is a Hamiltonian cycle containing the edge $\{\emptyset, min\}$. On the other hand, the second part has at least two minimal elements: *min* and x_2. By applying Theorem 1, Case 1, with x_2 as median, we obtain a Hamiltonian cycle in which $\emptyset$ is adjacent to *min* and to x_2. By adding x_1 to each down-set of this cycle we obtain a Hamiltonian cycle in which x_1 is adjacent to $x_1 min$ and x_1x_2. Since we applied Theorem 1, Case 1, to $\mathcal{P}$ with x_1 as median, we add the edges $\{\emptyset, x_1\}$ and $\{min, x_1min\}$ and remove the edges $\{\emptyset, min\}$ and $\{x_1, x_1min\}$. Finally, the required Hamiltonian cycle of $\mathcal{P}$ is on the third line of Figure 1.15. We note that this cycle contains the edge $\{x_1, x_1x_2\}$.

Induction step. If $n > 2$, then without loss of generality, x_n will not be chosen as median. In addition, we suppose that the median x_m is chosen among the elements $x_1x_2 \cdots x_{n-1}$. We distinguish three cases.

1. $\#(D[x_m]) = 2$, i.e. x_m covers a unique element, say x_{m-1}. Then this last element is minimal. By applying Theorem 2 with x_m as median we obtain the first line of Figure 1.16. The first part $\mathcal{P}\setminus U[x_m]$ has a minimal element x_{m-1}. Then, ac-

cording to Theorem 1, there is a Hamiltonian cycle containing the edge $\{\emptyset, x_{m-1}\}$. For the second part there are three cases.

(a) If $n = 3$ and E contains only 3 elements, then the cycle of the first part is the edge $\{\emptyset,\ x_1\}$ and the second part is the singleton $\{x_3\}$. Consequently x_3 is a minimal element in the second part. By applying Theorem 1 with x_3 as median we obtain the Hamiltonian cycle $\{x_1x_2,\ x_1x_2x_3\}$ since $D[x_m] = x_1x_2$. Since we applied Theorem 2 to $\mathcal{P}$, we add the edges $\{\emptyset,\ x_1x_2\}$ and $\{x_1,\ x_1x_2x_3\}$ according to Remark 1, we do not remove the edge $\{\emptyset,\ x_1\}$ nor the edge $\{x_1x_2,\ x_1x_2x_3\}$. Finally we obtain the required Hamiltonian cycle containing the edge $\{x_1x_2,\ x_1x_2x_3\}$ on the second line of Figure 1.16.

(b) If $n = 3$ and E contains more than three elements, then the cycle of the first part contains the edge $\{\emptyset,\ x_1\}$ and the second part has at least two minimal elements: x_3 and another one, say *min*. By applying Lemma 2 we obtain a Hamiltonian cycle containing the edges $\{x_1x_2,\ x_1x_2x_3\}$ and $\{x_1x_2,\ x_1x_2min\}$. Since we applied Theorem 2 to $\mathcal{P}$, we add the edges $\{\emptyset,\ x_1x_2\}$ and $\{x_1,\ x_1x_2min\}$ and remove the edges $\{\emptyset,\ x_1\}$ and $\{x_1x_2,\text{-}$ $x_1x_2min\}$. Finally we obtain the required Hamiltonian cycle containing the edge $\{x_1x_2,\ x_1x_2x_3\}$ on the third line of Figure 1.16.

(c) If $n > 3$, then the second part contains at least two elements from the set $\{x_1, x_2, \cdots, x_n\}$ and by the induction hypothesis there is a Hamiltonian cycle of down-sets containing the edge $\{x_1x_2 \cdots x_{m-2}x_{m+1} \cdots x_{n-1},\ x_1x_2 \cdots$ $x_{m-2}x_{m+1} \cdots x_n\}$. When we add to each down-set of the last cycle the set $D[x_m] = x_{m-1}x_m$, this cycle will contain the edge $\{\ x_1x_2 \cdots x_{n-1},\ x_1x_2 \cdots x_n\}$. In addition, when we apply Theorem 1 to the second part with the median, which we call *min*, this last cycle will contain the edge $\{x_{m-1}x_m,\ x_{m-1}x_mmin\}$. Since we applied Theorem 2 to $\mathcal{P}$, we add the edges $\{\emptyset,\ x_{m-1}x_m\}$ and $\{x_{m-1},\ x_{m-1}x_mmin\}$ and remove the edges $\{\emptyset,\ x_{m-1}\}$ and $\{x_{m-1}x_m,\ x_{m-1}x_m$- $min\}$. Finally we obtain the required Hamiltonian cycle containing the edge $\{x_1x_2 \cdots x_{n-1},\ x_1x_2 \cdots x_n\}$ on the fourth line of Figure 1.16.

2. If $\#(D[x_m]) = 1$, then when we apply Theorem 1 with x_m as median, the first part will either have a Hamiltonian cycle of down-sets containing the edge $\{\emptyset, min\}$, where min is a minimal element of the poset of the first part, or will have the cycle $\emptyset$, depending upon whether the first part is non-empty or empty. As for the second part, its underlying set contains at least two elements from the set $\{x_1, x_2, \cdots, x_n\}$. By the induction hypothesis, and by an argument similar to the one applied to the second part in Case 1 (c), there is a Hamiltonian cycle of down-sets containing the edge $\{x_1x_2\cdots x_{n-1},\ x_1x_2\cdots x_n\}$. Moreover, when we apply Theorem 1 to this second part with min as median, this last cycle will contain the edge $\{x_m, x_m min\}$. Finally, by applying Theorem 1 to $\mathcal{P}$, Case 1 or 2, with x_m as median, depending upon whether the first part is non-empty or empty, and adding and removing the appropriate edges we obtain the required Hamiltonian cycle containing the edge $\{\ x_1x_2\cdots x_{n-1},\ x_1x_2\cdots x_n\}$.

3. In this case $\#(D[x_m]) > 2$. By applying equation (1.1) to $\mathcal{P}$ with x_m as median, we obtain the first line of Figure 1.16. We suppose that x_m does not have any brothers (if x_m has a brother we choose as median another elements that has no brothers) and that $D[x_m] \setminus \{x_m\} = \{x_i, \cdots, x_{m-1}\}$, which must have a maximal element in $\mathcal{P}$, say x_r. Since $\#(D[x_m]) > 2$, the first part contains the set $\{x_i, \cdots, x_{m-1}\}$, which contains at least two elements. By the induction hypothesis there is a Hamiltonian cycle of down-sets containing the edge $\{x_i\cdots x_{r-1}\,x_{r+1}\cdots x_{m-1},\ x_i\cdots x_{m-1}\}$. By an argument similar to that of Case 1 above with all its subcases, there is a Hamiltonian cycle of down-sets for the second part containing both edges $\{x_1x_2\cdots x_{n-1},\ x_1x_2\cdots x_n\}$ and $\{x_i\cdots x_m,\ x_i\cdots x_m\,min\}$. Finally by adding the edges $\{x_i\cdots x_{r-1}\,x_{r+1}\cdots x_{m-1},$
$x_i\cdots x_m\}$ and $\{x_i\cdots x_{m-1},\ x_i\cdots x_m\,min\}$ and removing the edges $\{x_i\cdots x_{r-1}\,x_{r+1}$ $\cdots x_{m-1},\ x_i\cdots x_{m-1}\}$ and $\{x_i\cdots x_m,\ x_i\cdots x_m\,min\}$ we obtain the required Hamiltonian cycle containing the edge $\{\ x_1x_2\cdots x_{n-1},\ x_1x_2\cdots x_n\}$ on the fifth line of Figure 1.16. We note that if x_{n-1} is greater than each element of $\{x_1, \cdots, x_{n-2}\}$ and if we take x_{n-1} as median, then the vertex $x_1\cdots x_{n-1}$ will be reserved for an-

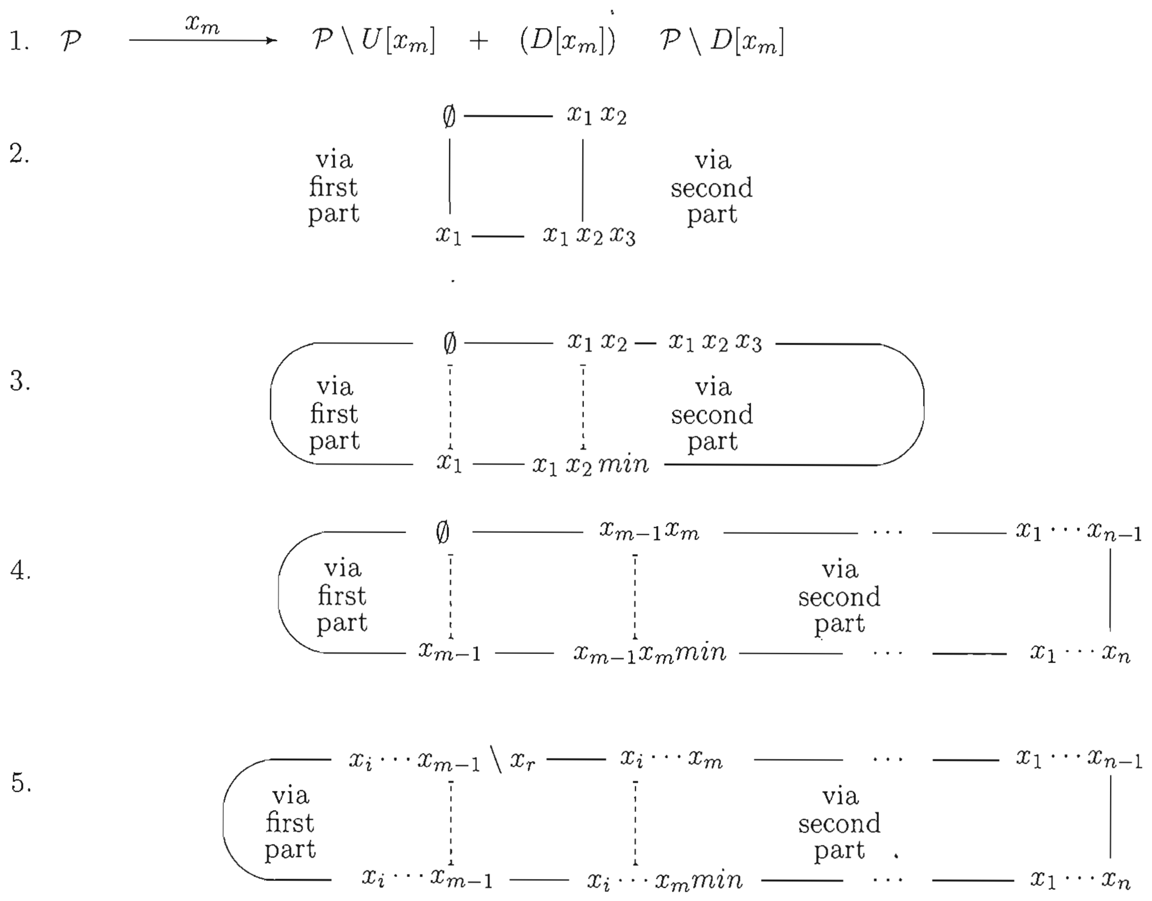


Figure 1.16. The induction step of Lemma 3.

other cycle containing the edge $\{x_1 \cdots x_{n-2},\, x_1 \cdots x_{n-1}\}$. Consequently, we avoid this choice in that case. $\square$

Remark 3. *In Lemma 3, we proved that there is a Hamiltonian cycle of down-sets of the poset satisfying Lemma 3's hypothesis, where two adjacent down-sets differ by one or two elements and this cycle contains the edge $\{x_1x_2 \cdots x_{n-1},\, x_1x_2 \cdots x_n\}$. When applying Squire's recurrence (Equation (1.1)) to this poset with the median x_m chosen among $\{x_1,\, x_2,\, \cdots x_n\}$ we noticed that the set $D[x_m]$ is added to each element of the cycle of the second part (see the first line of Figure 1.16), and induction on this second part enabled us to prove the existence of the edge $\{x_1x_2 \cdots x_{n-1},\, x_1x_2 \cdots x_n\}$. On the other hand, if* *we chose the median x_m outside the set $\{x_1, x_2, \cdots x_n\}$, the set $D[x_m]$ would be added to each element of the cycle of the second part, but we do not want $D[x_m]$ to be added to the elements of our cycle which will contain the edge $\{x_1x_2\cdots x_{n-1}, x_1x_2\cdots x_n\}$. Consequently, in this case this last edge is part of the cycle of the first part.*

Remark 4. *The choice of x_n as median is the last among the elements of the set $\{x_1, x_2, \cdots, x_n\}$. This last choice ensures the existence of the edge $\{\emptyset, x_n\}$ to which we must add the set $x_1x_2\cdots x_{n-1}$ built at this stage and this ensures the existence of the edge $\{x_1x_2\cdots x_{n-1}, x_1x_2\cdots x_n\}$ by creating it. In practice, we underline each element of the set $x_1x_2\cdots x_n$ (we draw a line under it) and these underlines will be transferred to the first part when x_m is chosen outside the set $\{x_1, x_2, \cdots, x_n\}$ and to the second part when it is chosen among the elements of this set so that x_n will be chosen after the elements $\{x_1, x_2, \cdots, x_{n-1}\}$.*

Remark 5. *Given the existence of the edge $\{x_1x_2\cdots x_{n-1}, x_1x_2\cdots x_n\}$ in Lemma 3, the edge $\{\emptyset, x_n\}$ must exist in a sub-cycle. Therefore, x_n, which is underlined, is also considered overlined. Consequently, this x_n cannot be chosen as median unless it is the unique element in the poset, since at this stage it loses its connection with $\emptyset$ and regains it when we add the edge $\{\emptyset, x_n\}$.*

Two elements are said to be *brothers* if there is a third one which is covered by each of them.

Theorem 3. *Let E be the underlying set of elements of the poset $\mathcal{P}$ of which x is an element having at least 2 elements less than it and does not have any brothers. There is a Hamiltonian cycle of down-sets of this poset where two adjacent down-sets differ by one or two elements and this cycle does not contain the edge $\{x_1x_2\cdots x_{k-1}, x_1x_2\cdots x_k\}$, where $x_1x_2\cdots x_kx = D[x]$ and x_k is maximal in $\mathcal{P}\setminus U[x]$.*

Proof. Let n be the cardinality of E ($n = \#(E)$). We have 2 cases.

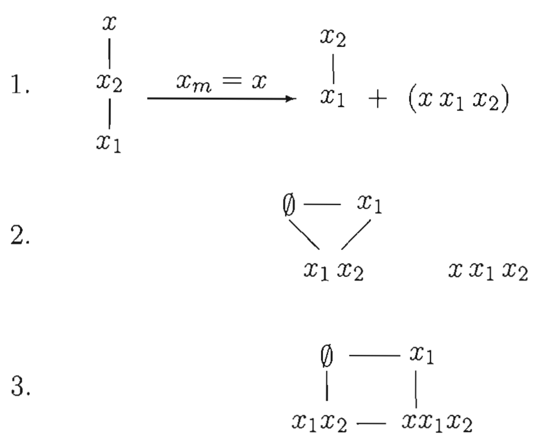


Figure 1.17. First subcase of first case in Theorem 3.

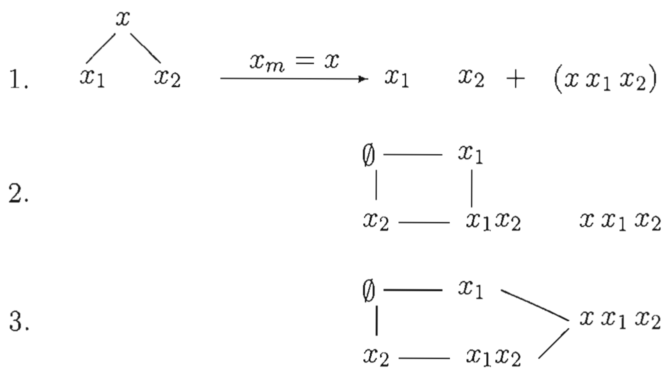


Figure 1.18. Second subcase of first case in Theorem 3.

1. In this case $n = 3$ and there are two elements less than x. We have 2 sub-cases, where the poset is the left side of the first line of either Figure 1.17 or Figure 1.18. By applying Equation (1.1) with x as median we obtain the first line of either Figure 1.17 or Figure 1.18. The Hamiltonian cycles of down-sets of first and second parts are on the second line of the same figure. By adding the edges $\{x\,x_1\,x_2,\, x_1\}$ and $\{x\,x_1x_2,\, x_1\,x_2\}$ and removing the edge $\{x_1,\, x_1\,x_2\}$ we obtain the required Hamiltonian cycle on the third line which does not contain the edge $\{x_1,\, x_1x_2\}$.

2. In this case $n > 3$. We suppose that $D[x] = x_1 \cdots x_k\, x$. By applying Equa-

1. $\mathcal{P} \xrightarrow{x_m = x} \mathcal{P} \setminus U[x] \;+\; (x_1 \cdots x_k\, x)\ \mathcal{P} \setminus D[x]$

2.

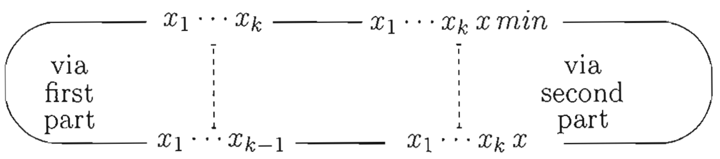


Figure 1.19. Second case of Theorem 3.

tion (1.1) we obtain the first line of Figure 1.19. The first part contains the elements $\{x_1, x_2, \cdots, x_k\}$, where $k \geq 2$, and by renaming them we may suppose that x_k is the greatest element covered by x in a linear extension of $\mathcal{P} \setminus U[x]$. Therefore, according to Lemma 3, there is a Hamiltonian cycle of down-sets of the first part, where two adjacent down-sets differ by one or two elements and this cycle contains the edge $\{x_1 x_2 \cdots x_{k-1},\ x_1 x_2 \cdots x_k\}$. Let min be a minimal element of the poset of the second part (this min does exist since x is a median and $n > 3$). By applying Theorem 1 to the second part with min as median we obtain a Hamiltonian cycle of down-sets of this poset, where two adjacent down-sets differ by one or two elements and this cycle contains the edge $\{\emptyset, min\}$. Since $x_1 \cdots x_k\, x$ is added to each down-set of this cycle, $\{x_1 \cdots x_k\, x,\ x_1 \cdots x_k\, x\, min\}$ is an edge of the cycle of the second part. By adding the edges $\{x_1 \cdots x_k\, x,\ x_1 \cdots x_{k-1}\}$ and $\{x_1 \cdots x_k\, x\, min,\ x_1 \cdots x_k\}$, which are similar to the edges added in Case 3 of the induction hypothesis of Lemma 3, and removing the edges $\{x_1 \cdots x_{k-1},\ x_1 \cdots x_k\}$ and $\{x_1 \cdots x_k\, x,\ x_1 \cdots x_k\, x\, min\}$ we obtain the required Hamiltonian cycle on the second line of Figure 1.19 which does not contain the edge $\{x_1 x_2 \cdots x_{k-1},\ x_1 x_2 \cdots x_k\}$. □

Remark 6. *In Theorem 3 we supposed that x had no brothers. If it happens that x has at least two elements less than it and that x has a brother, then we take as median one of the elements covered by x which does not have any brothers or for which the number of elements less than it is less than two. This is due to the difficulty of finding* *a Hamiltonian cycle containing the edge* $\{x_1x_2\cdots x_{n-1},\, x_1x_2\cdots x_n\}$ *in this last case.*

Example 7. *Consider the poset on the left side of the first line of Figure 1.20. The median is* 4, *which can not be chosen according to Remark 6; therefore we choose* 3, *which is a an element covered by* 4 *and which has no brothers. Since* $\#(D[3]) > 2$, *we apply Theorem 3 and we obtain the first line of Figure 1.20. We note that* 1 *and* 2 *on this line are each underlined and* 2 *is overlined, according to Theorem 3. The edges that must be added in this case are* $\{123,\, 1\}$ *and* $\{1234,\, 12\}$. *The second part has an overlined minimal element,* 4, *which ensures the existence of the edge* $\{123,\, 1234\}$ *according to Theorem 1. Consider the first part. Its median is* 1, *which is a minimal element. Then by applying Theorem 1 we obtain the second line of Figure 1.20. The cycle corresponding to each part as well as the cycle linking them, which is the cycle of the first part of the first line, is shown on the third line. We now consider the second part of the first line. Its median is* 5, *which is a minimal element, and it has another minimal element,* 4, *which is overlined. Therefore, the second part has two minimal elements, each of which is overlined. By applying Lemma 1 we obtain the fourth line. The cycle of each part, as well as the cycle linking them, which is the cycle of the second part of the first line, is shown on the fifth line. We note that the first cycle on the third line contains the edge* $\{1,\, 12\}$ *and the second cycle on the fifth line contains the edge* $\{\emptyset,\, 4\}$. *Since the set* 123 *is added to each down-set of the last cycle, this cycle contains the edge* $\{123,\, 1234\}$. *By adding the edges of Theorem 3 announced above,* $\{123,\, 1\}$ *and* $\{1234,\, 12\}$, *and removing the edges* $\{1,\, 12\}$ *and* $\{123,\, 1234\}$ *we obtain the required Hamiltonian cycle of our poset on the sixth line.*

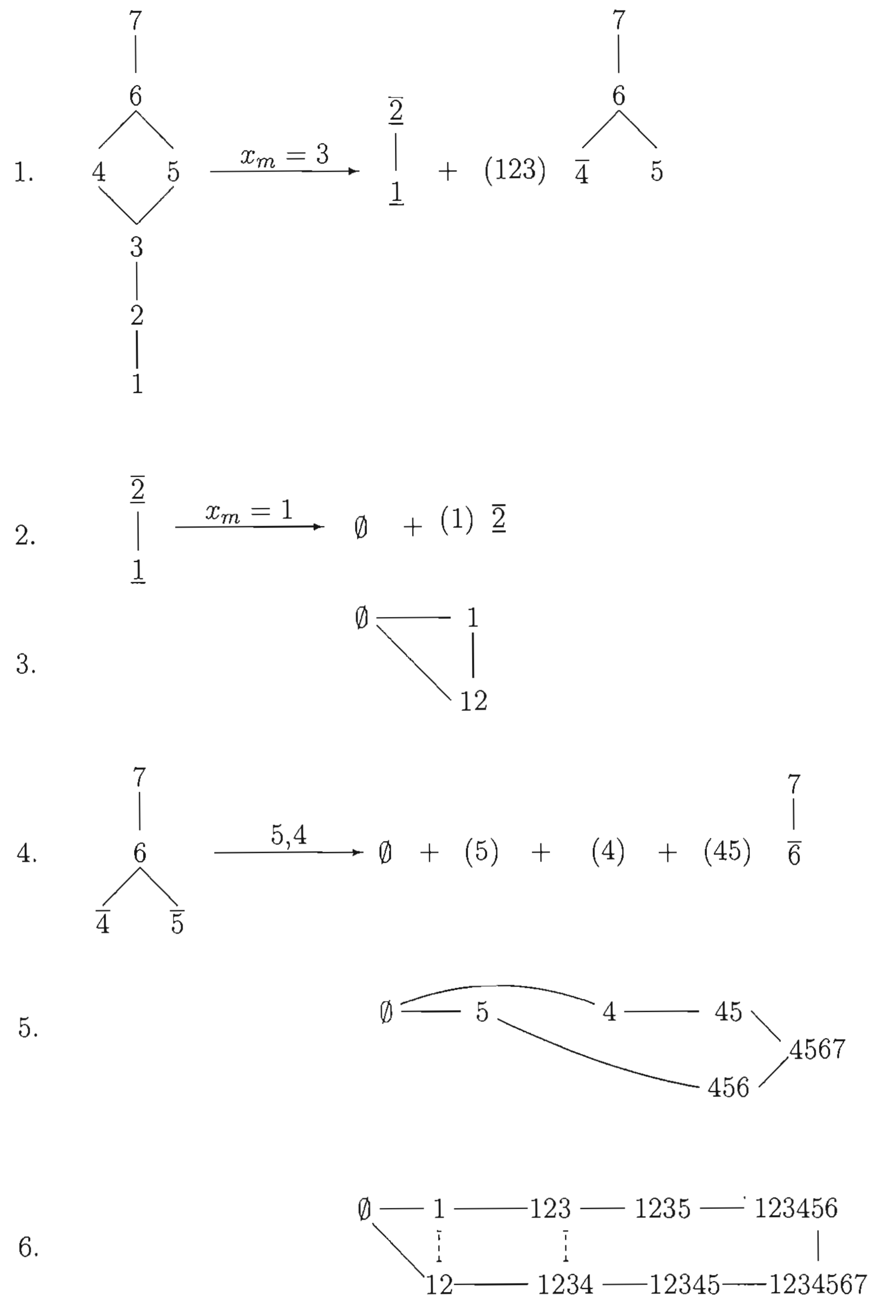


Figure 1.20. Example illustrating the use of Theorems 1 and 3 as well as Lemma 1.

CHAPTER II

ALGORITHMS

2.1 Representation and useful functions

A subset S of the set $\{x_0, x_1, \ldots, x_{n-1}\}$ is represented by the binary number

$$b_{n-1}b_{n-2}\cdots b_1b_0,$$

where $b_i = 1$ if i belongs to S and 0 otherwise. For example, 0111010 represents the set $\{1, 3, 4, 5\}$. The language C++ possesses two operators, left shift "<<" and right shift ">>", which make it easier to add or to remove an element from a set. The functions unsigned *getbit* (unsigned int *word*, int n) and unsigned *setbit* (unsigned int *word*, int n, unsigned v) make use of these two operators; the first makes it possible to check whether n is a member of the set represented by *word* and the second adds n to or removes n from the set represented by *word*. In the algorithms below, we use "$\cup$" and "$\backslash$" instead of *setbit*.

When we enter data (the number of elements of the underlying set and the relations between its elements), the relations are stored in tables, one for each element, in the following sense. *Table*$[i]$ = *word* if i is less than or equal to each integer represented by *word*. For example, *table*[1]=010110 means 1=1, $1 < 2$ and $1 < 4$. The function void *Process*() gives the transitive closure of this relation so that the tables will be filled in by the appropriate element(s). We copy *table*$[i]$ to *up*$[i]$ and use this later in all instructions of our programs. *Down*$[i]$ will contains the elements which are less than or equal to i. We also set *nup*$[i] = \#($*up*$[i])$ and *ndown* $= \#($*down*$[i])$.

The function int *Updateup* (int *poset* int *min*, int *num*) updates all the tables *down* and the numbers *ndown* of the elements remaining in the poset, and finally the number of minimal elements *num*, after removing *down*[*min*] from poset. The function int *Recoverup* (int *poset*, int *min*, int *num*) has the reverse effect of *Updateup*; i.e. it returns back the values of the tables *down*, and the numbers *ndown* and the number of minimal elements *num*, as they were before removing *down*[*min*] from *poset* and calling *Updateup*.

The functions int *Updatedown* (int *poset*, int *min*, int *num*) and int *Recoverdown* (int *poset*, int *min*, int *num*) have the same role as the two previous functions but instead of removing *down*[*min*] they remove *up*[*min*] and they deal with the tables *up* and the numbers *nup* instead of dealing with the tables *down* and the numbers *nup*.

2.2 Printing ideals

We opt to print the ideals of a poset when it is not empty. Since we have one empty ideal, we print it at the beginning. This is possible because Theorem 1 states that the cycle contains an edge $\{\emptyset, x\}$, where x is a minimal element chosen as median, and Theorem 2 states that the cycle contains an edge $\{\emptyset, x_1 x\}$, where x is the chosen median and $x_1 < x$. So, in these two cases, $\emptyset$ starts or ends the cycle and printing it at the beginning in all cases does not affect the cycle of ideals. As for Theorem 3, it inserts the cycle of the second part into the cycle of the first part and continues to do so until we can apply Theorem 1 or Theorem 2 to the first part. Consequently, printing $\emptyset$ always at the beginning is a good idea. When it must be printed at the end, we can shift back the cycle by one ideal so that $\emptyset$ is now at the end.

2.2.1 Order of printing the ideals when applying Theorem 1

When x is a minimal element chosen as median, we apply Theorem 1 to the poset $\mathcal{P}$ and we obtain Figure 1.2. We have 2 cases.

1. The poset has more than one minimal element. If the direction is forward, we must start by printing x. Since x will be added to each ideal of the cycle of the second part (see Figure 1.3), we print the cycle of the second part in the backward direction by applying Theorem 1 to this part with a minimal element *min* chosen as median and then we print the cycle of the first part in the forward direction by applying Theorem 1 with *min* as chosen median. If the direction is backward, we reverse the previous order, i.e. we print the cycle of the first part in the backward direction by applying Theorem 1 with a minimal element *min* chosen as median and then we print the cycle of the second part in the forward direction by applying Theorem 1 to this part with *min* as chosen median, and finally we print x.

2. We use the same process as in Case 1, but here the first part is the empty poset (see Figure 1.4).

2.2.2 Order of printing the ideals when applying Theorem 2

When x has one element less than it and this x is chosen as median, we apply Theorem 2 to the poset $\mathcal{P}$ and we obtain the first line of Figure 1.8. If the direction is forward, we must start by printing x_1x. Since x_1x will be added to each ideal of the cycle of the second part (see the second line of Figure 1.8), we print the cycle of the second part in the backward direction by applying Theorem 1 to this part with a minimal element *min* chosen as median and then we print the cycle of the first part in the forward direction by applying Theorem 1 with x_1, which is a minimal element, chosen as median. If the direction is backward, we reverse the previous order; i.e. we print the cycle of the first part in the backward direction by applying Theorem 1 with x_1 a minimal element chosen as median and then we print the cycle of the second part in the forward direction by applying Theorem 1 to this part with a minimal element *min* chosen as median, and finally we print x_1x. We note that the second part may be an empty poset (see Figure 1.7); in this case the recursive call to this poset gives nothing.

2.2.3 Order of printing the ideals when applying Theorem 3

When x has more than one element less than it and is chosen as median, we apply Theorem 3 to the poset $\mathcal{P}$ and we obtain the first line of Figure 1.19. When we apply Theorem 1 or Theorem 2, we start the cycle or end it by the minimal element x or by x_1x, where x is the median and x_1 is less than x, and these vertices are incident to the edges that we remove to make the connection with others cycles. So, when we apply Theorem 1 or Theorem 2, we can start or end the cycle with the vertices incident to the edges which must be removed, whereas when we apply Theorem 3, the first cycle is obtained by applying Theorem 1 or Theorem 2, but the vertices that are incident to the edge that must be removed are not necessarily those by which we start or end the first cycle (see the second line of Figure 1.19). So, while enumerating the first cycle, when we find the first vertex incident to the edge that must be removed, we enqueue the rest of the cycle in a queue, then enumerate the second cycle, and at the end we dequeue the ideals from the queue. This case becomes more complicated when we have many nested levels.

2.3 Use of Lemma 1 and Lemma 2

When we apply Theorem 1 to a poset which has an overlined minimal element and this element is not chosen as median, we must apply Lemma 1 or Lemma 2 depending on whether the number of minimal elements is two or more.

The above discussion should help the reader to program the algorithm given below. There are some special cases whose implementation depends on the analysis of the programmer.

2.4 Algorithms

In the next figures, we present rough outlines of our algorithm as well as that of the Pruesse-Ruskey algorithm and the Squire algorithm. The procedure "*Ideal*" calls each of the other procedures *Ideal1*, *Ideal2* and *Ideal3* which respectively code Theorem 1, Theorem 2 and Theorem 3. It also calls the procedures *lemma1* and *lemma2*, which respectively code Lemma 1 and Lemma 2, in the particular cases described above.

The initial call is $Ideal(\mathcal{P}, dir, 0, \emptyset)$, where *dir* may be chosen to be either *FORWARD* or *BACKWARD*.

The reader may find all these details and many others in the complete program in Appendix A.1.

```
Procedure Ideal1(poset $\mathcal{P}$, direction dir, int min, ideal D) {The meaning of $\mathcal{P}$, min and}
begin                                                        {D is the same as in the text}
  if ($\mathcal{P} \neq \emptyset$) then
    if (num=1) then                                          {num is the number of}
        begin                                                {minimal elements of the poset $\mathcal{P}$}
            $\mathcal{P} := \mathcal{P} \backslash \{$min$\}$;
            D:=D $\cup$ {min};
            if (dir=FORWARD) then begin print(D); dir1:=BACKWARD; end;
            else dir1:=FORWARD;
            Updateup($\mathcal{P}$);                        {Updates $\mathcal{P}$ after removing D[min]}
            min:=findMinimal($\mathcal{P}$);                {findMinimal finds a minimal}
            Ideal($\mathcal{P}$, dir1, min, D);             {element of $\mathcal{P}$}
            if (dir=BACKWARD) then print(D);
            Recoverup($\mathcal{P}$);        {Restores $\mathcal{P}$ to its status before the call to Updateup}
        end else     {num > 1}
        begin
            $\mathcal{P} := \mathcal{P} \backslash \{$min$\}$;
            min1:=findMinimal($\mathcal{P}$);
            D1:=D $\cup$ {min};
            if (dir=FORWARD)
            begin
                print(D1);
                Updateup($\mathcal{P}$);
                Ideal($\mathcal{P}$, BACKWARD, min1, D1);
                Recoverup($\mathcal{P}$);
                Updatedown($\mathcal{P}$);                  {Updates $\mathcal{P}$ after removing U[min]}
                Ideal($\mathcal{P}$\U[min], dir, min1, D);
                Recoverdown($\mathcal{P}$);  {Restores $\mathcal{P}$ to its status before the call to Updatedown}
            end else     {dir=BACKWARD}
            begin
                Updatedown($\mathcal{P}$);                  {Updates $\mathcal{P}$ after removing U[min]}
                Ideal($\mathcal{P}$\U[min], FORWARD, min1, D);
                Recoverdown($\mathcal{P}$);                 {Restores $\mathcal{P}$ to its status before}
                Updateup($\mathcal{P}$);
                Ideal($\mathcal{P}$, dir, min1, D1);
                Recoverup($\mathcal{P}$);
                print(D1);
            end;
        end;
end;
```

Figure 2.1. Algorithm coding Theorem 1.

```
Procedure Ideal2(poset $\mathcal{P}$, direction dir, int min, ideal D)
begin
  if ($\mathcal{P} \neq \emptyset$) then
  begin
     D1:=D[min] $\cup$ D;
     if (dir=FORWARD) then
     begin
          Updatedown($\mathcal{P}$);
          min1:=findMinimal(D[min]);
          Ideal($\mathcal{P}$\U[min], BACKWARD, min1, D);
          Recoverdown($\mathcal{P}$);
          Updateup($\mathcal{P}$);
          min1:=findMinimal($\mathcal{P}$\D[min]);
          Ideal($\mathcal{P}$\D[min], FORWARD, min1, D1);
          print(D1);
          Recoverup($\mathcal{P}$);
     end else      {dir=BACKWARD}
     begin
          print(D1);
          Updateup($\mathcal{P}$);
          min1:=findMinimal($\mathcal{P}$\D[min]);
          Ideal($\mathcal{P}$\D[min], BACKWARD, min1, D1);
          Recoverup($\mathcal{P}$);
          Updatedown($\mathcal{P}$);
          min1:=findMinimal(D[min]);
          Ideal($\mathcal{P}$\U[min], FORWARD, min1, D);
          Recoverdown($\mathcal{P}$);
     end;
  end;
end;
```

Figure 2.2. Algorithm coding Theorem 2.

```
Procedure Ideal3(poset $\mathcal{P}$, direction dir, int min, int x, ideal D)
begin
  if ($\mathcal{P} \neq \emptyset$) then
  begin
     Updatedown($\mathcal{P}$);
     Ideal($\mathcal{P}$\U[min], dir, x, D);          {When we find the first vertex incident to the}
                                                   {edge that must be removed, we enqueue the}
                                                   {rest of the cycle.}
     Recoverdown($\mathcal{P}$);
     Updateup($\mathcal{P}$);
     min1:=findMinimal($\mathcal{P}$\D[min]);
     dir1:=direction of the last call to Ideal3
     if (dir1= BACKWARD) then print(D[min] $\cup$ D);
     Ideal($\mathcal{P}$\D[min], dir1, min1, D[min] $\cup$ D);
     if (dir1= FORWARD) then print(D[min] $\cup$ D);      {Now we dequeue the ideals.}
     Recoverup($\mathcal{P}$);
  end;
end;
```

Figure 2.3. Algorithm coding Theorem 3.

```
Procedure Ideal(poset $\mathcal{P}$, direction dir, int x, ideal D)  {x represents an overlined minimal }
begin                                        {element i.e. an element which must be adjacent to $\emptyset$.}
  if ($\mathcal{P} \neq \emptyset$) then                          {If such element does not exist, x will be zero.}
  begin
     min:=findMedian($\mathcal{P}$);                  {This function finds the median elements of $\mathcal{P}$.}
     if #(D[min])=1 then
          Ideal1($\mathcal{P}$, dir, min, D);
     else if #(D[min])=2 then
          Ideal2($\mathcal{P}$, dir, min, D);
     else
          Ideal3($\mathcal{P}$, dir, min, x, D) ;
  end;
end.
```

Figure 2.4. Our algorithm for generating the ideals of a poset.

```
**Procedure** lemma1(poset $\mathcal{P}$, int x, int y, ideal D)
int min1:=0, min2;
poset $\mathcal{P}$1; idea D1;
begin
  if ($\mathcal{P} \neq \emptyset$) then
  begin
     $\mathcal{P}$1 := $\mathcal{P}$\(U[y] ∪ {x});               {First cycle}
     D1:=D ∪ {x};
     Update($\mathcal{P}$);
     print(D1);
     Ideal($\mathcal{P}$1, BACKWARD, min1, D1);
     Recover($\mathcal{P}$);
     $\mathcal{P}$1 := $\mathcal{P} \setminus \{x, y\}$;                   {Second cycle}
     D1:=D ∪ {x,y};
     Update($\mathcal{P}$);
     min1:=findMinimal($\mathcal{P}$1);
     Ideal($\mathcal{P}$1, FORWARD, min1, D1);
     print(D1);
     Recover($\mathcal{P}$);
     $\mathcal{P}$1 := $\mathcal{P}$\(U[x] ∪ {y});               {Third cycle}
     D1:=D ∪ {y};
     Update($\mathcal{P}$);
     min2:=findMinimal($\mathcal{P}$1);
     Ideal($\mathcal{P}$1, FORWARD, min2, D1);
     print(D1);
     Recover($\mathcal{P}$);
  end;
end;
```

Figure 2.5. Algorithm coding Lemma 1.

```
Procedure lemma2(poset $\mathcal{P}$, int x, int y, ideal D)
int z, t1=0, num1:=num;                {num is the number of minimal elements of $\mathcal{P}$}
poset $\mathcal{P}$1; ideal D1;
begin
  if ($\mathcal{P} \neq \emptyset$) then
  begin
     $\mathcal{P}$1 := $\mathcal{P}\backslash$U[y];
     z:=findMinimal($\mathcal{P}$1\ U[x]);
     if (num > 3) then t1 := findMinimal($\mathcal{P}$1\(U[z] $\cup$ U[x]))
     $\mathcal{P}$1 := $\mathcal{P}$1\{x};                {First cycle}
     D1:=D $\cup$ {x};
     Update($\mathcal{P}$);
     print(D1);
     Ideal($\mathcal{P}$1, BACKWARD, z, D1);
     Recover($\mathcal{P}$);
     $\mathcal{P}$1 := $\mathcal{P}$1\ (U[x] $\cup$ U[y]);        {Second cycle}
     Update($\mathcal{P}$);
     if (num1=3) then
     begin
          $\mathcal{P}$1 := $\mathcal{P}$1\{z};
          Update($\mathcal{P}$);
          t:=findMinimal($\mathcal{P}$1);
          D1:=D $\cup$ {z};
          Ideal($\mathcal{P}$1, FORWARD, t, D1);
          print(D1);
          Recover($\mathcal{P}$);
          t:=0;
     end else if (num1=4) then lemma1($\mathcal{P}$1, z, t1, D);
     else lemma2($\mathcal{P}$1, dir, z, t1, D);
     Recover($\mathcal{P}$);
     $\mathcal{P}$1 := $\mathcal{P}$\{x,y};               {Third cycle}
     D1:=D $\cup$ {x,y};
     Update($\mathcal{P}$);
     if (t1) then Ideal($\mathcal{P}$1, FORWARD, t1, D1);
     else Ideal($\mathcal{P}$1, FORWARD, z, D1);
     print(D1);
     Recover($\mathcal{P}$);
     $\mathcal{P}$1 := $\mathcal{P}$\ (U[x] $\cup$ {y});         {Fourth cycle}
     D1:=D $\cup$ {y};
     Update($\mathcal{P}$);
     Ideal($\mathcal{P}$1, FORWARD, z, D1);
     print(D1);
     Recover($\mathcal{P}$);
     end;
  end;
end;
```

Figure 2.6. Algorithm coding Lemma 2.

```
Procedure Ideal(poset $\mathcal{P}$, direction dir, ideal D)
begin
    if ($\mathcal{P} \neq \emptyset$) then
    begin
            min:=findMinimal($\mathcal{P}$);
            if (num = 1) then       {num is the number of minimal elements of poset}
            begin
                $\mathcal{P} := \mathcal{P}\backslash\{$min$\}$;
                Update($\mathcal{P}$);
                D:=D $\cup$ {x};
                print(D);
                Ideal($\mathcal{P}$, dir, D);
                print(D);
                Recover($\mathcal{P}$);
            end else
            begin
                $\mathcal{P}1 := \mathcal{P}\backslash\{$min$\}$;
                D1:=D $\cup$ {min};
                if (dir = FORWARD) then
                begin
                    print(D1);    {In print, there is a parameter which makes the}
                    print(D1);    {procedure print D1 one time every two calls.}
                    Update($\mathcal{P}1$);
                    Ideal($\mathcal{P}1$, BACKWARD, D1);
                    Recover($\mathcal{P}1$);
                    $\mathcal{P} := \mathcal{P}\backslash$U[min];
                    Ideal($\mathcal{P}$, FORWARD, D);
                end else
                begin
                    $\mathcal{P} := \mathcal{P}\backslash$U[min];
                    Ideal($\mathcal{P}$, BACKWARD, D);
                    Update($\mathcal{P}1$);
                    Ideal($\mathcal{P}1$, FORWARD, D1);
                    Recover($\mathcal{P}1$);
                    print(D1);    {In print, there is a parameter which makes the}
                    print(D1);    {procedure print D1 one time every two calls.}
                end;
            end;
    end;
end;
```

Figure 2.7. Pruesse and Ruskey's algorithm for generating a Gray code for the ideals of a poset.

```
Procedure Ideal(poset $\mathcal{P}$, ideal D)
begin
  if ($\mathcal{P} = \emptyset$) then
     print(D);
  else begin
     min:=findMedian($\mathcal{P}$);          {This function finds the median element of $\mathcal{P}$.}
     D1:=D ∪ U[min];
     Ideal($\mathcal{P}$\U[min], D);
     Ideal($\mathcal{P}$\D[min], D1);
  end;
end;
```

Figure 2.8. Squire's algorithm for generating the ideals of a poset.

2.5 How to choose the median element

2.5.1 Relaxed condition

Squire (see (Squire)) gave a recurrence relation for the ideals (Equation 1.1) which we recall here:

$$\mathcal{D}(\mathcal{P}, D, Y) = \mathcal{D}(\mathcal{P}, D, Y \setminus U[x]) \cup \mathcal{D}(\mathcal{P}, D \cup D[x], Y \setminus D[x]),$$

where $\mathcal{P}$ is a poset on a set E, D is a down-set of $\mathcal{P}$, Y is a subset of E such that $D \cap Y = \emptyset$ and $D \cup Y$ is a down-set, $\mathcal{D}(\mathcal{P}, D, Y)$ is the set of down-sets of $\mathcal{P}$ that contain the down-set D and whose other elements are chosen from Y, x is some element of Y, and $D[x]$ and $U[x]$ are, respectively, the elements of $\mathcal{P}$ that are $\leq x$ and $\geq x$. In what follows we call $\mathcal{D}(\mathcal{P}, D, Y \setminus U[x])$ the *first part* and $\mathcal{D}(\mathcal{P}, D \cup D[x], Y \setminus D[x])$ the *second part*. Usually we consider $\mathcal{D}(\mathcal{P}, D, Y \setminus D[x])$ as the second part and then we add $D[x]$ to each ideal of this part.

He also gave an algorithm (see Figure 2.8), based on this recurrence, for enumerating the ideals of a poset in an amortized time of $O(\log n)$ per ideal. To achieve this complexity, he chose x for the median element x_m in the above recurrence. We explain this choice by Figure 2.9. In what follows we call $\mathcal{P} \setminus U[x]$ the poset of the first part and $\mathcal{P} \setminus D[x]$

$$\mathcal{P} \xrightarrow{x_m = x} \mathcal{P} \setminus U[x] \;\; + (D[x])\,\mathcal{P} \setminus D[x]$$

Figure 2.9. Choosing x_m for the median element in a linear extension of $\mathcal{P}$.

the poset of the second part.

Squire (see (Squire)) shows that by choosing the median element x_m, the number of elements of the poset of the first part and that of the second part are each less than or equal to $n-1$ and greater than or equal to $\frac{n}{2}$, where n is the number of elements of E. He proved that these inequalities ensure the enumeration of the ideals of a poset in an amortized time of $O(\log n)$ per ideal.

In the case of down-sets, a minimal element that has at most $\frac{n}{2}$ elements greater than it satisfies the above inequalities. We define the relaxed condition as follows.

Definition 1. *A minimal element x satisfies the relaxed condition if*

$$|U[x]| - 1 \leq \lfloor \frac{n}{2} \rfloor.$$

Consequently, every element satisfying the above definition satisfies Squire's inequalities. Thus, when we have to choose a minimal element to make a link between each cycle of the two parts, as in Theorem 1, if this element satisfies the relaxed condition it will be taken as median and every minimal element which satisfies this condition will be considered as median.

We prove in the next sections that in all cases of the algorithm either we choose a median or we overline a minimal element to choose it later when it becomes a median. There is still a problem we have to resolve: in Lemma 3 and consequently in Theorem 3, we excluded the case where a median has at least two elements less than it and has a brother.

$$\mathcal{P} \xrightarrow{x_m = x} \mathcal{P} \setminus U[x] \quad + \quad (x) \quad \mathcal{P} \setminus \{x\}$$

Figure 2.10. Left side is the poset having x as a minimal element.

2.5.2 Choosing the median for Theorem 1

If the median is a minimal element, say x, we apply Theorem 1 to the poset $\mathcal{P}$ and we obtain Figure 2.10. We have two cases.

1. The number of minimal elements of the poset of the first part is greater than 1.

   (a) If one of these minimal elements, say *min*, satisfies the relaxed condition, we choose it as median for the poset of the first part. Since the poset of the first part is included in the poset of the second part, $\mathcal{P} \setminus U[x] \subset \mathcal{P} \setminus \{x\}$, and the set $U[min]$ is the same in each of the two parts, this *min* also satisfies the relaxed condition for the poset of the second part and will be taken as median for this poset.

   (b) If none of these minimal elements satisfies the relaxed condition, the median of the poset of the first part, say *med*, is not a minimal element. However there is a minimal element, say *min*, less than this median, i.e. belonging to $D[med]$, and we overline this element. In the case of several minimal elements, say min_1, min_2, $\cdots$, min_k, we choose one of them, say *min*, so that $\#(U[min]) \leq \#(U[min_i]$, where $1 \leq i \leq k$, and that is not the maximum of $D[med] \setminus \{med\}$ in a linear extension of $\mathcal{P}$ (see Remark 5). This choice ensures that this minimal element will satisfy, in a later step, the relaxed condition before the other minimal elements. By an argument similar to that of the previous case, the median *med* of the poset of the first part will also be the median of the poset of the second part. Thus we overline this minimal element *min* for this last poset. Consequently, for each of the two posets, we overline a minimal element belonging to $D[med]$, where *med* is the median of the poset of the first part. We note that overlining an element postpones the choice of this element as a median until it satisfies the relaxed condition.

2. The number of minimal elements of the poset of the first part is 1. We follow the same procedure as that of the above part b.

Example 8. *Consider the poset on the left side of the first line of Figure 2.11. Every minimal element satisfies the relaxed condition, since the number of elements which are greater than each of these minimal elements is less than or equal to 2, which is half the number of elements of the poset. We take* 1 *as median. By applying Theorem 1 to this poset with 1 as median we obtain the first line of Figure 2.11. The minimal element* 4 *of the poset of the first part of the first line satisfies the relaxed condition. By applying Theorem 1 to this poset with* 4 *as median, we obtain the first cycle on the second line, which is similar to the cycle obtained on the third line of Figure 1.6. The cycle of the poset of the second part of the first line is obtained by adding* 3 *to each down-set of the precedent cycle and is the second one on the second line. Since we applied Theorem 1 with* 1 *as median, we add the edges* $\{1, \emptyset\}$ *and* $\{14, 4\}$ *to the last two cycles and remove the edges* $\{\emptyset, 4\}$ *and* $\{1, 14\}$ *from them. Consequently, we obtain the cycle on the third line of the poset on the left member of the first line.*

2.5.3 Choosing the median for Theorem 2

We apply Theorem 2 when there is exactly one element less than the median; so taking this minimal element as median will not affect the choice of the median since the difference will consist of one element to each part in contrast with the previous choice. Although not including this theorem does not greatly affect the performance of the algorithm, including it is necessary because it makes possible the processing of several special cases.

2.5.4 Choosing the median for Theorem 3

If there are at least two elements less than the median x, we apply Theorem 3. Then we obtain Figure 2.9, where $\#(D[x]) \geq 3$ since it includes x. If the poset $\mathcal{P}$ has a minimal element whether or not it is overlined (if it has an overlined element, it had been chosen

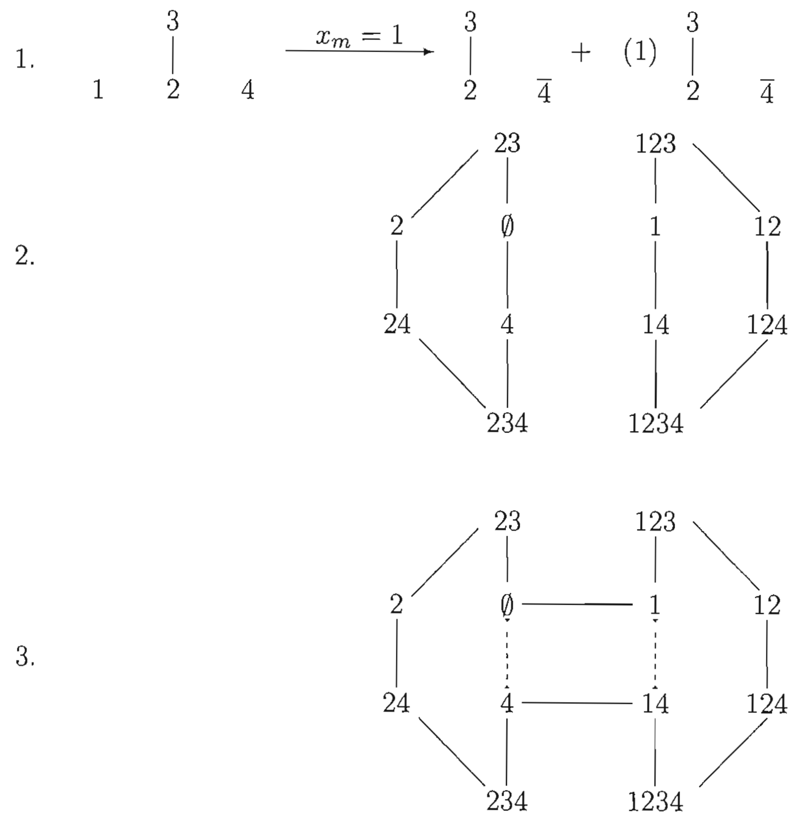


Figure 2.11. Example illustrating the use of Theorem 1 with the relaxed condition.

to be less than the median x in an earlier step; so it belongs to $D[x]$), the poset of the second part will not have any overlined minimal element, so we apply to this part the same procedure as that of Section 2.5.2. As for the poset of the first part, it keeps the same number of minimal elements as that of $\mathcal{P}$, and if $\mathcal{P}$ has one overlined minimal element, $\mathcal{P} \setminus U[x]$ will have this overlined minimal element. However the number of elements of this last poset decreases as long as we apply Theorem 3 or Theorem 2 until this minimal element satisfies the relaxed condition, where we can apply Theorem 1.

Example 9. *Consider the poset on the left side of the first line of Figure 2.12. Every minimal element satisfies the relaxed condition since the number of elements which are greater than each of these minimal elements is less than or equal to* 4 *which is half the number of elements of the poset. We take* 8 *as median. By applying Theorem 1 to this poset with* 8 *as median we obtain the first line of Figure 2.12. No minimal element of the first part satisfies the relaxed condition. The median of this poset is* 4*, and this median has at least two elements less than it; thus we overline a minimal element belonging to* $D[4]$ *and which is not the maximum of* $D[4] \setminus \{4\}$ *in the linear extension of the poset. We can choose* 2*, and we also overline this element for the poset of the second part. By applying Theorem 3 to the poset of the first part of the first line with* 4 *as chosen median we obtain the second line of Figure 2.12. As stated in Theorem 3, we underline each element of* $D[4] \setminus \{4\} = \{1, 2, 3\}$ *and we overline the maximum,* 3*, of the last set in its linear extension. We note that* 2 *keeps its overline. Each minimal element of the poset of the first part of line 2 satisfies the relaxed condition, particularly* 2*, which we take as median. By applying Theorem 1 to this last poset with* 2 *as median, we obtain the third line of the same figure. By applying Theorem 1 to each of the posets of the first and second parts twice we easily obtain the two cycles on the fourth line. By removing the edges* $\{\emptyset, 1\}$ *and* $\{2, 12\}$ *and adding the edges* $\{\emptyset, 2\}$ *and* $\{1, 12\}$*, we obtain the cycle on the fifth line which corresponds to the poset of the first part of the second line. As for the poset of the second part of this last line,* 5 *satisfies the relaxed condition. By applying Theorem 1 to this poset with* 5 *as median, we obtain a cycle which contains the edge* $\{\emptyset, 5\}$*. Since the set* 1234 *is added to each down-set of the last cycle, this cycle contains the edge* $\{1234, 12345\}$*. We find on the sixth line the cycle of the poset of the first part and that of the second part of the second line. Since we applied Theorem 3, the added edges are* $\{1234, 12\}$ *and* $\{12345, 123\}$ *while the removed edges are* $\{1234, 12345\}$ *and* $\{12, 123\}$*. The cycle obtained is on the seventh line. The cycle of the poset of the second part of the first line is the same as that of the first part after adding* 8 *to each of its down-sets. By adding the edges* $\{8, \emptyset\}$ *and* $\{28, 2\}$ *and removing the edges* $\{\emptyset, 2\}$ *and* $\{2, 28\}$ *we obtain the cycle corresponding to the poset on the left side of the eighth line.*

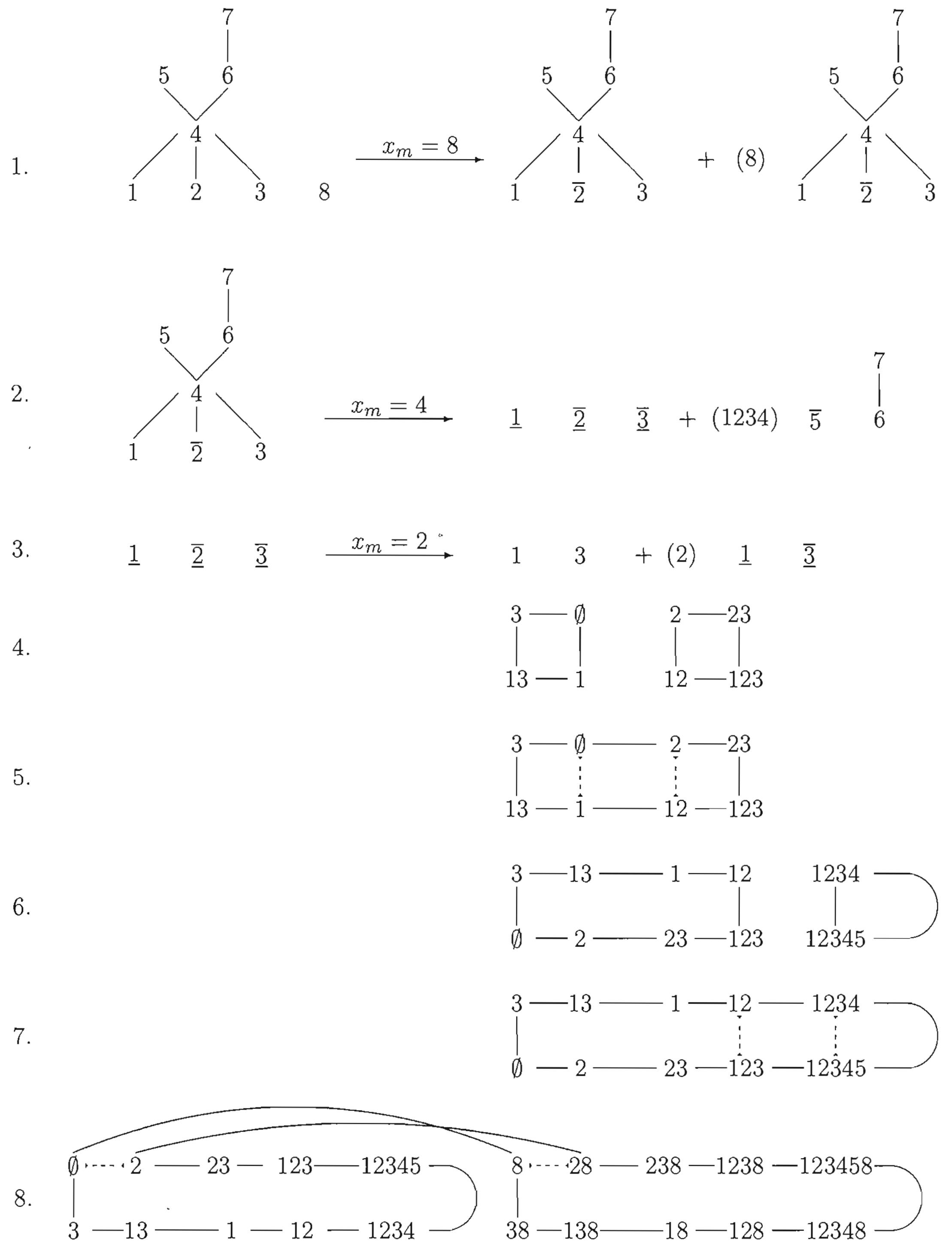


Figure 2.12. Example illustrating the use of Theorem 3 and Theorem 1 with the relaxed condition.

CHAPTER III

COMPARISON OF ALGORITHMS

We implemented our algorithm in the style used in Ruskey's program which was communicated to my supervisor T. Walsh by mail. Our program contains about 1600 lines whereas Ruskey's program contains about 300 lines. Ruskey used one theorem in his program, which is equivalent to Theorem 1 in this thesis, whereas we used in our program the three theorems and the three lemma stated and proved in this thesis. Consequently, there is a substantial difference in number of lines between these two programs. We used some functions that Ruskey had used, such as "*Update*", "*Recover*", etc. However, we extended and upgraded these functions to fit all the cases of our algorithm. For example, "*Update*" in Ruskey's program updates some information on the poset, such as the number of minimal elements and the number of elements that are less than or equal to each element of the poset, after removing one of its minimal elements. We extended this function and divided it into two types: "*Updateup*" and "*Updatedown*", which differ from "*Update*" by considering the median element (which is not necessarily a minimal element) instead of a minimal element. We also implemented Squire's algorithm; our implementation contains 220 lines. In this last program we did not need any treatment such as "*Update*", "*Recover*", etc.

We tested these three programs on several examples to measure their performance. We concluded that they have the same running time when the median is a minimal or satisfies the relaxed condition. In other cases, our program has almost the same running time as our implementation of Squire's algorithm and these two programs perform much

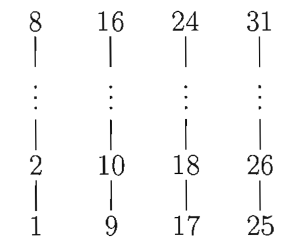
poset1 on the set $\{1, 2, \cdots, 13\}$	Running time in milliseconds and ratio		
	Squire	Our	Ruskey
1 2 $\cdots$ 13	8734	8984	8890
	Our/Squire		Ruskey/Squire
	1.029		1.018

Table 3.1. Running times of the programs for poset1.

better than Ruskey's program.

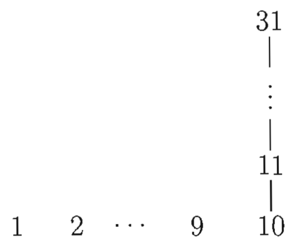
For example, poset1 in Table 3.1 is an antichain (where no two elements are related); each element of this poset satisfies the relaxed condition. Thus, Ruskey's program, which first chooses the least minimal element then the second minimal element and so on, performs as well as the two other programs. For poset2 in Table 3.2 which consists of four chains (a chain is a poset with a relation between every pair of elements), Ruskey's program also performs as well as the two other programs. Ruskey's program first exhausts the first two chains of which each element satisfies the relaxed condition then continues with the two left chains, and a minimal element taken with Ruskey's algorithm always satisfies the relaxed condition. On the other hand, with respect to poset3 in Table 3.3, Ruskey's program first exhausts the first 9 elements, each of which satisfies the relaxed condition, then continues with the rest of the poset which is a chain of which no minimum satisfies the relaxed condition. This explains why our program our is 1.8% slower than Squire's and Ruskey's is 45.8% slower than Squire's. The complexity of Squire's algorithm for a chain is $O(\log n)$ per ideal whereas the complexity of Pruesse and Ruskey's algorithm is $O(n)$ per ideal; this explains the superiority of our algorithm and Squire's algorithm over Pruesse and Ruskey's algorithm. Everything we said for poset3 applies to poset4 of Table 3.4, but instead of having a chain we have a tree of which no minimal element taken with Pruesse and Ruskey's algorithm satisfies the relaxed condition.

Remark 7. *We note that left and right shifts, which are used in the three implementations, are allowed only for integers. Consequently, a poset, which is represented by an*

poset2 on the set $\{1, 2, \cdots, 31\}$	Running time in milliseconds and ratio
	

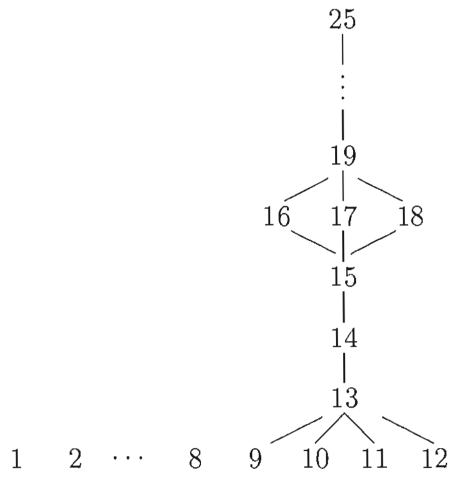
Squire	Our	Ruskey
15265	15218	15515
Our/Squire		Ruskey/Squire
0.997		1.016

Table 3.2. Running times of the programs for poset2.

poset3 on the set $\{1, 2, \cdots, 31\}$	Running time in milliseconds and ratio
	

Squire	Our	Ruskey
27312	27812	39813
Our/Squire		Ruskey/Squire
1.018		1.458

Table 3.3. Running times of the programs for poset3.

poset4 on the set $\{1, 2, \cdots, 25\}$	Running time in milliseconds and ratio
	

Squire	Our	Ruskey
13015	13421	21718
Our/Squire		Ruskey/Squire
1.031		1.667

Table 3.4. Running times of the programs for poset4.

integer, may be a set of at most 32 elements. Most often we include in a poset some isolated elements which makes it possible to repeat the process on the poset or a part of it depending on whether the median is chosen from those isolated elements. For example, when we take a poset consisting of 9 isolated elements, from 1 to 9, and of a linear poset of 23 elements $\{10 < 11, 11 < 12, \cdots, 31 < 32\}$*, each of the elements 1, 2,* $\cdots$*, 9 doubles the running time of the linear poset; thus the running time of this poset is* 2^9 *times the running time of the poset* $\{10 < 11, 11 < 12, \cdots, 31 < 32\}$*. Thus, adding isolated elements to a poset simulates taking a poset having more than 32 elements. We used this idea on five families: boolean lattices, posets whose Hasse diagram is a* $2 \times n$ *grid tilted by* 45^o*, posets where the total number of covers of all the* n *elements is* $\Gamma(n)$*, linear posets and empty posets.*

3.1 Boolean lattice family

Definition 2. *In a* boolean lattice, *we say that an element* $i_1 i_2 \cdots i_n$*, where* $i = 0$ *or* $i = 1$*, covers an element* $j_1 j_2 \cdots j_n$*, where* $j = 0$ *or* $j = 1$*, if there is a unique* k *for which* $i_k = 1$ *and* $j_k = 0$ *and* $\forall\ l \neq k\ i_l = j_k$*.*

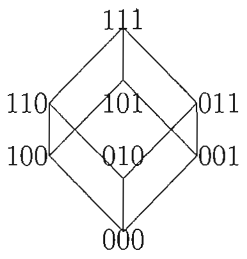


Figure 3.1. Boolean lattice of 8 elements.

Using the above definition when $n = 4$ and starting from 1 instead of 0, we obtain the following poset: lattice16 = $\{1 < 2, 1 < 3, 1 < 5, 1 < 9, 2 < 4, 2 < 6, 2 < 10, 3 < 4, 3 < 7, 3 < 11, 5 < 6, 5 < 7, 5 < 13, 9 < 10, 9 < 11, 9 < 13, 4 < 8, 4 < 12, 6 < 8, 6 < 14, 7 < 8, 7 < 15, 10 < 12, 10 < 14, 11 < 12, 11 < 15, 13 < 14, 13 < 15, 8 < 16, 12 < 16, 14 < 16, 15 < 16\}$. When $n = 5$, we obtain lattice32 which consists of 32 elements. The other posets: lattice$I$-16, where $1 \leq I \leq 6$, are obtained by adding I isolated elements, from 1 to I, to the poset "lattice16" after increasing by I each element of this latter poset. For lattice16, we note that, except for the maximum and the minimum, each element has a brother; so Theorem 3 can never be used and this is affirmed by Table 3.6. This also applies to all other examples but lattice32 on this family. We note that our program performs better as the number of applications of Theorem 2 increases (see Table 3.5 and Table 3.6).

3.2 Grid poset family

Definition 3. *In a* grid poset *of* $2n$ *elements, if* k *is even it covers* $k-1$ *and* $k-2$ *except that 2 covers only 1, and if* k *is odd it covers* $k-2$ *except that 1 covers nothing.*

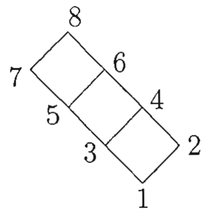


Figure 3.2. Grid poset of 8 elements.

Using the above definition when the number of elements is 16, we obtain the following poset: grid16 $= \{1 < 2, 1 < 3, 3 < 4, 3 < 5, 5 < 6, 5 < 7, 7 < 8, 7 < 9, 9 < 10, 9 < 11,$ $11 < 12, 11 < 13, 13 < 14, 14 < 15, 15 < 16, 2 < 4, 4 < 6, 6 < 8, 8 < 10, 10 < 12,$ $12 < 14, 14 < 16\}$. The poset "grid8-16" is obtained by adding 8 isolated elements, from 1 to 8, to the poset "grid16" after increasing by 8 each element of this latter poset. The other posets are obtained in the same manner.

For grid16, we note that, except for the maximum and the minimum, no element has a brother; so Theorem 3 can never be used unless we choose the second element "2" as a median. In this case, all the even-numbered elements belong to up[this element] and this element covers only 1. Thus, the first part, after applying Theorem 2, consists of the linear poset $\{1 < 3, 3 < 5, 5 < 7, 7 < 9, 9 < 11, 11 < 13, 13 < 15\}$ and for the next steps, Theorem 3 will be used. This also applies to all other examples in this family when we have isolated elements. We note that our program performs better as the number of applications of Theorem 2 increases (see Table 3.7 and Table 3.8).

3.3 Gamma poset family

Definition 4. *In a* gamma poset *of* $2n$ *elements,* j *covers* i *if* $1 \leq i \leq n$ *and* $n + 1 \leq j \leq 2n$.

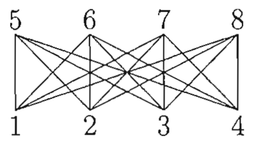


Figure 3.3. Gamma poset of 8 elements.

Using the above definition when the number of elements is 16, we obtain the following poset: gamma5 $= \{1 < 6, 1 < 7, 1 < 8, 1 < 9, 1 < 10, 2 < 6, 2 < 7, 2 < 8, 2 < 9, 2 < 10, 3 < 6, 3 < 7, 3 < 8, 3 < 9, 3 < 10, 4 < 6, 4 < 7, 4 < 8, 4 < 9, 4 < 10, 5 < 6, 5 < 7, 5 < 8, 5 < 9, 5 < 10\}$. The other posets are obtained in the same manner.

For all the posets taken from this family, we note that Theorem 3 can never be used. However, the performance of our program is still as good as when Theorem 3 and/or Theorem 2 can be used most often. We also note that our program performs a little better than Ruskey's (see Table 3.9 and Table 3.10).

3.4 Linear posets

Definition 5. *A* linear poset *of* n *elements is a poset in which* i *covers* $i - 1$ $\forall\, i = 2 \cdots n$.

Using the above definition, when the number of elements is 23 we obtain the following poset: linear23 $= \{1 < 2, 2 < 3, \cdots, 22 < 23\}$. The poset linear8-23 is obtained by adding 8 isolated elements, from 1 to 8, to the poset "linear23" after increasing by 8 each element of this latter poset. The other posets are obtained in the same manner.

For all posets taken from this family, we note the use of Theorem 3 and/or Theorem 2. We also note that our program performs as well as it does for the preceding families and its performance improves as the number of applications of Theorem 3 and/or Theorem 2 increases. On the other hand, Ruskey's program doesn't perform as well as ours for this family (see Table 3.11 and Table 3.12).

3.5 Empty posets

Definition 6. *An* empty poset *of n elements is a poset where no two elements are related.*

For all posets taken from this family, we note that Theorem 3 can never be used and neither can Theorem 2. However, the performance of our program is still as good as when Theorem 3 and/or Theorem 2 can be used most often. We also note that Ruskey's program performs a little better than ours; this is due to the useless treatment time when we cannot use Theorem 3 or Theorem 2 (see Table 3.13 and Table 3.14).

The test for those families show that our program is on the average slower from 1.7% to 4.7% than Squire's while Ruskey's is from 1.8% to 37.7% slower than Squire's.

Consequently, for all posets, our algorithm is as fast as Squire's to a constant factor (see Tables 3.6, 3.8, 3.10, 3.12 and 3.14). For most of these posets, those for which we use neither Theorem 2 nor Theorem 3 in our algorithm, i.e. those for which the median is minimal (see Tables 3.10 and 3.14) the Pruesse-Ruskey algorithm is as fast as the two others to a constant factor. But for some posets, those for which we use either Theorem 2 or Theorem 3 in our algorithm, i.e. those for which the median element may not be minimal (see Tables 3.6, 3.8 and 3.12) it is slower than the other two algorithms.

Finally, we conclude that the more we use Theorem 2 or Theorem 3 in our algorithm, the closer the Pruesse-Ruskey algorithm comes to its worst case and our algorithm to its best case (see Table 3.12).

For each of the three algorithms, we assumed that the poset was represented by a $n \times n$ matrix, where n is the number of elements in the poset. For Squire's algorithm and ours we use the linear extension $1 < 2 < \cdots < n$, which is compatible with all our posets, since $i < j$ for each poset if and only if $i < j$ as integers.

Example	Running time in milliseconds of			Number of ideals
	Squire	Our	Ruskey	
lattice16	281	281	281	168
lattice32	17432	18141	18218	7581
lattice1-16	515	546	531	336
lattice2-16	968	1046	1078	672
lattice3-16	1969	2110	2250	1344
lattice4-16	4000	4234	4578	2688
lattice5-16	8376	8671	9609	5376
lattice6-16	17406	17875	19890	10752

Table 3.5. Running time of execution on a family of boolean lattices.

Example	Ratio of running times		Number of applications of			
	Our/Squire	Ruskey/Squire	Lemma1	Lemma2	Theorem2	Theorem 3
lattice16	1.000	1.000	14	13	13	0
lattice32	1.040	1.045	89	634	239	15
lattice1-16	1.060	1.031	30	25	16	0
lattice2-16	1.080	1.114	51	56	39	0
lattice3-16	1.072	1.143	106	106	56	0
lattice4-16	1.058	1.145	159	253	94	0
lattice5-16	1.035	1.147	341	482	108	0
lattice6-16	1.027	1.143	626	1025	171	0
Average	1.047	1.096				

Table 3.6. For boolean lattices, our program is 4.7% on the average slower than Squire's and Ruskey's is 9.6% slower than Squire's.

Example	Running time in milliseconds of			Number of ideals
	Squire	Our	Ruskey	
grid28	296	296	296	120
grid8-10	7609	7796	9296	5376
grid8-12	11766	12078	14716	7168
grid8-14	15578	15953	20344	9216
grid8-16	21046	21312	28140	11520
grid10-20	153313	155141	199875	67584
grid11-16	189813	195296	247250	92160
grid11-18	247235	252390	323359	112640

Table 3.7. Running time of execution on a family of grid posets.

Example	Ratio of running times		Number of applications of			
	Our/Squire	Ruskey/Squire	Lemma1	Lemma2	Theorem2	Theorem 3
grid28	1.000	1.000	5	0	27	6
grid8-10	1.025	1.222	640	354	40	0
grid8-12	1.026	1.250	368	706	153	0
grid8-14	1.024	1.306	592	847	194	0
grid8-16	1.013	1.337	880	1128	215	0
grid10-20	1.012	1.304	5936	6278	1261	0
grid11-16	1.029	1.301	9216	6847	1346	0
grid11-18	1.021	1.308	11552	9384	1383	0
Average	1.019	1.254				

Table 3.8. For grid posets, our program is 1.9% on the average slower than Squire's and Ruskey's is 25.4% slower than Squire's.

Example	Running time in milliseconds of			Number of ideals
	Squire	Our	Ruskey	
gamma10	2921	3015	3000	2047
gamma11	6328	6453	6468	4095
gamma12	13734	13953	13968	8191
gamma13	29737	30047	30125	16383
gamma14	64563	65031	65437	32767
gamma15	137047	139234	138860	65535

Table 3.9. Running time of execution on a family of gamma posets.

Example	Ratio of running times		Number of applications of			
	Our/Squire	Ruskey/Squire	Lemma1	Lemma2	Theorem2	Theorem 3
gamma10	1.032	1.027	32	262	10	0
gamma11	1.020	1.022	128	225	11	0
gamma12	1.016	1.017	130	503	12	0
gamma13	1.010	1.013	514	912	13	0
gamma14	1.007	1.014	520	2027	14	0
gamma15	1.016	1.013	3074	7678	0	0
Average	1.017	1.018				

Table 3.10. For gamma posets, our program is 1.7% on the average slower than Squire's and Ruskey's is 1.8% slower than Squire's.

Example	Running time in milliseconds of			Number of ideals
	Squire	Our	Ruskey	
linear32	141	141	141	33
linear8-23	14140	14640	20984	6144
linear9-21	25297	25593	36562	11264
linear9-22	27312	27812	39813	11776
linear10-20	49797	50266	71141	21504
linear11-19	93610	96891	133360	40960
linear12-18	171406	178250	240062	77824

Table 3.11. Running time of execution on a family of linear posets.

Example	Ratio of running times		Number of applications of			
	Our/Squire	Ruskey/Squire	Lemma1	Lemma2	Theorem2	Theorem 3
linear32	1.000	1.000	0	0	8	7
linear8-23	1.035	1.484	518	575	65	12
linear9-21	1.012	1.445	1057	1056	33	36
linear9-22	1.018	1.458	1056	1057	49	34
linear10-20	1.009	1.429	1792	1940	160	2
linear11-19	1.035	1.425	3264	4020	3	0
linear12-18	1.040	1.401	4224	8383	2	0
Average	1.021	1.377				

Table 3.12. For linear posets, our program is 2.1% on the average slower than Squire's and Ruskey's is 37.7% slower than Squire's.

Example	Running time in milliseconds of			Number of ideals
	Squire	Our	Ruskey	
empty10	890	906	890	1024
empty11	1906	1937	1906	2048
empty12	4093	4171	4109	4096
empty13	8734	8984	8890	8192
empty14	18063	18562	18688	16384
empty15	39000	39656	40515	32768
empty16	81453	83968	84437	65536

Table 3.13. Running time of execution on a family of empty posets.

Example	Ratio of running times		Number of applications of			
	Our/Squire	Ruskey/Squire	Lemma1	Lemma2	Theorem2	Theorem 3
empty10	1.018	1.000	0	138	0	0
empty11	1.016	1.000	192	202	0	0
empty12	1.019	1.004	0	554	0	0
empty13	1.029	1.018	768	810	0	0
empty14	1.028	1.035	0	2218	0	0
empty15	1.017	1.039	3072	3242	0	0
empty16	1.031	1.037	0	8874	0	0
Average	1.023	1.019				

Table 3.14. For empty posets, our program is 2.3% on the average slower than Squire's and Ruskey's is 1.9% slower than Squire's.

CONCLUSION

In order to find an efficient algorithm for the generation of a Gray code of the ideals of a poset such that two successive ideals differ in one or two elements, we used Squire's recurrence (see (Squire)). We could choose the median element at each step of the algorithm or postpone the choice of a minimum until it becomes a median in a later step. Thus, we kept the same idea as Squire's algorithm, that is: always choose the median element. Our algorithm generates the ideals in Gray code order whereas Squire's algorithm does not. We programmed our algorithm and tested in on several examples which together illustrate most possible cases (see Table 3.1 through Table 3.14). We found that the running time of our program is almost the same as that of our program of Squire's algorithm for all those posets. For most of those posets, the running time of Ruskey's program is close to each of the other two programs whereas for some posets it is much greater than that of each of the other two programs.

APPENDIX A

PROGRAMS

A.1 Our Program

```
//--------------------------------------------------------------------//
//   This program is written in Dev-C++, version 4.9.9.2         //
//--------------------------------------------------------------------//

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#define MAX_POINTS 50
#define MAXSTACK 10000
# define MaxQueue 11000

typedef enum{FORWARD, BACKWARD}Direction;
unsigned int table[100];
   //table[i] contains the elements which are >=i
unsigned int parents[100];
//parents[i] contains the parents of i (>=i), not the ancestors.
unsigned int up[100], down[100]; //up[el] contains the elements
                //which are >=el, and down[el] those which are<=el.
unsigned int children[100]; //children[Max] contains the elements
                            // which are <= Max, see Updatedown.
unsigned int med[100];//If med[5]=1, then 5 had been chosen for the
                 //median having more than 2 elements less than it.
               // med is initialised to 1 in Updatedown and updated
                // to zero in Recoverdown.
unsigned int next;
int nup[100], ndown[100];
//nup (ndown) = number of elements in up (ndown).
int  Max;   // It is initialised in Updatedown and corresponds
            // to max in Theorem 3 or Lemma 3.
int  Min; //It appears in Updateup and contains current min. This
         // allows us, where appropriate, to use the previous min
```

```
     //(item.min) in order to treat the elements which have 2 marks
     // under them
//Min=0 if min have 2 marks under it. Once used in Recoverup, it is
//set to zero. No need to have it, or Max global.

int nbefore[100];   // nbefore[pos] = nb of elements that must be
                    // chosen for "min" before pos + 1.
int ancetre[100];   // ancetre[el] = Max if el is in children[Max].

int  flag =-1;    // flag initialised once Theorem 3 used, it
//increases form -1 to 1, and the data will be stored in item
//(item.min, item.poset, item.mask, item.max). When flag exceeds 1,
// we push old item and we store the new data in the new
// item. Thus, flag = "the number of elements in stack"+ 1
int N;          //then we pop and enqueue.
int counter = 0;  // Number of ideals

typedef short Boolean;
typedef struct{
   int min;//The minimum of the current poset when using Theorem 3.
   unsigned int poset;//Each element in the poset is
//represented by its rank in the binary representation of the poset
   int mask;//Corresponds to (D U D[min]) in Squire's recurrence.
   int max;       //Corresponds to max in lemma 3.
   } Item_type;

class Stack {
 // we start pushing item when flag >1. See flag's comments.
  private:      //We push onto the stack from index 0 to MAXSTACK-1.
    int top;        //If top=MAXSTACK (i) then the last element
    Item_type rgItem[MAXSTACK]; //is at the index MAXSTACK-1 (i-1).
  public:
    Stack(){ top = 0; };
    Boolean EmptyStack(){ return(top <= 0); };// < not necessary
    Boolean FullStack(){return(top >= MAXSTACK);};//> not necessary
    int Push(Item_type item);
    int Pop(Item_type &item);
};
  Stack stack;
  Item_type item;
//------------------------------------------------------------------
// push an item on the stack, return zero for success, -1 on error.
//------------------------------------------------------------------
int Stack::Push(Item_type item)
{
    if (top >= MAXSTACK)
        return(-1);
    else {
        rgItem[top++] = item;
        return(0);
    }
}
//------------------------------------------------------------------
// pop an item off the stack. return zero for success, -1 on error.
//------------------------------------------------------------------
int Stack::Pop(Item_type &item)
{
    if (top <= 0)
        return(-1);
    else {
        item = rgItem[--top];
        return(0);
    }
}

//------------------------------------------------------------------
class SStack {
//This stack is intended to form children[Max], the elements <= Max
  private:
    int ttop;          /* top of stack */
    unsigned int rg[MAXSTACK];
  public:
    SStack(){ ttop = 0; };
    Boolean EmptySStack(){ return(ttop <= 0); };
    Boolean FullSStack(){ return(ttop >= MAXSTACK); };
    int PPush(unsigned int iitem);
    int PPop(unsigned int &iitem);
};
  SStack sstack, direc;
  unsigned int iitem;
//------------------------------------------------------------------
// push an item on the stack, return zero for success, -1 on error
//------------------------------------------------------------------
int SStack::PPush(unsigned int iitem)
{
    if (ttop >= MAXSTACK)
        return(-1);
    else {
        rg[ttop++] = iitem;
        return(0);
    }
}
//------------------------------------------------------------------
// pop an item off the stack, return zero for success, -1 on error.
//------------------------------------------------------------------
int SStack::PPop(unsigned int &iitem)
{
    if (ttop <= 0)
        return(-1);
    else {
        iitem = rg[--ttop];
        return(0);
    }
}

//----------------------------------------------------------------------
//---------------------------------- Queue ------------------------------
//----------------------------------------------------------------------
class queue{
      int first, last;
      unsigned int storage[MaxQueue];
public:
       queue(){ first=last=-1; }
       // We enqueue from index 0 to MaxQueue-1
       void enqueue(const unsigned int);
      // If last=MaxQueue-1 then the last element is
      //at the index MaxQueue-1
        unsigned int dequeue();
       int isfull() const {return first==0 && last ==MaxQueue-1
                            || first==last+1; }
       int isempty() const {return first==-1; }
       };

void queue::enqueue(const unsigned int  el){
     if(!isfull())
       if(last==MaxQueue-1 || last==-1){
                       storage[0]=el;
                       last=0;
                       if(first==-1) first=0;
                       }
       else storage[++last]=el;
     else {printf( "Full queue.\n");exit(1);}
}

unsigned int queue::dequeue(){
    unsigned int tmp;
    if(!isempty()){
                    tmp=storage[first];
                    if(first==last) last=first=-1;
                    else if(first == MaxQueue-1) first=0;
                         else first++;
                    return tmp;
                    }
    else {printf( "Empty queue.\n");exit(1);}
}

//----------------------------------------------------------------------
void lemma3(unsigned int mask1, unsigned int poset, int &listack);
void lemma1(unsigned long poset, int x, int y, int mask, int num,
           int n, queue &p, queue &q, int &listack);
```

```
void lemma2(unsigned long poset, int x, int y, int mask, int num,
            int n, queue &p, queue &q, int &listack);
void Ideal(unsigned long poset, Direction dir, int x, int mask,
          int num, int n, queue &p, queue &q, int &listack);
void lemma1inv(unsigned long poset, int x, int y, int mask, int
               num, int n, queue &p, queue &q, int &listack);
//-----------------------------------------------------------------

unsigned getbit(unsigned int word, int n)
{
  return (word >> n) & 01 ;
}

unsigned int setbit(unsigned int word, int n, unsigned v)
{
  if (v != 0)
    return word | (01 << n);
  else
    return word & ~(01 << n);
}
//-----------------------------------------------------------------
void Process()
{
  int i, j, k, count, bit;
  // transitive closure
  for (k=1; k <= N; k++){
    for (i=1; i <= N; i++){
      for (j=1; j <= N; j++){
        bit = getbit(table[i], j) | (getbit(table[i], k) &
                            getbit(table[k], j));
        table[i] = setbit(table[i], j, bit);
      }
    }
  }
  for (i=1; i <= N; i++){ nup[i]=0; ndown[i]=0; }
  for (i=1; i <= N; i++){
     up[i] = 0, down[i]=0;
     count = 0;
     for (j=1; j <= N; j++){
         if(getbit(table[i], j)){
           up[i] = setbit(up[i], j, 1);
           nup[i]++;
         }
         if(getbit(table[j], i)){
            down[i]=setbit(down[i],j,1);
            ndown[i]++;
         }
     }
  }
}
```

```
//------------------------------------------------------------------
// ----------------        prints the ideal in the format {i, j, ... } ---
//------------------------------------------------------------------
void prints(unsigned int number)
{
  int i, flag = 0;
         printf("{");
         for(i=1 ; i <= N ; i++)
            if(getbit(number, i)){
               if(flag)
                  printf(" %d", i);
               else
                  printf("%d", i);
               if(!flag)
                  flag = 1;
            }
            printf("}\n");  counter++;
 }

//------------------------------------------------------------------
// ----------------        prints the ideal in the format {i, j, ... } ---
//------------------------------------------------------------------
void print(unsigned int number, queue &p, queue &q, int &listack)
{
  int i, flag1 = 0;
     switch(listack){
      case 0: p.enqueue(number);
         break;
      case 1: q.enqueue(number);
     }
  }

//------------------------------------------------------------------
int search(unsigned int mask, int value, int start, int end){
    int pos;
    if(start > end){
       printf("index error\n");
       exit(1);
    }
    if(start == end)
      return start;
    if(end == start + 1){
      if(getbit(mask, end) == value)
          return end;
      else if (getbit(mask, start) == value)
          return start;
      else{
         printf("\nfatal error!!!\n");
         printf("mask is %d,start is %d,end is %d", mask,start,end);
```

```
          exit(1);}
      }
      else{
         pos = (start + end) / 2;
         if(getbit(mask, pos) == value)
           return (search(mask, value, pos, end));
         else
           return (search(mask, value, start, pos));
      }
   }

//-----------------------------------------------------------------------
// -------- find the number of elements of poset --------
//-----------------------------------------------------------------------
int NumElement(int poset)
{
  long unsigned mask, res;
  int pos, num =0;
  mask = poset;
  pos = 0;
  while(mask !=0){
    res = (mask -1) ^ mask;
    pos = search(res, 1, pos, N);
    num++;
    mask = setbit(mask, pos, 0);
  }
  return num;
}
//-----------------------------------------------------------------------
// -------- find num of poset (number of miuima of poset) --------
//-----------------------------------------------------------------------
int Num(int poset)
{
  long unsigned mask, res;
  int pos, num =0;
  mask = poset;
  pos = 0;
  while(mask !=0){
    res = (mask -1) ^ mask;
    pos = search(res, 1, pos, N);
    if(ndown[pos] == 1)
        num++;
    mask = setbit(mask, pos, 0);
  }
  return num;
}

//-----------------------------------------------------------------------
// ------      Updateup the list for finding median element --------
//-----------------------------------------------------------------------
```

```
int Updateup(int poset, int min, int num)
{               //Updates up[min] after removing down[min] from poset.
  long unsigned mask, res, other, fils, fils1=0;
  int pos, x, n=0;
  mask = (~down[min]) & poset & up[min];//min is removed from poset
  pos = 0;

  fils=children[ancetre[min]] & down[min] & poset;

  if(fils) {                    //We remove min here so that it keeps
         fils=setbit(fils, min, 0);//its link with its ancestor Max
         nbefore[ancetre[min]]--;
         }
  else if(flag && nbefore[ancetre[item.max]]>0)
        { Min=min;
        fils1=children[item.max] & down[item.min] & poset;
        }

  while(mask !=0){
        res = (mask -1) ^ mask;
        pos = search(res, 1, pos, N);
        ndown[pos]=ndown[pos]-ndown[min];

        if(ndown[pos] == 1) num++;

        mask = setbit(mask, pos, 0);
  }

//--------------------> Remove the elements with marks from children
  pos=0;                          //--> which are in down[min]
  while(fils){
        res = (fils -1) ^ fils;
        pos = search(res, 1, pos, N);
        ancetre[pos]=0;
        nbefore[ancetre[min]]--;
        fils=setbit(fils, pos, 0);
        }
//--------------------> Remove the elements with marks from children
  pos=0;                     //--> which are not in down[min]
  while(fils1){
        res = (fils1 -1) ^ fils1;
        pos = search(res, 1, pos, N);
        ancetre[pos]=0;
        nbefore[item.max]--;
        fils1=setbit(fils1, pos, 0);
        }
//--------------> // to treat elements that don't belong to up[min]
  mask = (poset & up[min])^((~down[min])&poset); //equivalent to:
  mask=setbit(mask, min, 0);              //poset |down[min] | up[min]
  pos = 0;
  while(mask !=0){
        res = (mask -1) ^ mask;
        pos = search(res, 1, pos, N);
        other = down[pos]& down[min] & poset;
        if(other){
               ndown[pos]=ndown[pos]- NumElement(other);
               if(ndown[pos]==1)  num++;
               }
        mask = setbit(mask, pos, 0);
        }
//-----------------------------------------
  if(ndown[min]==1 || ndown[min]==2) x=1;
  else  x=Num(down[min]& poset);
  return num - x;
}

//------------------------------------------------------------------------
// ------------  Recoverup the list for finding median element  ------
//------------------------------------------------------------------------
int Recoverup(int poset, int min, int num, int mini){ //the reverse
  long unsigned mask, res, other, fils, fils1=0;       //effect of
  int pos, x;                                          //Updateup
  mask = (~down[min]) & poset & up[min];
  pos = 0;

  fils=children[ancetre[min]] & down[min] & poset;
  if(fils) {                 // we remove min here so that it keeps
        fils=setbit(fils, min, 0);//its link with its ancestor Max
        nbefore[ancetre[min]]++;
        }
  else if(min==mini)
       {fils1=children[item.max] & down[item.min] & poset;
        }


  while(mask != 0){
        res = (mask -1) ^ mask;
        pos = search(res, 1, pos, N);

        if(ndown[pos] == 1)    num--;
        ndown[pos]=ndown[pos]+ ndown[min];
        mask = setbit(mask, pos, 0);
  }

//--------------------> Remove the elements with marks from children
  pos=0;                          //--> which are in down[min]
  while(fils){
        res = (fils -1) ^ fils;
        pos = search(res, 1, pos, N);
        ancetre[pos]=ancetre[min];
        nbefore[ancetre[min]]++;
        fils=setbit(fils, pos, 0);
        }
//——————————————> Remove the elements with marks from children
  pos=0;                       //—> which are not in down[min]
  while(fils1){
        res = (fils1 -1) ^ fils1;
        pos = search(res, 1, pos, N);
        ancetre[pos]=item.max;

        nbefore[item.max]++;
        fils1=setbit(fils1, pos, 0);
        }
//——————————> // to treat elements that don't belong to up[min]
  mask = (poset & up[min])^((~down[min])&poset);//equivalent to:
  mask=setbit(mask, min, 0);           // poset |down[min] | up[min]
  pos = 0;
  while(mask !=0){
        res = (mask -1) ^ mask;
        pos = search(res, 1, pos, N);
        other = down[pos]& down[min] & poset;
        if(other){
               if(ndown[pos]==1) num--;
               ndown[pos]=ndown[pos]+ NumElement(other);
               }
        mask = setbit(mask, pos, 0);
        }

  if(ndown[min]==1 || ndown[min]==2) x=1;
  else  x=Num(down[min] & poset);
  return num + x;
}

//————————————————————————————————————————————————————————————
// ——  Updatedown the list for finding median element  ————————
//————————————————————————————————————————————————————————————
int Updatedown(int poset, int min, int num)
{            // updates down[min] after removing up[min] from poset
  long unsigned mask, res, other, fils;
  int pos, n=0;

  mask = (~up[min])& poset & down[min]; //min is removed from poset
  pos = 0;

  fils=children[ancetre[min]] & poset;
  if(fils) {                    //We remove min here so that it keeps
        fils=setbit(fils, min, 0);//its link with its ancestor Max.
        nbefore[ancetre[min]]--;
        }
  while(mask !=0){
        res = (mask -1) ^ mask;
        pos = search(res, 1, pos, N);
        nup[pos]=nup[pos]-nup[min];

        if(ndown[min]>2){
             med[min]=1;
             Max=pos;
             sstack.PPush(pos); n++;
             }
//--------------------------> Remove the elements with marks from children
        if(fils){                    //---> which are in down[min]
            fils=setbit(fils, pos, 0);
            ancetre[pos]=0;
            nbefore[ancetre[min]]--;
            }

        mask = setbit(mask, pos, 0);
  }
//--------------------------> Remove the elements with marks from children
  pos=0;                             //---> which are not in down[min]
  while(fils){
        res = (fils -1) ^ fils;
        pos = search(res, 1, pos, N);
        ancetre[pos]=0;
        nbefore[ancetre[min]]--;
        fils=setbit(fils, pos, 0);
        }

  if(n) {
        nbefore[Max]=n;            //---> gives value to children <---
        children[Max]= poset & setbit(down[min],min,0);

        for(int i=1; i<n+1; i++) {
            sstack.PPop(iitem);  ancetre[iitem]=Max ;
            }
  }

//--------------------> to treat elements that don't belong to down[min]
  mask = (poset & down[min])^((~up[min])& poset); // equivalent to:
  mask=setbit(mask, min, 0);              // poset \ down[min] | up[min]
  pos = 0;
  while(mask !=0){
        res = (mask -1) ^ mask;
        pos = search(res, 1, pos, N);
        other = up[pos]& up[min] & poset;
        if(other) nup[pos]=nup[pos]- NumElement(other);
        mask = setbit(mask, pos, 0);
        }
  if(ndown[min]!=1) num++;    // We suppose that no child of min
  return num -1;              // (element < min) has another parent.
}                         //This is the brother condition of Theorem 3

//------------------------------------------------------------------
// ------------ Recoverdown the list for finding median element ---
//------------------------------------------------------------------
int Recoverdown(int poset, int min, int num){         // the reverse
  long unsigned mask, res, other, fils=0, filsprec=0;  // effect of
  int pos;                                              //Updatedown
  mask = (~up[min])& poset & down[min];
  pos = 0;
  // med[min] allows us to find Recoverdown when we use
  // Theorem 3 in Updatedown: i.e. when ndown[min]>2
  if(med[min]==1)
    { fils=children[item.max] & poset; //fils contains the elements
      filsprec=children[ancetre[min]] & poset;      // with 2 marks
     }                    //filsprec exists when min has 2 marks
  else  fils=children[ancetre[min]] & poset;

  if(fils) {
         fils=setbit(fils, min, 0); //ancetre[min]=0;
         if(!med[min])  nbefore[ancetre[min]]++; //i.e. we used
         }                          //Theorem 3 in the previous step

  while(mask != 0){
       res = (mask -1) ^ mask;
       pos = search(res, 1, pos, N);
       nup[pos]=nup[pos]+ nup[min];
//-------------------> Remove the elements with marks from children
                                         //--> which are in down[min]
       if(fils){
            fils=setbit(fils, pos, 0);
            if(med[min])
               {ancetre[pos]=0;
                nbefore[item.max]--;
                }
            else
               {ancetre[pos]=ancetre[min];
                nbefore[ancetre[min]]++;
                }
            }

       mask = setbit(mask, pos, 0);
  }
//-------------------> Remove the elements with marks from children
  pos=0;                              //--> which are not in down[min]
  while(fils){
        res = (fils -1) ^ fils;
        pos = search(res, 1, pos, N);
```

```
            ancetre[pos]=ancetre[min];
            nbefore[ancetre[min]]++;
            fils=setbit(fils, pos, 0);
            }

    pos=0;
    while(filsprec){
            res = (filsprec -1) ^ filsprec;
            pos = search(res, 1, pos, N);
            ancetre[pos]=ancetre[min];
            nbefore[ancetre[min]]++;
            filsprec=setbit(filsprec, pos, 0);
            }

  //-----------------> to treat elements that don't belong to down[min]
    mask = (poset & down[min])^((~up[min])& poset); // equivalent to:
    mask = setbit(mask, min, 0);          //poset | down[min] | up[min]
    pos = 0;
    while(mask !=0){
            res = (mask -1) ^ mask;
            pos = search(res, 1, pos, N);
            other = up[pos]& up[min] & poset;
            if(other) nup[pos]=nup[pos]+ NumElement(other);
            mask = setbit(mask, pos, 0);
            }
    if(med[min])     med[min]=0;
    if(ndown[min]!=1) num--;
    return num +1;
}

//--------------------------------------------------------------------------
// ----------------   find the minimal element in the poset   --------------
//--------------------------------------------------------------------------
int findMinimal(unsigned long poset)
{
    unsigned long res;
    int pos = 0;

    while(poset !=0){
      res = poset ^ (poset -1);
      pos = search(res, 1, pos, N);
      if(ndown[pos] == 1 && nbefore[pos]<1) //if pos is a minimum and
                                 //(if pos does not have 2 marks or if
         break;                  //is the unique element with 2 marks)
      poset = setbit(poset, pos, 0);
    }
    return pos;
}

//--------------------------------------------------------------------------
```

```
// ----------------        parent is a unique parent of child   --------
//------------------------------------------------------------------------
Boolean uniqueParent(unsigned long poset, int child, int parent)
{
        return (child && parent && setbit(up[child]& poset,child,0)
                == (up[parent] & poset) );
}

//------------------------------------------------------------------------
//choose one of coched1, coched2 or coched3 that has a unique child
//------------------------------------------------------------------------
int chooseCoched(unsigned long poset, int coched1, int coched2,
                int coched3)
{ int pos=0;                    //coched2 < coched1 in the extension
  if(coched1 && (!coched2 || ndown[coched1]<3)) pos=coched1;
   //if coched1 and it is the unique element with a mark, or it
   //has a unique child with one mark we choose coched1
  else if(coched2 && (!coched3 || ndown[coched2]<3) ) pos=coched2;
  else if(uniqueParent(poset,coched3,coched2))   pos=coched2;
        // In this case, we apply Theorem 3 and
 //the new Max is on the elements with a mark of the previous step.
  else pos=findMinimal(poset);
return pos;
}
//------------------------------------------------------------------------
// choose one of pos, prec1, prec2 or prec3 that is a unique parent
//------------------------------------------------------------------------
int chooseParent(unsigned long poset, int pos, int prec1, int prec2
                , int prec3)
{
if(ndown[pos]>2)
  {if(uniqueParent(poset, prec1, pos));       //pos is unique parent
   else if(ndown[prec1]>2)                    //of prec1
        {if(uniqueParent(poset, prec2, prec1))
                pos=prec1; // prec1 is unique parent of prec2
         else if(uniqueParent(poset, prec2, pos)); //pos is unique
         else if(ndown[prec2]>2)                  //parent of prec2
              {if(uniqueParent(poset, prec3, prec2))
                   pos=prec2;//prec2 is unique parent of prec3
               else if(uniqueParent(poset, prec3, prec1))
                   pos=prec1;
               else if(uniqueParent(poset, prec3, pos)); //pos is
               else if(ndown[prec3]>2)   //unique parent of prec3
                    pos=findMinimal(poset);
               else pos=prec3;   //prec3<=2
              }
         else pos=prec2; //prec2<=2
        }
   else  pos=prec1; // prec1<=2
  }
else
   ; //ndown[pos] < 3
     return pos;
 }
//----------------------------------------------------------------------
// ---------------------    find median element    --------------------
//----------------------------------------------------------------------
int findMedian(unsigned long poset, int n)//find the median element
 {
     unsigned long res, poset1;
     int pos=0, compt=0, coched1=0, coched2=0, coched3=0,coched4=0;
     int prec1=0, prec2=0, prec3=0;
     poset1=poset;
     while(compt!=n){          //coched2 < coched1 in the extension
                 if(ancetre[pos])
                      {coched4=coched3; coched3=coched2;
                       coched2=coched1; coched1=pos;}
                 prec3=prec2; prec2=prec1; prec1=pos;
                 res=poset1^(poset1-1);
                 pos=search(res,1,pos,N);
                 poset1=setbit(poset1,pos,0);
                 compt++;
                 }
     //At the end of this loop: if pos does not have 2 marks,
     // coched1=Max or 0.
     if(nbefore[pos]==1 || n==1)  ;
     else
     if(nbefore[pos]>1)       // pos = Max and children[pos] contain
                             //at least one element different from pos.
         pos=chooseCoched(poset, coched1, coched2, coched3);
     else if(ancetre[pos]&& ndown[pos]>=2 && nbefore[ancetre[pos]]
           ==ndown[pos]+1)
          pos=chooseCoched(poset, pos, coched1, coched2);
          //pos=element before Max, so we take the one before
     else if(!ancetre[pos] && ndown[pos]>2 && nbefore[item.max]){
           if(nbefore[coched1])
              pos=chooseCoched(poset, coched2, coched3, coched4);
              // coched1 = Max
           else  pos=chooseCoched(poset,coched1,coched2,coched3);
           }// If in the last repetition pos has not a mark
     else if(ndown[pos]>2)  // add !ancetre[pos] &&
          pos= chooseParent(poset, pos, prec1, prec2, prec3);
     return pos;
 }

//----------------------------------------------------------------------
//  ------------------           Theorem 1            ----------------
//----------------------------------------------------------------------
void Ideal1(unsigned long poset, Direction dir, int x, int mask,
                 int num, int n, queue &p, queue &q, int &listack)
```

```
{
  int min, z, mini;
  unsigned int poset1, res;
  Direction dir1;

  if(poset != 0){
      if(x)            // if x then the minimum is x which is a
          min = x;     // posponed minimum from a precedent step
      else
          min = findMinimal(poset);
      if(num == 1){
        poset1 = setbit(poset, min, 0);
        mask = setbit(mask, min, 1);

        if(dir==FORWARD) {
                       if(flag>=0) lemma3(mask,item.poset, listack);
                       print(mask, p, q, listack);
                       dir1=BACKWARD;}
        else dir1=FORWARD;
        num = Updateup(poset, min, num);
        mini=Min; Min=0;
        x=findMinimal(poset1);

        if(x) Ideal(poset1, dir1, x, mask, num, n-1, p, q,listack);

        if(dir==BACKWARD) {
                       if(flag>=0) lemma3(mask,item.poset, listack);
                       print(mask, p, q,  listack);}
        num = Recoverup(poset, min, num, mini);
        }
      else{ // num > 1
        poset1 = poset;
        poset1 = setbit(poset1, min, 0);
        x = findMinimal(poset1);
        res = setbit(mask, min, 1);

        if(dir == FORWARD){
          if(flag>=0) lemma3(res,item.poset, listack);
          print(res, p, q,  listack);
          num = Updateup(poset, min, num);
          mini=Min; Min=0;
          Ideal(poset1, BACKWARD, x, res, num, n-1, p, q, listack);
          num = Recoverup(poset, min, num, mini);
          num = Updatedown(poset, min, num);
          Ideal((~up[min])& poset, FORWARD, x, mask, num,n-nup[min]
                                                  , p, q,  listack);
          num = Recoverdown(poset, min, num);    .
          }
        else{// BACKWARD
          num = Updatedown(poset, min, num);
          Ideal((~up[min])& poset, BACKWARD, x, mask, num,
                                    n-nup[min], p, q,  listack);
          num = Recoverdown(poset, min, num);
          num = Updateup(poset, min, num);
          mini=Min; Min=0;
          Ideal(poset1, FORWARD, x, res, num, n-1, p, q,  listack);
          num = Recoverup(poset, min, num, mini);
          if(flag>=0) lemma3(res,item.poset, listack);
          print(res, p, q,  listack);
        }
      }
    }
  }

//----------------------------------------------------------------
//  ---------------            In               ---------------
//----------------------------------------------------------------
int In(unsigned long poset, int n){
    return(getbit(poset, n)==1);
}

//----------------------------------------------------------------
//  ---------------         Theorem 2           ---------------
//----------------------------------------------------------------
void Ideal2(unsigned long poset, Direction dir, int x, int y, int
             mask, int num, int n, queue &p, queue &q, int &listack)
{
  int min, mini;
  unsigned long d, res, res1, res2, poset1, poset2; //x is a marked
min=y;                                              // element
if(poset != 0){
  res=(down[min] & poset)|mask;
  poset1=~up[min] & poset;
  poset2=~down[min] & poset;
  if(x){
  d=down[min] & poset;
  d=setbit(d,min,0);d=setbit(d,x,0);
  }
  //---------> direction is FORWARD ---------------------------
  if(dir == FORWARD)
   if(num==1)
     if(nup[min]==n-1){ // we treat here either x or !x
         if (!x) x=findMinimal(poset);
         res1=setbit(mask, x, 1);
         if(flag>=0) lemma3(res1,item.poset, listack);
         print(res1, p, q,  listack);
         num=Updateup(poset, min, num); mini=Min;    Min=0;
         y=findMinimal(poset2);
         Ideal(poset2, FORWARD, y, res, num, n-2, p, q,  listack);
         if(flag>=0) lemma3(res,item.poset, listack);
        print(res, p, q,  listack);
        num=Recoverup( poset, min, num, mini);
        }
    else{                   // we treat here either x or !x
        if(!x) x=findMinimal(poset);
        num=Updatedown(poset, min, num);
        num=Updateup(poset1, x, num); mini=Min;    Min=0;
        y=findMinimal(setbit(poset2,x,0));
        num=Recoverup(poset1, x, num, mini);
        Ideal2(poset1, FORWARD, x, y, mask, num, n-nup[min], p, q,
                                                    listack);
        num=Recoverdown(poset, min, num);
        num=Updateup(poset, min, num); mini=Min;    Min=0;
        Ideal(poset2, FORWARD, y, res, num, n-2, p, q,  listack);
        if(flag>=0) lemma3(res,item.poset, listack);
        print(res, p, q,  listack);
        num=Recoverup( poset, min, num, mini);
        }
  else if(num==2 && x)
    if(!ancetre[item.max] || !In(poset, item.max) ||
                                      ndown[item.max]!=1) {
      if(d==0){
        y=findMinimal(setbit(poset,x,0));
        num=Updatedown(poset, min, num);
        lemma1(poset1, x, y, mask, num, n-nup[min], p, q,listack);
        num=Recoverdown(poset, min, num);
        num=Updateup(poset, min, num); mini=Min;    Min=0;
        Ideal(poset2, FORWARD, y, res, num, n-2, p, q,listack);
        }
      else{
        y=findMinimal(down[min] & poset);
        num=Updatedown(poset, min, num);
        lemma1(poset1, x, y, mask, num, n-nup[min], p, q,listack);
        num=Recoverdown(poset, min, num);
        num=Updateup(poset, min, num); mini=Min;    Min=0;
        Ideal(poset2, FORWARD, x, res, num, n-2, p, q,  listack);
        }
      if(flag>=0) lemma3(res,item.poset, listack);
      print(res, p, q,  listack);
      num=Recoverup( poset, min, num, mini);
      }
    else //In(poset, item.max) and ndown[item.max]=1
         if(nbefore[item.max]==1 || nbefore[item.max]==2)
           lemma1(poset, x, item.max, mask, num, n, p, q,listack);
    else if(nup[min]==n-2){
           res1=setbit(mask, x, 1);
           if(flag>=0) lemma3(res1,item.poset, listack);
           print(res1, p, q,  listack);

           res2=setbit(mask, item.max, 1);
        if(flag>=0) lemma3(res2,item.poset, listack);
        print(res2, p, q,  listack);

        res2=setbit(res2, x, 1);
        if(flag>=0) lemma3(res2,item.poset, listack);
        print(res2, p, q,  listack);

        num=Updateup( poset, min, num); mini=Min;    Min=0;
        if(num==1){
             res1=setbit(res, item.max, 1);
             if(flag>=0) lemma3(res1,item.poset, listack);
             print(res1, p, q,  listack);
             }
        else  if(num==2)
              {y=findMinimal(poset2);
              lemma1inv(poset2, y, item.max, res, num, n-2, p,
                                                   q, listack);}
        else // num>=3
            if(nbefore[item.max]==2)   ;
        else  {printf("Case not treated, see cas 8"); exit(1);}
        if(flag>=0) lemma3(res,item.poset, listack);
        print(res, p, q,  listack);
        num=Recoverup( poset, min, num, mini);
        }
   else{ //nup[min]!=n-2
      num=Updatedown(poset, min, num);
      num=Updateup(poset1, x, num); mini=Min;    Min=0;
      y=findMinimal(setbit(poset2,x,0));
      num=Recoverup(poset1, x, num, mini);
      Ideal2(poset1, FORWARD, x, y, mask, num, n-nup[min], p,
                                                   q, listack);
      num=Recoverdown(poset, min, num);
      num=Updateup(poset, min, num); mini=Min;    Min=0;
      Ideal(poset2, FORWARD, y, res, num, n-2, p, q,  listack);
      if(flag>=0) lemma3(res,item.poset, listack);
      print(res, p, q;  listack);
      num=Recoverup( poset, min, num, mini);
      }
  else // num>=3 or x=0
      if(x==0){ // num=2 or num>2
      x=findMinimal(down[min] & poset);//in the case where x!=0,
//x is the minimum of down[min] since ndown[item.max]=1, and num =2
      num=Updatedown(poset, min, num);
      Ideal(poset1, BACKWARD, x, mask, num, n-nup[min], p, q
                                                    ,listack);
      num=Recoverdown(poset, min, num);
      num=Updateup(poset, min, num); mini=Min;    Min=0;
      y=findMinimal(poset2);
      Ideal(poset2, FORWARD, y, res, num, n-2, p, q,  listack);
      if(flag>=0) lemma3(res,item.poset, listack);
```

```
        print(res, p, q,  listack);
        num=Recoverup( poset, min, num, mini);
        }
  else{// x!=0 and num>2
      if(d==0){
        y=findMinimal(setbit(poset,x,0));
        num=Updatedown(poset, min, num);
        lemma2(poset1, x, y, mask, num, n-nup[min], p, q,listack);
        num=Recoverdown(poset, min, num);
        num=Updateup(poset, min, num); mini=Min;    Min=0;
        Ideal(poset2, FORWARD, y, res, num, n-2, p, q,  listack);
        }
      else{
        y=findMinimal(down[min] & poset);
        num=Updatedown(poset, min, num);
        lemma2(poset1, x, y, mask, num, n-nup[min], p, q,listack);
        num=Recoverdown(poset, min, num);
        num=Updateup(poset, min, num); mini=Min;    Min=0;
        Ideal(poset2, FORWARD, x, res, num, n-2, p, q,  listack);
        }
      if(flag>=0) lemma3(res,item.poset, listack);
      print(res, p, q,  listack);
      num=Recoverup( poset, min, num, mini);
      }
 else //-------------------------dir=BACKWARD-----------------------
  if(num==1)
    if(nup[min]==n-1){          // we treat here either x or !x
        if(flag>=0) lemma3(res,item.poset, listack);
        print(res, p, q,  listack);
        num=Updateup(poset, min, num); mini=Min;    Min=0;
        y=findMinimal(poset2);
        Ideal(poset2, BACKWARD, y, res, num, n-2, p, q,  listack);
        num=Recoverup( poset, min, num, mini);
        if(!x) x=findMinimal(poset);
        res1=setbit(mask, x, 1);
        if(flag>=0) lemma3(res1,item.poset, listack);
        print(res1, p, q,  listack);
        }
    else{                       // we treat here either x or !x
        if(flag>=0) lemma3(res,item.poset, listack);
        print(res, p, q,  listack);
        if(!x) x=findMinimal(poset);
        num=Updateup(poset, min, num); mini=Min;    Min=0;
        y=findMinimal(~up[min] & poset2);
        Ideal(poset2, BACKWARD, y, res, num, n-2, p, q,  listack);
        num=Recoverup( poset, min, num, mini);

        num=Updatedown(poset, min, num);
        Ideal2(poset1, BACKWARD, x, y, mask, num, n-nup[min], p, q
                                                      , listack);
```

```
         num=Recoverdown(poset, min, num);
         }
  else  if(num==2 && x)
    if(!ancetre[item.max] || !In(poset, item.max) ||
                                              ndown[item.max]!=1) {
       if(flag>=0) lemma3(res,item.poset, listack);
       print(res, p, q,  listack);
       if(d==0){
         y=findMinimal(setbit(poset,x,0));
         num=Updateup(poset, min, num); mini=Min;    Min=0;
         Ideal(poset2, BACKWARD, y, res, num, n-2, p, q,  listack);
         num=Recoverup( poset, min, num, mini);

         num=Updatedown(poset, min, num);
         lemma1(poset1, y, x, mask, num, n-nup[min], p, q,listack);
         num=Recoverdown(poset, min, num);
         }
       else{//Here num=2 and x and ndown[item.max]!=1 and d=0
         y=findMinimal(down[min] & poset);
         num=Updateup(poset, min, num); mini=Min;    Min=0;
         Ideal(poset2, BACKWARD, x, res, num, n-2, p, q,  listack);
         num=Recoverup( poset, min, num, mini);

         num=Updatedown(poset, min, num);
         lemma1(poset1, y, x, mask, num, n-nup[min], p, q,listack);
         num=Recoverdown(poset, min, num);
         }
       }
    else
         if(nbefore[item.max]==1 || nbefore[item.max]==2)
           lemma1(poset, item.max, x, mask, num, n, p, q,listack);
    else  if(nup[min]==n-2){
           if(flag>=0) lemma3(res,item.poset, listack);
           print(res, p, q,  listack);
           num=Updateup( poset, min, num); mini=Min;    Min=0;
           if(num==1){
                 res1=setbit(res, item.max, 1);
                 if(flag>=0) lemma3(res1,item.poset, listack);
                 print(res1, p, q,  listack);
                 }
           else  if(num==2)
                  {y=findMinimal(poset2);
                   lemma1inv(poset1, x, item.max, res, num, n-2, p
                                                       , q, listack);}
           else  // num>=3
                 if(nbefore[item.max]==2)  ;
           else    {printf("Case not treated, see case 8"); exit(1)↩
               ;}
           num=Recoverup( poset, min, num, mini);
```

```
            res2=setbit(res2, x, 1);res2=setbit(mask, item.max, 1);
            if(flag>=0) lemma3(res2,item.poset, listack);
            print(res2, p, q,  listack);

            res2=setbit(res2, x, 0);
            if(flag>=0) lemma3(res2,item.poset, listack);
            print(res2, p, q,  listack);

            res1=setbit(mask, x, 1);
            if(flag>=0) lemma3(res1,item.poset, listack);
            print(res1, p, q,  listack);
            }
    else{ //nup[min]!=n-2
          if(flag>=0) lemma3(res,item.poset, listack);
          print(res, p, q,  listack);
          if(!x) x=findMinimal(poset);
          num=Updateup(poset, min, num); mini=Min;    Min=0;
          y=findMinimal(~up[min] & poset2);
          Ideal(poset2, BACKWARD, y, res, num, n-2, p, q,  listack);
          num=Recoverup( poset, min, num, mini);

          num=Updatedown(poset, min, num);
          Ideal2(poset1, BACKWARD, x, y, mask, num, n-nup[min], p,
                                                         q,  listack);
          num=Recoverdown(poset, min, num);
          }
  else // num>=3 or x=0
         if(x==0){ // num=2 or num>2
          if(flag>=0) lemma3(res,item.poset, listack);
          print(res, p, q,  listack);
          num=Updateup(poset, min, num); mini=Min;    Min=0;
          y=findMinimal(poset2);
          Ideal(poset2, BACKWARD, y, res, num, n-2, p, q,  listack);
          num=Recoverup( poset, min, num, mini);

          x=findMinimal(down[min] & poset);
          num=Updatedown(poset, min, num);
          Ideal(poset1, FORWARD, x, mask, num, n-nup[min], p, q,
                                                            listack);
          num=Recoverdown(poset, min, num);
          }

  else{ // x!=0 and num>2
        if(flag>=0) lemma3(res,item.poset, listack);
        print(res, p, q,  listack);
        if(d==0){
          y=findMinimal(setbit(poset,x,0));
          // We may have a problem if y is just before Max.
          num=Updateup(poset, min, num); mini=Min;    Min=0;
          Ideal(poset2, BACKWARD, y, res, num, n-2, p, q,  listack);
          num=Recoverup( poset, min, num, mini);

          num=Updatedown(poset, min, num);
          lemma2(poset1, y, x, mask, num, n-nup[min], p, q,listack);
          num=Recoverdown(poset, min, num);
          }
        else{//Here num=2 and x and udown[item.max]!=1 and d=0
          y=findMinimal(down[min] & poset);
          num=Updateup(poset, min, num); mini=Min;    Min=0;
          Ideal(poset2, BACKWARD, x, res, num, n-2, p, q,  listack);
          num=Recoverup( poset, min, num, mini);

          num=Updatedown(poset, min, num);
          lemma2(poset1, y, x, mask, num, n-nup[min], p, q,listack);
          num=Recoverdown(poset, min, num);
          }
        }

  }// end if(poset!=0)
}// end Ideal2

//------------------------------------------------------------------
//   ------------           Lemma 1                  ------------
//------------------------------------------------------------------
void lemma1(unsigned long poset, int x, int y, int mask, int num,
                              int n, queue &p, queue &q, int &listack)
{
  int min1=0, min2, mini1, mini;
  unsigned int poset1, res;

  if(poset != 0){
    if(!x || !y) {printf("In lemma 1 x or y is null\n"); exit(1); }
    poset1=(~up[y]) & poset;
    poset1 = setbit(poset1, x, 0);
    res = setbit(mask, x, 1);
    if (poset1==0) {
                    if(flag>=0) lemma3(res,item.poset, listack);
                    print(res, p, q,  listack);
                    }
    else {
        num = Updatedown(poset, y, num);
        num = Updateup((~up[y]) & poset, x, num);
        mini=Min; Min=0;
                                                  //-----------------//
                                                  //   First cycle   //
                                                  //-----------------//
        min1 = findMinimal(poset1);
        if(flag>=0)
          lemma3(res,item.poset, listack);print(res, p, q,listack);
        Ideal(poset1, BACKWARD, min1, res, num, n-nup[y]-1, p, q,
                                                        listack);
    num = Recoverup((~up[y]) & poset, x, num, mini);
    num = Recoverdown(poset, y, num);
    }

poset1=poset;
poset1=setbit(poset1, x, 0); poset1=setbit(poset1, y, 0);
res = setbit(mask, x, 1);
res = setbit(res, y, 1);
if (poset1==0 ) {
                  if(flag>=0) lemma3(res,item.poset, listack);
                  print(res, p, q,  listack);
                  }
else {

  num = Updateup(poset, x, num);
  mini=Min; Min=0;
  num = Updateup(setbit(poset, x, 0), y, num);
  mini1=Min; Min=0;

                                                    //------------------//
                                                    //  Second cycle   //
                                                    //-----------------//
    if(!mini1)
         min1 = findMinimal(poset1);

    Ideal(poset1, FORWARD, min1, res, num, n-2, p, q, listack);

    if(flag>=0) lemma3(res,item.poset, listack);
    print(res, p, q, listack);
    num = Recoverup(setbit(poset, x, 0), y, num, mini1);
    num = Recoverup(poset, x, num, mini);
}

poset1 = (~up[x]) & poset;
poset1 = setbit(poset1, y, 0);
res = setbit(mask, y, 1);
if (poset1 ==0) {
                  if(flag>=0) lemma3(res,item.poset, listack);
                  print(res, p, q,  listack); }
else {
  num = Updatedown(poset, x, num);
  num = Updateup((~up[x]) & poset, y, num);
  mini=Min; Min=0;
                                                    //------------------//
      min2 = findMinimal(poset1);                   //  Third cycle    //
                                                    //------------------//
  Ideal(poset1, FORWARD, min2,res,num,n-nup[x]-1,p, q,listack);

  if(flag>=0) lemma3(res,item.poset, listack);
```

```
      print(res, p, q, listack);

      num = Recoverup((~up[x]) & poset, y, num, mini);
      num = Recoverdown(poset, x, num);

    }
//Ideal(0, FORWARD, 0, mask, num-2, n-nup[x]-nup[y], p, q,listack);
  }
}

//-----------------------------------------------------------------------
//   ------------           Lemmalinv                  ------------
//-----------------------------------------------------------------------
void lemmalinv(unsigned long poset, int x, int y, int mask, int num
                         , int n, queue &p, queue &q, int &listack)
{
  int min1=0, min2, mini1, mini;
  unsigned int poset1, res;

  if(poset != 0){
  if(!x || !y) {printf("In lemmalinv x or y is null\n"); exit(1); }

    poset1=poset;
    poset1=setbit(poset1, x, 0); poset1=setbit(poset1, y, 0);
    res = setbit(mask, x, 1);
    res = setbit(res, y, 1);
    if (poset1==0 ) {
                     if(flag>=0) lemma3(res,item.poset, listack);
                     print(res, p, q,  listack);}
    else {
      if(flag>=0) lemma3(res,item.poset, listack);
      print(res, p, q,listack);
      num = Updateup(poset, x, num);
      mini=Min; Min=0;
      num = Updateup(setbit(poset, x, 0), y, num);
      mini1=Min; Min=0;
                                                     //---------------//
                                                    //  Second cycle  //
                                                   //---------------//
      min1 = findMinimal(poset1);

      Ideal(poset1, BACKWARD, min1, res, num, n-2, p, q,  listack);

      num = Recoverup(setbit(poset, x, 0), y, num, mini1);
      num = Recoverup(poset, x, num, mini);
    }

    poset1=(~up[y]) & poset;
    poset1 = setbit(poset1, x, 0);
    res = setbit(mask, x, 1);
    if (poset1==0) {
                      if(flag>=0) lemma3(res,item.poset, listack);
                      print(res, p, q,  listack);}
    else {
      num = Updatedown(poset, y, num);
      num = Updateup((~up[y]) & poset, x, num);
      mini=Min; Min=0;
                                                  //---------------------//
                                                  //   First cycle       //
                                                  //---------------------//
      Ideal(poset1, FORWARD, min1, res,num,n-nup[y]-1,p,q,listack);
      if(flag>=0) lemma3(res,item.poset, listack);
      print(res, p, q,  listack);
      num = Recoverup((~up[y]) & poset, x, num, mini);
      num = Recoverdown(poset, y, num);
    }
    poset1 = (~up[x]) & poset;
    poset1 = setbit(poset1, y, 0);
    res = setbit(mask, y, 1);
    if (poset1 ==0) {
                       if(flag>=0) lemma3(res,item.poset, listack);
                       print(res, p, q,  listack); }
    else {printf("There is a problem in lemma1inv"); exit(1);}
                                                  //---------------------//
                                                  //   Third cycle       //
                                                  //---------------------//

   //Ideal(0, FORWARD, 0, mask, num-2,n-nup[x]-nup[y],p,q,listack);
  }
}

//-----------------------------------------------------------------------
//            ----------------        Lemma 2        ---------------
//-----------------------------------------------------------------------
void lemma2(unsigned long poset, int x, int y, int mask, int num,
                              int n, queue &p, queue &q, int &listack)
{
  int z, t, t1=0, num1=num, mini1, mini ;
  unsigned int poset1, res;
  // We may have nbefore[z]=1 or >3 when num=3. See d3 in Ideal.
  if(poset != 0){
    if(!x || !y) {printf("In lemma 2 x or y is null\n"); exit(1); }
    poset1=(~up[y]) & poset;
    z=findMinimal((~up[x]) & poset1);
    if(num>3) t1=findMinimal((~up[x]) & (~up[z]) & poset1);

    poset1 = setbit(poset1, x, 0);
    res = setbit(mask, x, 1);
    num = Updatedown(poset, y, num);
    num = Updateup((~up[y]) & poset, x, num);
```

```
mini=Min; Min=0;
if(flag>=0) lemma3(res,item.poset, listack);
print(res, p, q,  listack);    // Print x
                                                  ////////////////////
                                                 //  First cycle   //
                                                ////////////////////
   Ideal(poset1, BACKWARD, z, res,num, n-nup[y]-1,p,q,listack);

num = Recoverup((~up[y]) & poset, x, num, mini);
num = Recoverdown(poset, y, num);
                                                  ////////////////////
                                                  // Second cycle  //
                                                 ////////////////////
poset1 = (~up[x]) & poset;
poset1 = (~up[y]) & poset1;
num = Updatedown(poset, x, num);
num = Updatedown((~up[x])& poset, y, num);

if(num1==3){
     poset1= setbit(poset1, z, 0);

     num=Updateup((~up[x])&(~up[y])& poset, z, num);
     mini=Min; Min=0;
     t=findMinimal(poset1);  //This t might be zero
     res=setbit(mask, z, 1);

     if(t && nbefore[ancetre[z]]==1  && ancetre[z]!=t)
               //If ancetre[z]=Max, Max will be considered t
         Ideal(poset1, FORWARD, ancetre[z], res, num,
                             n-nup[x]-nup[y]-1, p, q,  listack);
     else
         Ideal(poset1, FORWARD, t, res, num, n-nup[x]-nup[y]-1,
                                               p, q,  listack);

     if(flag>=0) lemma3(res,item.poset, listack);
     print(res, p, q,  listack);
     num=Recoverup((~up[x])&(~up[y])& poset, z, num, mini);
     t=0;
     }
else {
  if(num1==4)
    lemma1(poset1, z,t1,mask,num, n-nup[x]-nup[y],p,q,listack);
  else
    lemma2(poset1, z,t1,mask,num, n-nup[x]-nup[y],p,q,listack);
    }
num = Recoverdown((~up[x])& poset, y, num);
num = Recoverdown(poset, x, num);
poset1=poset;
poset1=setbit(poset1, x, 0); poset1=setbit(poset1, y, 0);
res = setbit(mask, x, 1);
```

```
    res = setbit(res, y, 1);
    num = Updateup(poset, x, num);
    mini=Min; Min=0;
    num = Updateup(setbit(poset, x, 0), y, num);
    mini1=Min;  Min=0;
                                                            //--------------------//
                                                            //    Third cycle     //
                                                            //--------------------//
    if(t1)
        Ideal(poset1, FORWARD, t1, res, num, n-2, p, q,  listack);
    else
        Ideal(poset1, FORWARD, z, res, num, n-2, p, q,  listack);


    if(flag>=0) lemma3(res,item.poset, listack);
    print(res, p, q,  listack);
    num = Recoverup(setbit(poset, x, 0), y, num, mini1);

    num = Recoverup(poset, x, num, mini);

    poset1 = (~up[x]) & poset;
    poset1 = setbit(poset1, y, 0);
    res = setbit(mask, y, 1);
    num = Updatedown(poset, x, num);

    num = Updateup((~up[x]) & poset, y, num);
    mini=Min; Min=0;

                                                            //--------------------//
                                                            //   Fourth cycle     //
                                                            //--------------------//
    Ideal(poset1, FORWARD, z, res, num, n-nup[x]-1, p, q, listack);

    if(flag>=0) lemma3(res,item.poset, listack);
    print(res, p, q,  listack);

    num = Recoverup((~up[x]) & poset, y, num, mini);
    num = Recoverdown(poset, x, num);
  }
}

//-----------------------------------------------------------------------
//  -----   To link the first poset to the second in Lemma 3 ----------
//-----------------------------------------------------------------------
void lemma3(unsigned int mask1, unsigned int poset, int &listack){
   unsigned int res;
   int min;

     if(next)
       {if(next==mask1)
```

```
            listack=1 ;                    // enqueuing
        else  {printf("-----> Exit because next != mask"); exit(1); }
        next=0;
        if(!flag) flag--;
      }
    else
        {min=item.min;          // item.min is the minimum of Theorem 3
           // mask1 is an ideal in the first part resulting from the
        res=mask1; res=setbit(res,min,1);//application of Theorem 3
        if(res==item.mask){//item.mask is down[min] in Theorem 3.<-
           case 1.
          flag--;
          direc.PPush(1);
          next=setbit(mask1,item.max,0);//here next is given a value
          }
        else if(setbit(res,item.max,1)== item.mask) {        // case 2
          flag--;
          direc.PPush(0);
          next=setbit(mask1,item.max,1);//here next is given a value
            }
         }
}

//-------------------------------------------------------------------
//void Ideal3(unsigned long poset,unsigned long mask,int n,int num)
//-------------------------------------------------------------------
void Ideal3(unsigned long poset, Direction dir, int x, int mask,
          int num,int min, int n, queue &p, queue &q, int &listack)
{
    queue r;
    int aux, mini;
    Direction dir1;

            stack.Push(item);
            if(flag==-1) flag=1; else flag++;
            num=Updatedown(poset, min, num);
            item.max=Max; item.min = min ; item.poset=poset;
            item.mask=(down[min]& poset)|mask;


             Ideal((~up[min])& poset, dir, x, mask, num, n-nup[min]
                                                , p, q,  listack);
             num=Recoverdown(poset, min, num);
             if(nbefore[item.max]<0)
               {printf("nbefore[%d] is negatif",item.max);exit(1);}
//-------------------------------------------------------------------
             listack=0;
             while(!q.isempty()){
                     aux=q.dequeue(); r.enqueue(aux) ;
                     }
            num=Updateup(poset, min, num);
            mini=Min;  Min=0;
            unsigned long res=item.mask;

            x=findMinimal((~down[min])& poset);
            stack.Pop(item);

            unsigned int d;
            direc.PPop(d);
            if(d) dir1=FORWARD; else dir1=BACKWARD;
            if(dir1==BACKWARD){
                       if(flag>=0) lemma3(res, poset, listack);
                       print(res, p, q,  listack);
                       }

            Ideal((~down[min])& poset, dir1, x, res, num,
                                 n-ndown[min], p, q,  listack);
            num=Recoverup(poset, min, num, mini);

            if(dir1==FORWARD){
                      if(flag>=0) lemma3(res, poset, listack);
                      print(res, p, q, listack);
                      }
           if(q.isempty())
               while(!r.isempty()){
                       aux= r.dequeue() ;  p.enqueue(aux);
                       }
          else
               while(!r.isempty()){
                       aux= r.dequeue() ;  q.enqueue(aux);
                       }
}

//------------------------------------------------------------------
//void Ideal(unsigned long poset, unsigned long mask,int n,int num)
//   ------------             Theorem 3               ------------
//------------------------------------------------------------------
void Ideal(unsigned long poset, Direction dir, int x, int mask, int
                        num, int n, queue &p, queue &q, int &listack)
 {
    int n0, min, z, mini, i, max0, min0;
    unsigned long poset0, mask0;
    Direction dir1;
    if(n>0){
      n0=(n+1)/2;
      if(nbefore[item.max]==1)
               min=findMinimal(poset & children[item.max]);
      else  min = findMedian(poset, n0);

     //case where the element preceding min is with one
    // mark and ndown[min]>2
     if(x && ndown[min]>2 && setbit(parents[x],min,0)!=parents[x]
       && setbit(0,x,1)>setbit(setbit(down[min]&poset,min,0),x,0))
              min= x;
// x can not be Max

     if(min==x)    //Theorem 1, case 1, two minima; one is min and
                  // the other x is marked; it may be min=x
         Ideal1(poset, dir, x, mask, num, n, p, q,  listack);

     else if(x && ndown[min]==1)      //Theorem 1, case c,  x!=min
            if(num==2)
              if(dir==FORWARD)
                lemma1(poset, x, min, mask, num, n, p,q,listack);
              else
                lemma1(poset, min, x, mask, num, n, p,q,listack);
            else if(num==3) {     //Theorem 1, case d.
                 z=findMinimal((~up[x])&(~up[min])& poset);
               if(nbefore[z]) //&& ndown[z]==1
               // z is a minimum thus ndown[z]=1
                  {if(ancetre[x] && ancetre[min] &&nbefore[z]==3)
                     if(dir==FORWARD)                        // d1
                      lemma2(poset,x,min,mask,num,n,p,q,listack);
                     else
                      lemma2(poset,min,x,mask,num,n,p,q,listack);
                  else if(nbefore[z]==2)    //d2
                     if(dir==FORWARD)
                      lemma2(poset,x,min,mask,num,n,p,q,listack);
                     else
                      lemma2(poset,min,x,mask,num,n,p,q,listack);
                  else    //d3
                      Ideal1(poset,dir,x,mask,num,n,p,q,listack);
                   }
               else   //d4
                  if(dir==FORWARD)
                     lemma2(poset,x,min,mask,num,n,p,q,listack);
                  else
                      lemma2(poset,min,x,mask,num,n,p,q,listack);
                  }

            else {  //num>3          Theorem 1, case d5,
               if(dir==FORWARD)
                    lemma2(poset, x, min, mask,num,n,p,q,listack);
               else
                    lemma2(poset, min, x, mask,num,n,p,q,listack);
                    }

     else if(ndown[min]==1)        //Theorem 1, case e.
               Ideal1(poset, dir, min, mask, num, n,p,q,listack);
```

```
      else if(ndown[min]==2)         //Theorem 2, case g,
               Ideal2(poset, dir, x, min, mask, num,n,p,q,listack);
               ////////////////
      else{  // Theorem 3 //
            ////////////////
              if(listack==0)
                Ideal3(poset, dir, x, mask, num,min,n,p,q,listack);
              else{int aux;
                  queue p1, q1;
                  listack=0;
                  Ideal3(poset,dir,x,mask,num,min,n,p1,q1,listack);
                  while(!p1.isempty()){
                      aux=p1.dequeue();
                      q.enqueue(aux);
                      }
                  while(!q1.isempty()){
                       aux=q1.dequeue();
                       q.enqueue(aux);
                       }
                  listack=1;
                  }
            }//end else Ideal3
       }
    }

//----------------------------------------------------------------
//////////////////////////////////////////////////////////////////
int main(int argc, char* argv[])
{
  FILE *input;
  int i, left, right, num_zeros;
  unsigned long poset;
  char *fname;
  char *mode;
  queue p, q;
  int listack=0;

  if(argc != 2){
     fprintf(stderr, "Usage: gendIdeal in_file_name\n");
     exit(1);
  }

  printf("Ideals in Gray Code Order\n");

  fname = argv[1];    //printf("fname\n").
  if((input = fopen(fname, "r")) == NULL){
    fprintf(stderr, "Error: could not read file %s\n", fname);
    exit(1);
  }
  // read number of elements in the set
  fscanf(input, "%d" ,&N);
  printf("\n The number of elements in the poset is :  %d\n\n",N);

  for(i=1; i<=N ; i++){
    table[i] = 0; parents[i]=0;
    table[i] = setbit(table[i], i, 1); //parents[i] = table[i];
  }

  // store partial orderings
  int comma = 0;
  printf(" The poset is \n { ");
  do{
    fscanf(input, "%d %d", &left, &right);
    table[left] = setbit(table[left], right, 1);
    parents[left]=setbit(parents[left], right, 1);
    if(left)
      if(comma) printf(" , %d<%d", left, right);
      else      printf("%d<%d", left, right);
    comma = 1;
  }while(left != 0);
  printf(" }\n");
  fclose(input);
  printf("\n\nIdeals in Gray Code Order\n");
  // initialize ideal and poset
  poset = 0;
  for(i=1; i <= N; i++)
    poset = setbit(poset, i, 1);

  Process();
  num_zeros = 0;
  for(i=1; i <= N; i++){
    if(ndown[i] == 1)
       num_zeros ++;
  }

long  ratio;
clock_t time1, time2;
ratio = 1000/CLK_TCK;
time1 = clock();

print(0, p, q,  listack);
//---------------------------------------------------------------------------
Ideal(poset, FORWARD, 0, 0, num_zeros, N, p, q,  listack);
    while(!p.isempty())
                  prints(p.dequeue());

printf("%d Ideals in total\n", counter);
  time2 = clock();
  printf(" %ld\n",ratio*(long) time2 - ratio*(long) time1);
  return 0;
}
```

A.2 Ruskey's program

```
//--------------------------------------------------------------------//
//   This program is written in Dev-C++, version 4.9.9.2              //
//--------------------------------------------------------------------//

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#define MAX_POINTS 50
#define STANDARD 0
#define GRAY 1

typedef enum{FORWARD, BACKWARD}Direction;

unsigned int table[100];
unsigned int up[100];
unsigned int steiner[100];
int flip ;
int N;
int counter = 0;
int Y;

unsigned getbit(unsigned int word, int n)
{
  return (word >> n) & 01 ;
}

unsigned int setbit(unsigned int word, int n, unsigned v)
{
  if (v != 0)
    return word | (01 << n);
  else
    return word & ~(01 << n);
}

void Process()
{
  int i, j, k, count, bit;
  // transitive closure
  for (k=1; k <= N; k++){
    for (i=1; i <= N; i++){
      for (j=1; j <= N; j++){
        bit = getbit(table[i], j) | (getbit(table[i], k) &
                                       getbit(table[k], j));
        table[i] = setbit(table[i], j, bit);
      }
    }
  }
  for (i=1; i <= N; i++){
     up[i] = 0;
     count = 0;
     for (j=1; j <= N; j++){
         if(getbit(table[i], j)){
           up[i] = setbit(up[i], j, 1);
         }
         if(getbit(table[j], i)){
            count++;
         }
     }
    steiner[i] = count - 1;
  }
}

/*
 * prints the ideal in the format {i, j, .. }
 */
void print(int number)
{
  int i, flag = 0;
  if(!flip)
     flip = 1;
  else{
     flip = 0;
     printf("{");
     for(i=1 ; i <= N ; i++)
        if(getbit(number, i)){
           if(flag)
              printf(" %d", i);
           else
              printf("%d", i);
           if(!flag)
              flag = 1;
        }
        printf("}\n");
        counter++;
     }
 }

int search(unsigned int mask, int value, int start, int end){
   int pos;
   if(start > end){
      printf("index error\n");
      exit(1);
   }
   if(start == end)
```

```
      return start;
    if(end == start + 1){
      if(getbit(mask, end) == value)
          return end;
      else if (getbit(mask, start) == value)
          return start;
      else{
        printf("\nfatal error!!!\n");
        printf("mask is %d, start is %d, end is %d",mask,start,end);
        exit(1);}
    }
    else{
       pos = (start + end) / 2;
       if(getbit(mask, pos) == value)
         return (search(mask, value, pos, end));
       else
         return (search(mask, value, start, pos));
    }
}

// find the minimal element in the poset
int findMinimal(unsigned long poset)
{
  unsigned long res;
  int pos = 0;

  while(poset !=0){
    res = poset ^ (poset -1);
    pos = search(res, 1, pos, N);
    if(steiner[pos] == 0)
        break;
    poset = setbit(poset, pos, 0);
  }
  return pos;
}

// update the list for finding minimal element
int Update(int poset, int min, int num)
{
  long unsigned mask, res;
  int pos;
  mask = poset & up[min];
  pos = 0;
  while(mask !=0){
    res = (mask -1) ^ mask;
    pos = search(res, 1, pos, N);
    steiner[pos]--;
    if(steiner[pos] == 0)
      num++;
    mask = setbit(mask, pos, 0);
  }
  return num -1;
}

int Recover(int poset, int min, int num){
  long unsigned mask, res;
  int pos;
  mask = poset & up[min];
  pos = 0;
  while(mask != 0){
    res = (mask -1) ^ mask;
    pos = search(res, 1, pos, N);
    if(steiner[pos] == 0)
      num--;
    steiner[pos]++;
    mask = setbit(mask, pos, 0);
  }
  return num;
}

// generate ideals recursively
void Ideal(unsigned long poset, Direction dir, int two_flag,
                                                        int mask, int num)
{
  int min;
  unsigned int poset1, res;

  if(poset != 0){
    if(two_flag && num > 1)
      min = Y;
    else
      min = findMinimal(poset);
    if(num == 1)
      {
        poset = setbit(poset, min, 0);
        num = Update(poset, min, num);
        mask = setbit(mask, min, 1);
        print(mask);
        Ideal(poset, dir, 0, mask, num);
        print(mask);
        num = Recover(poset, min, num);
      }
    else
      {
        poset1 = poset;
        poset1 = setbit(poset1, min, 0);
        Y = findMinimal(poset1);
        res = setbit(mask, min, 1);

        if(dir == FORWARD){
          print(res);
          print(res);
          num = Update(poset1, min, num);
          Ideal(poset1, BACKWARD, 1, res, num);
          num = Recover(poset1, min, num);
          poset = (~up[min]) & poset;
          Y = findMinimal(poset);
          Ideal(poset, FORWARD, 1, mask, num);
        }
        else{
          poset = (~up[min]) & poset;
          Ideal(poset, BACKWARD, 1, mask, num -1);
          Y = findMinimal(poset1);
          num = Update(poset1, min, num);
          Ideal(poset1, FORWARD, 1, res, num);
          num = Recover(poset1, min, num);
          print(res);
          print(res);
        }
      }
    }
  }

int main(int argc, char* argv[])
{
 long  ratio;
 clock_t time1, time2;
 ratio = 1000/CLK_TCK;


  FILE *input;
  int i, left, right, num_zeros;
  unsigned long poset;
  char *fname;
  char *mode;

  if(argc != 2){
     fprintf(stderr, "Usage: gendIdeal in_file_name\n");
     exit(1);
  }

  printf("Ideals in Gray Code Order\n");

  fname = argv[1];    //printf("fname\n");
  if((input = fopen(fname, "r")) == NULL){
    fprintf(stderr, "Error: could not read file %s\n", fname);
    exit(1);
  }

  // read number of elements in the set
  fscanf(input, "%d" ,&N);
  printf("\n The number of elements in the poset is . %d\n\n",N);

  for(i=1; i<=N ; i++){
    table[i] = 0;
    table[i] = setbit(table[i], i, 1);
  }

  // store partial orderings
  int comma = 0;
  printf(" The poset is \n { ");
  do{
    fscanf(input, "%d %d", &left, &right);
    table[left] = setbit(table[left], right, 1);
    if(left)
      if(comma) printf(" . %d<%d", left, right);
      else      printf("%d<%d", left, right);
    comma = 1;
  }while(left != 0);
  printf(" }\n");
  fclose(input);
  printf("\n\nIdeals in Gray Code Order\n");

  // initialize ideal and poset
  poset = 0;
  for(i=1; i <= N; i++)
    poset = setbit(poset, i, 1);

  Process();
  num_zeros = 0;
  for(i=1; i <= N; i++){
    if(steiner[i] == 0)
       num_zeros ++;
  }
  flip = 1;

  time1 = clock();
  print(0);
  Ideal(poset, FORWARD, 0, 0, num_zeros);

  printf("%d Ideals in total\n", counter);
  time2 = clock();
  printf(" %ld\n",ratio*(long) time2 - ratio*(long) time1);
  return 0;
}
```

A.3 Our implementation of Squire's algorithm

```
//------------------------------------------------------------------//
//   This program is written in Dev-C++, version 4.9.9.2            //
//------------------------------------------------------------------//

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#define MAX_POINTS 50

unsigned int table[100];
unsigned int up[100], down[100];
int nup[100], ndown[100];
int N;
int counter = 0;
int Y;

unsigned getbit(unsigned int word, int n)
{
  return (word >> n) & 01 ;
}

unsigned int setbit(unsigned int word, int n, unsigned v)
{
  if (v != 0)
    return word | (01 << n);
  else
    return word & ~(01 << n);
}
//-------------------------------------------------------------------
void Process()
{
  int i, j, k, count, bit;
  // transitive closure
  for (k=1; k <= N; k++){
    for (i=1; i <= N; i++){
      for (j=1; j <= N; j++){
        bit = getbit(table[i], j) | (getbit(table[i], k)
                                  & getbit(table[k], j));
        table[i] = setbit(table[i], j, bit);
      }
    }
  }
  for (i=1; i <= N; i++){ nup[i]=0; ndown[i]=0; }
  for (i=1; i <= N; i++){
     up[i] = 0, down[i]=0;
```

```
      count = 0;
      for (j=1; j <= N; j++){
          if(getbit(table[i], j)){
            up[i] = setbit(up[i], j, 1);
            nup[i]++;
          }
          if(getbit(table[j], i)){
             down[i]=setbit(down[i],j,1);
             count++;
             ndown[i]++;
          }
      }
   }
}
//--------------------------------------------------------------------------
// ------       prints the ideal in the format {i, j, ... } ---------------
//--------------------------------------------------------------------------
void print(unsigned int number)
{
  int i, flag = 0;

      printf("{");
      for(i=1 ; i <= N ; i++)
         if(getbit(number, i)){
            if(flag)
               printf(" %d", i);
            else
               printf("%d", i);
            if(!flag)
               flag = 1;
         }
         printf("}\n");
         counter++;
 }
//--------------------------------------------------------------------------
int search(unsigned int mask, int value, int start, int end){
   int pos;
   if(start > end){
      printf("index error\n");
      exit(1);
   }
   if(start == end)
     return start;
   if(end == start + 1){
     if(getbit(mask, end) == value)
         return end;
     else if (getbit(mask, start) == value)
         return start;
     else{
       printf("\nfatal error!!!\n");
```

```
        printf("mask is %d, start is %d, end is %d",mask,start,end);
        exit(1);}
    }
    else{
       pos = (start + end) / 2;
       if(getbit(mask, pos) == value)
         return (search(mask, value, pos, end));
       else
         return (search(mask, value, start, pos));
    }
}
//-----------------------------------------------------------------------
// ---------- find the number of elements of poset ----------
//-----------------------------------------------------------------------
int NumElement(int poset)
{
  long unsigned mask, res;
  int pos, num =0;
  mask = poset;
  pos = 0;
  while(mask !=0){
    res = (mask -1) ^ mask;
    pos = search(res, 1, pos, N);
    num++;
    mask = setbit(mask, pos, 0);
  }
  return num;
}
//-----------------------------------------------------------------------
// ------------------ find median element ------------------
//-----------------------------------------------------------------------
int findMedian(unsigned long poset, int n)//find the median element
 {
      unsigned long res, poset1;
      int pos=0, compt=0;
      poset1=poset;
      while(compt!=n){
                        res=poset1^(poset1-1);
                        pos=search(res,1,pos,N);
                        poset1=setbit(poset1,pos,0);
                        compt++;//printf("compt=%d\n",compt);
                        }
      return pos;
 }
//-----------------------------------------------------------------------
// generate ideals recursively ---------------- Squire ----------
//-----------------------------------------------------------------------
void Ideal(unsigned long poset, int mask, int n)
{
   int min, n0;
```

```
    unsigned long res;
    if(poset == 0) print(mask);
    else{
      n0=(n+1)/2;
      min = findMedian(poset, n0);
      res=mask |(down[min]& poset);
      Ideal((~up[min])& poset, mask, n-NumElement(up[min]&poset));
      Ideal((~down[min])& poset,res, n-NumElement(down[min]&poset));
   }
}
//--------------------------------------------------------------------------------------------
int main(int argc, char* argv[])
{
  FILE *input;
  int i, left, right, num_zeros;
  unsigned long poset;
  char *fname;
  char *mode;

  if(argc != 2){
     fprintf(stderr, "Usage: gendIdeal in_file_name\n");
     exit(1);
  }

  printf("Ideals in Gray Code Order\n");

  fname = argv[1];    //printf("fname\n").
  if((input = fopen(fname, "r")) == NULL){
    fprintf(stderr, "Error: could not read file %s\n", fname);
    exit(1);
  }

  // read number of elements in the set
  fscanf(input, "%d" ,&N);
  printf("\n The number of elements in the poset is . %d\n\n",N);

  for(i=1; i<=N ; i++){
    table[i] = 0;
    table[i] = setbit(table[i], i, 1); //parents[i] = table[i];
  }

  // store partial orderings
  int comma = 0;
  printf(" The poset is \n { ");
  do{
    fscanf(input, "%d %d", &left, &right);
    table[left] = setbit(table[left], right, 1);
    if(left)
      if(comma) printf(" , %d<%d", left, right);
      else      printf("%d<%d", left, right);
    comma = 1;
  }while(left != 0);
  printf(" }\n");
  fclose(input);
  printf("\n\nIdeals in Gray Code Order\n");

  // initialize ideal and poset
  poset = 0;
  for(i=1; i <= N; i++)
    poset = setbit(poset, i, 1);

  Process();
 long  ratio;
 clock_t time1, time2;
 ratio = 1000/CLK_TCK;
 time1 = clock();
//----------------------------------------------------------------------
Ideal(poset, 0, N);

printf("%d Ideals in total\n", counter);

time2 = clock();
printf(" %ld\n",ratio*(long) time2 - ratio*(long) time1);
return 0;
}
```

A.4 Data entering program

```
//------------------------------------------------------------------//
//   This program is written in Dev-C++, version 4.9.9.2           //
//------------------------------------------------------------------//

#include <stdio.h>
#include <stdlib.h>

int main()
{
   FILE *fp;
   int i , a, b;

   if((fp=fopen("test.txt", "w"))==NULL)
   {
     printf("Impossible to open the file testfile\n");
     exit(1);
   }

      printf("Enter the number N of integers in the poset \n");
      scanf("%d",&i);
      fprintf(fp,"%d ",i);
   do{
      printf("Enter a < b or (0 0) to stop\n");
      scanf("%d %d",&a, &b);
      fprintf(fp,"%d %d ",a, b);
   }while(a!=0);
   fclose(fp);
 // open the file to read
   if((fp=fopen("test.txt", "r"))==NULL)
   {
     printf("Impossible to open the file testfile\n");
     exit(1);
   }
   while(!feof(fp))
   {
     fscanf(fp,"%d",&i);
   }
   fclose(fp);
      return 0 ;
}
```

BIBLIOGRAPHY

Abdo, M. 2009. "Efficient Generation of the Ideals of a Poset in Gray Code Order", *Information Processing Letters*, 109(13)687–689.

Amalraj, D.J., Sundararajan, N. and Dhar, G. 1990. "A data structure based on Gray code encoding for graphics and image processing", *Proceedings of the SPIE: International Society for Optical Engineering*, 65-76.

Ball, M.O. and Provan, J.S. 1983. "Calculating bounds on reachability and connectedness in stochastic networks", *Networks*, 13:253-278.

Bitner, J.R., Ehrlich, G. and Reingold, E.M. 1976. "Efficient generation of the binary reflected Gray code and its applications", *Communications of the ACM*, 19(9):517-521.

Buck, M. and Wiedemann, D. 1984. "Gray codes with restricted density", *Discrete Mathematics*, 48:163-171.

Chang, C.C., Chen, H.Y. and Chen, C.Y. 1992. "Symbolic Gray codes as a data allocation scheme for two disc systems", *Computer Journal*, 35(3):299-305.

Chase, P.J. 1970. "Algorithm 382: Combinations of M out of N objects", *Communications of the ACM*, 13(6):368.

——— P.J. 1989. "Combination generation and graylex ordering", *Proceedings of the 18th Manitoba Conference on Numerical Mathematics and Computing, Winnipeg*, Congressus Numerantium 69:215-242.

Chen, M. and Shin, K.G. 1990. "Subcube allocation and task migration in hypercube machines", *IEEE Transactions on Computers*, 39(9):1146-1155.

Chow, S. and Ruskey, F. 2009. "Gray Codes for Polyominoes and a New Class of Distributive lattices", *Discrete Mathematics*, 309:5284-5297.

Cori, R. 75. "Un code pour les cartes planaires et ses applications", *Astérisque*, Paris, 1975.

Diaconis, P. and Holmes, S. 1994. "Gray codes for randomization procedures", *Statistics and Computing*, 4:287-302.

Eades, P., Hickey, B. and Read, R.C., 1984. "Some Hamilton paths and a minimal change algorithm", *Journal of the ACM*, 31(1):19-29.

Eades, P. and McKay, B. 1984. "An algorithm for generating subsets of fixed size with a strong minimal change property", *Information Processing Letters*, 19:131-133.

Ehrlich, G. 1973. "Loopless algorithms for generating permutations, combinations, and other combinatorial configurations", *Journal of the ACM*, 20(1):500-513.

Faloutsos, C. 1988. "Gray codes for partial match and range queries", *IEEE Transactons on Software Engineering*, 14(10):1381-1393.

Gardner, Martin 1972. "Curious properties of the Gray code and how it can be used to solve puzzles", *Scientific American*, 227(2):106-109.

Gilbert, E.N. 1958. "Gray codes and paths on the n-cube", *Bell Systems Technical Journal*, 37:815-826.

Gray, F. March 1953. "Pulse code communications", *U.S. Patent*, 2632058.

Heath, F.G. 1972. "Origins of the binary code", *Scientific American*, 227(2):76-83.

Johnson, S.M. 1963. "Generation of permutations by adjacent transpositions", *Math. Comp.* , 17:282-285.

Joichi, J.T., White Dennis, E. and Williamson, S.G. 1980. "Combinatorial Gray codes", *SIAM Journal on Computing*, 9(1):130-141.

Knuth, D.E. and Ruskey, F. 2003. "Efficient Coroutine Generation of Constrained Gray Sequences". From Object-Orientation to Formal Methods: Dedicated to The Memory of Ole-Johan Dah, *Lecture Notes in Computer Science, Springer-Verlag*, 2635:183-208.

Koda, Y. and Ruskey, F. 1993. "A Gray Code for the ideals of a forest poset", *J. Algorithms*, 15:324-340.

Lawler, E.L. 1979. "Efficient implementation of dynamic programming algorithms for sequencing problems", *Stichting Mathematisch Centrum*, Technical Report BW106/79.

Lehmer, D.H. 1964. "The machine tools of combinatorics, in Applied Combinatorial Mathematics", *E. Beckenbach, ed., John Wiley & Sons, New York*, 5-31.

Lehmer, D.H. 1965. "Permutation by adjacent interchanges", *American Mathematical Monthly*, 72:36-36.

Liu, C.N. and Tang, D.T. 1973. "Algorithm 452, Enumerating M out of N objects", *Comm. ACM*, 16:485.

Losee, R.M. 1992. "A Gray code based ordering for documents on shelves: Classification for browsing and retrieval", *Journal of the American Society for Information Science*, 43(4):312-322.

Lucas, J. 1987. "The rotation graph of binary trees is hamiltonian", *Journal of Algorithms*, 8:503-585.

Lucas, J.M., Roelants van Baronaigien, D. and Ruskey, F.1993. "On rotations and the generation of binary trees", *Journal of Algorithms*, 15(3):343-366.

Ludman, J.E. 1981. "Gray code generation for MPSK signals", *IEEE Transactions on Communications*, COM-29:1519-1522.

Nijenhuis, A. and Wilf, H.S. 1978. *Combinatorial Algorithms for Computers and Calculators*. Academic Press.

Proskurowski, A. and Ruskey, F. 1990. "Generating binary trees by transpositions", *J. Algorithms*, 11:68-84.

Pruesse, G. and Ruskey, F. 1991. "Generating the linear extensions of certain posets by adjacent transpositions", *SIAM Journal of Discrete Mathematics*, 4:413-422.

—— 1993. "Gray Codes from Antimatroids", *Order*, 10:239-252.

Richard, D. 1986. "Data compression and Gray-code sorting", *Information Processing Letters*, 22:210-215.

Robinson, J. and Cohn, M. 1981."Counting sequences", *IEEE Transactions on Computers*, C-30:17-23.

Ruskey, F. 1988. "Adjacent interchange generation of combinations", *Journal of Algorithms*, 9:162-180.

—— 1992. "Generating linear extensions of posets by transpositions", *Journal of Combinatorial Theory Series B*, 54:77-101.

—— 2003. "Combinatorial Generation", *http://www.1stworks.com/ref/RuskeyCombGen.pdf*, iii.

Savage, C.D. 1989. "Gray code sequences of partitions", *Journal of Algorithms*, 10(4):577-595.

—— 1997. "A survey of combinatorial Gray codes", *SIAM*, 39(4):605-629.

Schrage, L. and Baker, K.R. 1987. "Dynamic programming solution of sequencing problems with precedence constraints", *Operations Research*, 26(3):444-449.

Squire, Matthew. "Enumerating the Ideals of a Poset", available electronically at: *http://citeseer.ist.psu.edu/465417.html*.

Stachowiak, G. 1992. "Hamilton paths in graphs of linear extensions for unions of posets", *SIAM Journal on Discrete Mathematics*, 5:199-206.

Steiner, G. 1986. "An algorithm to generate the ideals of a partial order", *Operations*

Research Letters, 5(6):317-320.

Trotter, H.F. 1962. "PERM (Algorithm 115)", *Communications of the ACM*, 5(8):434 - 435.

Walsh, T.R. 1995. "A simple sequencing and ranking method that works on almost all Gray codes", *Department of Mathematics and Computer Science, Université du Québec à Montréal*, Research report 243, 53 pages.

——— 1998. "Generation of Well-Formed Parenthesis Strings in Constant Worst-Case Time", *Journal of Algorithms*, 29:165-173.

——— 2000. "Loop-free sequencing of bounded integer compositions", *The Journal of Combinatorial Mathematics and Combinatorial Computing*, 33:323-345.

——— 2001. "Gray Codes for involutions", *The Journal of Combinatorial Mathematics and Combinatorial Computing*, 36:95-118.

Wells, M.B. 1961. "Generation of permutations by transposition", *Mathematics of Computation*, 15:192-195.

West, D.B. 1993. "Generating linear extensions by adjacent transpositions", *Journal of Combinatorial Theory Series B*, 57:58-64.