Order **12**: 57–75, 1995.
© 1995 *Kluwer Academic Publishers. Printed in the Netherlands.*

A Loop-Free Algorithm for Generating the Linear Extensions of a Poset

E. RODNEY CANFIELD[1]
Department of Computer Science, University of Georgia, Athens, GA 30602, U.S.A.

and

S. GILL WILLIAMSON
Department of Computer Science, University of California, San Diego, La Jolla, CA 92093, U.S.A.

Communicated by I. Rival

(Received: 1 July 1993; accepted: 25 October 1994)

Abstract. A precise concept of when a combinatorial counting problem is "hard" was first introduced by Valiant (1979) when he defined the notion of a $\#P$-complete problem. Correspondingly, there has been consistent interest in the notion of when a combinatorial listing problem admits a very special regular structure in which transition times between objects being listed are uniformly bounded by a fixed constant. Early descriptions of such *loop-free* listing algorithms may be found in the book *Algorithmic Combinatorics* by Even (1973). Recently, the problem of counting all linear extensions of a partially ordered set has received attention with regard to both of these combinatorial concepts. Brightwell and Winkler (1991) have shown, by a very ingenious argument, that the poset-extension counting problem is $\#P$-complete. Pruesse and Ruskey (1992) have shown that the corresponding listing problem can be solved in constant amortized time and have posed the problem of finding a loop-free algorithm for the poset-extension problem. The present paper presents a solution to this latter problem. This sequence of results represents an interesting juxtaposition, in a fixed, naturally-occurring combinatorial problem, of intricate and precisely defined "irregularities" with respect to counting with very strong regularities with respect to listing.

Mathematics Subject Classifications (1991). 06A07, 68Q15, 68P05, 05A05.

Key words. Loop-free algorithms, linear extensions, $\#P$-complete.

1. Introduction

The subject of this paper is linear extensions of a partially ordered set (poset). (All formal definitions concerning posets are collected at the beginning of Section 3.) Our paper involves two themes: $\#P$-completeness and loop-free algorithms. We begin with brief reviews of these.

Consider a two-variables predicate $B(x, y)$; a good example to keep in mind is the assertion that labeled graph x contains hamiltonian cycle y. Associated with the

[1] Research supported by the National Security Agency.

predicate we have a language, X_B, and a counting function, f_B:

$$X_B = \{x\colon\ \exists y B(x,y)\}$$

$$f_B(x) = \#\{y\colon\ B(x,y)\}.$$

The language X_B consists of those elements x for which $f_B(x)$ is positive. As B ranges over all predicates which are decidable by an algorithm whose running time is polynomial in the size of x, the corresponding languages X_B constitute the class NP, those languages recognizable in polynomial time by a nondeterministic Turing machine [1]. The corresponding counting functions f_B constitute the class #P, first defined and studied by Valiant [13].

One language X_1 is said to be easier than another, X_2, written $X_1 \leqslant X_2$, when there is a polynomial time algorithm for recognizing X_1 given an X_2 recognizer as a subroutine, at unit cost per invocation. Those languages which dominate the entire class NP under this relation are called NP-complete. In a similar way Valiant defined when one counting problem is harder than another, thus identifying the class #P-complete. He showed that computing the permanent of a matrix is #P-complete; this corresponds to the predicate $B(x,y)$ which asserts that matching y is contained in bipartite graph x. Through the years proving membership of various counting problems in the class #P-complete has challenged theoreticians; an impressive recent result is that of Brightwell and Winkler [2] dealing with linear extensions of a poset. (In the latter problem, predicate $B(x,y)$ asserts that poset x is a subrelation of linear order y.)

Another well studied topic in combinatorial algorithms is efficient listing of combinatorial objects. A famous early example [8, 12] concerns the permutations of a set. The problem under consideration in this paper is a generalization of the latter. In the Trotter–Johnson algorithm each permutation is obtained from its predecessor by an adjacent transposition. Such listing are referred to as Gray codes, the requirement being that one object differs very little from the predecessor. The original Gray code [5 and 15, p. 1] involves the subsets of a finite set, and appears as an example in Section 2, below.

The general problem of passing systematically once through every member of a combinatorial family, each movement being limited to some specified elementary change, has in its specific instances given rise to many challenging problems [14, 15]. The existence of such Gray codes can be formulated as the existence of a hamiltonian path in a suitably defined graph.

Once we have shown that a Gray code exists for a combinatorial family, the natural next step is to seek an efficient algorithm for listing the objects in the order of the Gray code. By definition, traversing from one object to the next in a Gray code involves changing a small amount of information. But the problem of deciding by an algorithm which information to change may not be easy; thus a number of investigators have studied *loop-free* algorithms [3, 4] for listing a family. In a loop-free algorithm, we choose a representation of the combinatorial objects, and the algorithm is required to convert one representation to the next in worst-case constant time. A general framework for developing loop-free algorithms for families of combinatorial objects satisfying a certain recursive structure, do not include any $\#P$-complete counting problems. Numerous examples of loop-free algorithms have been studied since the work of Ehrlich [3], a recent one related to poset structure is that of Koda and Ruskey [10] which gives a loop-free listing of the ideals of a forest poset.

Pruesse and Ruskey [11] have studied the problem of listing all linear extensions of a given poset. In the case of a relationless poset, this reduces to the problem of generating permutations. The relevant notion of closeness is the adjacent transposition. The underlying graph of linear orders related by adjacent transposition is bipartite; if the two parts have different sizes then of course no Gray code is possible. Pruesse and Ruskey consider instead *signed* linear extensions. (We remind the reader that formal definitions are postponed until Section 3.) They show that the graph whose vertices are the signed linear extensions of a poset P, and whose edges correspond to either sign changes or adjacent transpositions, contains a hamiltonian cycle, always. Using this they give a constant amortized time algorithm for listing the linear extensions of a given poset. They raise the question of whether a loop-free listing algorithm can be found.

The purpose of the present paper is to answer the latter question in the affirmative. In Section 2 we identify a general class of listing algorithms, and prove (Theorem 1) that all members of the class run in constant amortized time. Readers familiar with the Pruesse–Ruskey algorithm (hereafter, the PR algorithm) should think about why it belongs to this class. We next show (Theorem 2) that all algorithms in the general class can be modified to be loop-free. The extent to which classical results do or do not fall into this framework will be left for future study, but it appears that most of the examples of [7] are of this type. Sections 3 and 4 are devoted to proving that the PR algorithm belongs to this class, and Section 5 is a brief conclusion.

2. A General Framework, and a Data Structure

Let us begin with an elementary problem in data structures. The reader already familiar with the PR algorithm may see its relevance. Suppose as part of a larger algorithm we are maintaining an array $v[i]$, $1 \leqslant i \leqslant k$, of Boolean variables. We repeatedly require the following sequence of steps. First, find the least i such that $v[i]$ is **true**, terminating the algorithm if there is no such i. Next, execute a sub-procedure, SP, which has as a parameter the value of i found in the first step. Finally, update the v-array, setting $v[j]$ to **true** for $j < i$, and setting $v[i]$ to either **true** or **false**, the exact new value of $v[i]$ depending on the outcome of the sub-procedure SP. No $v[j]$ for $j > i$ changes.

Let us restate the above formally, and call it GLP, for *general listing process*.

Algorithm *GLP*.

```
1. Initialize all $v[i]$ to **true** and all other variables appropriately.
comment The specifics of Step 1 depend on the particular application of $GLP$.
2. If $v[i]$ is **false** for $1 \leqslant i \leqslant k$, then stop.
3. Otherwise, let $i$ be $\min\{j\colon\ v[j] = \textbf{true}\}$.
4. Execute $SP(i)$.
5. Reset $v[i]$ to **true** or **false**, depending on the outcome of the sub-procedure executed in Step 4.
6. Set $v[j]$ to **true** for $1 \leqslant j < i$, and go back to Step 2.   □
```

For the sake of illustration we give a particular instance. Initialize the v-array to all **true**, and initialize a second array of bits, $G[i]$, of the same size, to all zeroes. Now carry out the algorithm *GLP*, with the sub-procedure *SP*(i) being "flip the ith bit of the G-array." The updating of $v[i]$ in Step 5 is simply "set $v[i]$ to **false**." With a little experimentation the reader will prove that the sequence of values assumed by the v-array is, in essence, that of a binary counter, and the sequence of bits in the G-array is the famous binary Gray code [16, p. 243]. This concludes our illustration.

Returning to the general situation, observe that if $v[k]$ is ever set to **false** in Step 5, then it remains **false** thereafter. Moreover, as soon as $v[k]$ becomes **false** then $v[k-1]$ inherits this property of remaining always **false** if ever set to **false**. Thus we see that the process under consideration has a chance to terminate. But if the nature of the unspecified sub-procedure *SP* is such that one always sets $v[i]$ to **true** in Step 5, then the process runs endlessly. Despite all the unspecified aspects of *GLP*, we have the following theorem to the effect that any instance of the above general process will, ignoring the sub-procedure, run in constant amortized time.

THEOREM 1. *Suppose that algorithm GLP is executed for N iterations. The total amount of work spent in scanning and changing the v-array* (*Steps* 2, 3, 5, *and* 6) *is* $O(N)$.

Proof. Call a finite string of integers $x = (i_1\, i_2\, \dots\, i_N)$ *feasible* if it can arise as the sequence of integers i chosen at Step 3 of some *GLP* executed for N iterations. We consider the empty string to be feasible. Define $|x|$ to be N, the length of string x, and, for $|x| > 0$, define avg(x) by

$$\operatorname{avg}(x) = \frac{1}{|x|}\sum_{t=1}^{|x|} i_t.$$

If suffices to show that $\operatorname{avg}(x) \leqslant 4$ for all feasible and nonempty x.

When $x = (i_1\, i_2\, \dots\, i_N)$ is feasible, of course $i_t \leqslant k$ for all t; define α_j to be the set of feasible strings x for which in fact $i_t \leqslant j$ for all t. Also, define β_j to be those strings $x \in \alpha_j$ which can possibly (with suitable choices at Step 5 of *GLP*) leave all of $v[1], v[2], \dots, v[j]$ equal to **false**. Define constants c_j, $j \geqslant 1$, by

$$c_j = \sum_{t=1}^{j} \frac{t}{2^{t-1}}.$$

We shall show by mathematical induction on j that

$$\operatorname{avg}(x) \leqslant c_j, \quad x \in \alpha_j \tag{2.1}$$

and

$$|x| \geqslant 2^j - 1, \quad x \in \beta_j. \tag{2.2}$$

Because $c_j \to 4$ as $j \to \infty$, this will prove the Theorem.

Strings in α_1 contain only 1's, and so the base case of the induction is established. For the inductive step, we shall use the notation of regular expressions [6, p. 28]. In this notation, we have the following relationships among sets:

$$\alpha_{j+1} = \big(\beta_j(j+1)\big)^*\alpha_j \tag{2.3}$$

$$\beta_{j+1} = \alpha_{j+1}\beta_j(j+1)\beta_j. \tag{2.4}$$

For readers unfamiliar with regular expressions, we explain that (2.3) asserts that α_{j+1} consists precisely of those strings made by concatenating zero or more strings taken from the set $(\beta_j j+1)$, followed by a string from the set α_j. The set $(\beta_j j+1)$ itself is all concatenations of strings from β_j with the single element $j+1$. The reason for equality (2.3) is that in order for an integer $j+1$ to arise at Step 3 in *GLP* each of $v[1], v[2], \ldots, v[j]$ must have been set previously to **false**. The second equation (2.4) states that set β_{j+1} consists precisely of those strings which can be factored into the concatenation of four strings, the first belonging to α_{j+1}, the second to β_j, the third being the singleton $j+1$, and the fourth belonging to β_j. The reason for equality (2.4) is that in order to leave all of $v[1], v[2], \ldots, v[j+1]$ **false**, we must first access $v[j+1]$. This requires that all previous $v[t]$ be set to **false**; but then the access itself causes the earlier $v[t]$ to be reset to **true**, in Step 6, and so they must be set to **false** again.

We can now proceed with the induction. Let $x \in \beta_j$; for the concatenation $x\ j+1$ we find

$$\operatorname{avg}(x\,j+1) = \frac{|x|}{|x|+1}\operatorname{avg}(x) + \frac{1}{|x|+1}(j+1).$$

By induction, $|x| \geqslant 2^j - 1$, and so

$$\operatorname{avg}(x\ j+1) < c_j + \frac{j+1}{2^j} = c_{j+1}.$$

The avg operator obeys the simple concatenation law

$$\operatorname{avg}(xy) \leqslant \max\big(\operatorname{avg}(x), \operatorname{avg}(y)\big),$$

and so from (2.3) we conclude that $\operatorname{avg}(x) \leqslant c_{j+1}$ for all $x \in \alpha_{j+1}$. It is also readily seen from (2.4) and the induction hypothesis (2.2) that $|x| \geqslant (2^j - 1) + 1 + (2^j - 1) = 2^{j+1} - 1$ for $x \in \beta_{j+1}$, and so the proof is complete. □

Consequently, our toy example above provides a way to generate the binary Gray code in constant amortized time. It is our desire to generate the Gray code, and other combinatorial sequences governed by the *GLP* mechanism, in a *loop-free* manner: we want worst case constant time, not just amortized constant time. This can be accomplished by choosing an alternative data structure for the information of the v-array, one which makes Steps 2, 3, 5, and 6 of *GLP* easy to perform. We now describe such a data structure; replace the v-array by an array $p[i]$ of integers. The correspondence between $v[i]$ and $p[i]$ is this: For each i such that $v[i]$ is **true**, $p[i]$ is set to 0. For each i such that $v[i]$ is **false** *and* such that either i is 1 or $v[i-1]$ is **true**, set $p[i]$ equal to the smallest $j > i$ such that $v[j]$ is **true**. In the latter case, if there is no such j, then set $p[i]$ to ∞, a special value. Finally, set all other $p[i]$ to 0. For example, if

$$v[1\ldots 10] = \textbf{false}, \textbf{false}, \textbf{true}, \textbf{true}, \textbf{false}, \textbf{false}, \textbf{false}, \textbf{true}, \textbf{false}, \textbf{false},$$

then we should have

$$p[1\ldots 10] = 3, 0, 0, 0, 8, 0, 0, 0, \infty, 0.$$

THEOREM 2. *Let the array* $v[i]$, $1 \leqslant i \leqslant k$, *of Boolean variables be represented by an array* $p[i]$ *of integers, as described in the preceding paragraph. Then the operations needed for Steps* 2, 3, 5, *and* 6 *of algorithm GLP, namely*: (1) *let* i *equal* $\min\{j\colon v[j] = \textbf{true}\}$, *if it exists*; (2) *modify* $v[i]$; (3) *set* $v[j]$ *to* **true** *for* $j < i$; *can be carried out in constant time.*

Proof. In Steps 2 and 3, to find the smallest i, if there is one such that $v[i]$ is **true**, we consult $p[1]$. If the latter is ∞, then there is no such i, and we may terminate the process; if $p[1] = 0$, then $i = 1$; otherwise, $i = p[1]$. In Step 5 we need to assign a new value to $v[i]$. If it is desired to set $v[i]$ to **true**, then set $p[i]$ to 0. On the other hand, if it is desired to set $v[i]$ to **false**, then one first examines $p[i+1]$: if there is no $p[i+1]$, (that is, if $i = k$), then set $p[i]$ to ∞; if $p[i+1]$ is 0, then set $p[i]$ to $i+1$; if $p[i+1]$ is positive, then set $p[i]$ to $p[i+1]$, and set $p[i+1]$ to 0. Finally, in Step 6, to set all $v[j]$ to **true** for $j < i$, simply set $p[1]$ to 0. □

As a consequence, we have a loop-free algorithm for generating the binary Gray code. Further, any algorithm which can be put into the general framework of *GLP*, and in which the unspecified sub-procedure $SP(i)$ is loop-free, can be converted by Theorem 2 to a loop-free algorithm. In Section 4 we shall see that the Pruesse–Ruskey algorithm for generating all linear extensions of a partially ordered set can be so recast.

We close this section with an observation and a conjecture. In the v-presentation, one may tell in constant time the value of $v[i]$, given i, but a loop is required for the operations of interest in the process *GLP*. With the p-representation, the situation is reversed: we may carry out the various operations required by process *GLP* in constant time, but a loop of unpredictable duration is needed to answer the query "what is the value of $v[i]$?" (The loop is required when $p[i]$ is 0; in that case we must back up an unpredictable distance in the p-array to verify that position i is not preceded by any nonzero entries, or to find the largest $j < i$ such that $p[j] > 0$.) We conjecture that it is impossible to make all operations, those required by process GLP as well as reporting the value of a particular $v[i]$, simultaneously doable in constant time.

3. The Procedure *SP* for Linear Extensions

In the next section we show that the Pruesse–Ruskey algorithm for generating all linear extensions of a partially ordered set fits the framework of the general process GLP just described. However, first we want to identify the sub-problem which will play the role of SP in Step 4. We begin by stating all definitions we need related to posets and linear extensions.

Let n be an integer, $[n]$ be the set $\{1, 2, \ldots, n\}$, and P be a finite partially ordered set with n elements. A *linear extension* of P is a $1-1$ and onto function L: $[n] \to P$, such that $x \prec y$ in P implies $L^{-1}(x) < L^{-1}(y)$. We can display a linear extension by listing it in one-line notation thus: $L(1)L(2)\ldots L(n)$. A *signed linear extension* is a linear extension along with a formal $+$ or $-$ sign, written in one-line notation for example thus: $-L(1)L(2)\ldots L(n)$. Signed linear extensions may be represented for algorithmic purposes by an ordered pair (S, L), in which S is either $+$ or $-$, and L is a linear extension. Two linear extensions L and L' are said to be related by an *adjacent transposition* if they are identical except for the swapping of two consecutive elements $L(i)$ and $L(i + 1)$. The notion of differing by an adjacent transposition can also be applied, with the obvious meaning, to two signed linear extensions, it being required that both have the same sign. We denote the set of linear extensions of P by $E(P)$, and the set of signed linear extensions of P by $E_{-}^{+}(P)$. We define L_i, for $L \in E(P)$, to be the sequence $L(i), L(i+1), \ldots, L(n)$. This sequence may be interpreted as a subposet, or as a linear extension of a subposet, as required by context. The set $\mathrm{Min}(P)$ denotes the minimal elements of a poset P.

Given a linear extension L of P, we define $E_2(L)$ to be the set of all signed linear extensions of P in which $L(1), L(2)$ appear in that order, and in which $L(3)\ldots L(n)$ appear in that order. In other words, $E_2(L)$ consists of all signed linear extensions of P obtainable from L by prefixing an arbitrary sign, and inserting the two elements $L(1), L(2)$ in new positions compatible with P, leaving the latter two elements in the same relative order as that given by L, and likewise the other $n - 2$ elements. For example, let P be the discrete order on 4 elements, with no relations whatsoever, and $L = x_1x_2x_3x_4$; then $E_2(L)$ consists of twelve signed linear extensions: $+x_1x_2x_3x_4$, $+x_1x_3x_2x_4$, $+x_1x_3x_4x_2$, $+x_3x_1x_2x_4$, $+x_3x_1x_4x_2$, $+x_3x_4x_1x_2$, as well as their six formal negations.

Notice that to specify and L' such that $(S, L') \in E_2(L)$ it is only necessary to give the two integers I_1 and I_2 such that $L'(I_1) = L(1)$ and $L'(I_2) = L(2)$. Thus, the

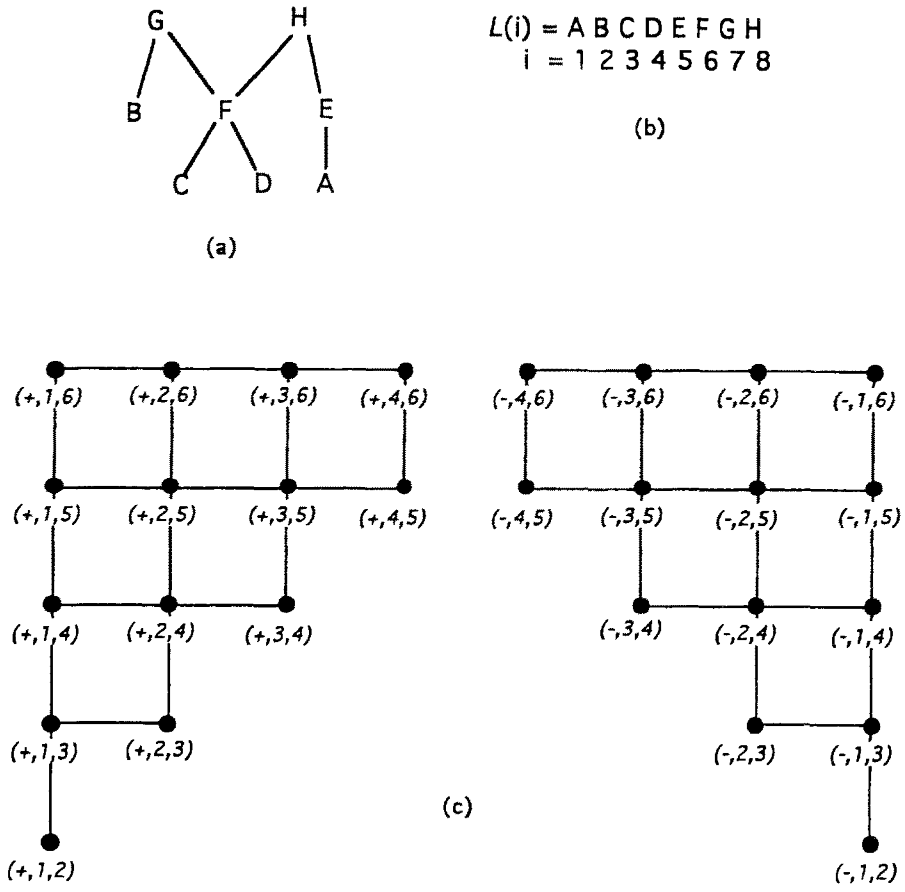

Fig. 1. (a) Hasse diagram of poset P. (b) Linear extension $L = E(P)$. (c) Graph of $E_2(L)$.

set $E_2(L)$ may be conveniently pictured as the points $(\pm, I_1, I_2)$ in Figure 1(c); the edges of that figure indicate the pairs of signed linear extensions which are related by adjacent transposition. Pruesse and Ruskey [11] have given a traversal of the set $E_2(L)$, beginning at $(+, L)$ and ending at $(-, L)$, in which each move consists of either a sign change or an adjacent transposition. Although somewhat tedious to write out in full mathematical precision, the traversal is visually obvious from Figures 2 and 3, which typify the only two possible cases. We shall call this traversal the *canonical path* $P^+E_2(L)$.

The next theorem states that each transition through the canonical path $P^+E_2(L)$ can be determined in constant time. A few words of explanation about terminology used in the proof are in order. We shall use a and b to denote the two elements $L(1), L(2)$ of P. When we speak of switching element a or b with its "left" or "right" neighbor, we mean its immediate predecessor or its immediate successor, respectively. We will use a number of descriptive phrases suggested by Figures 2 and 3. The "express lane" is the rightmost column of the figure, the line we move steadily

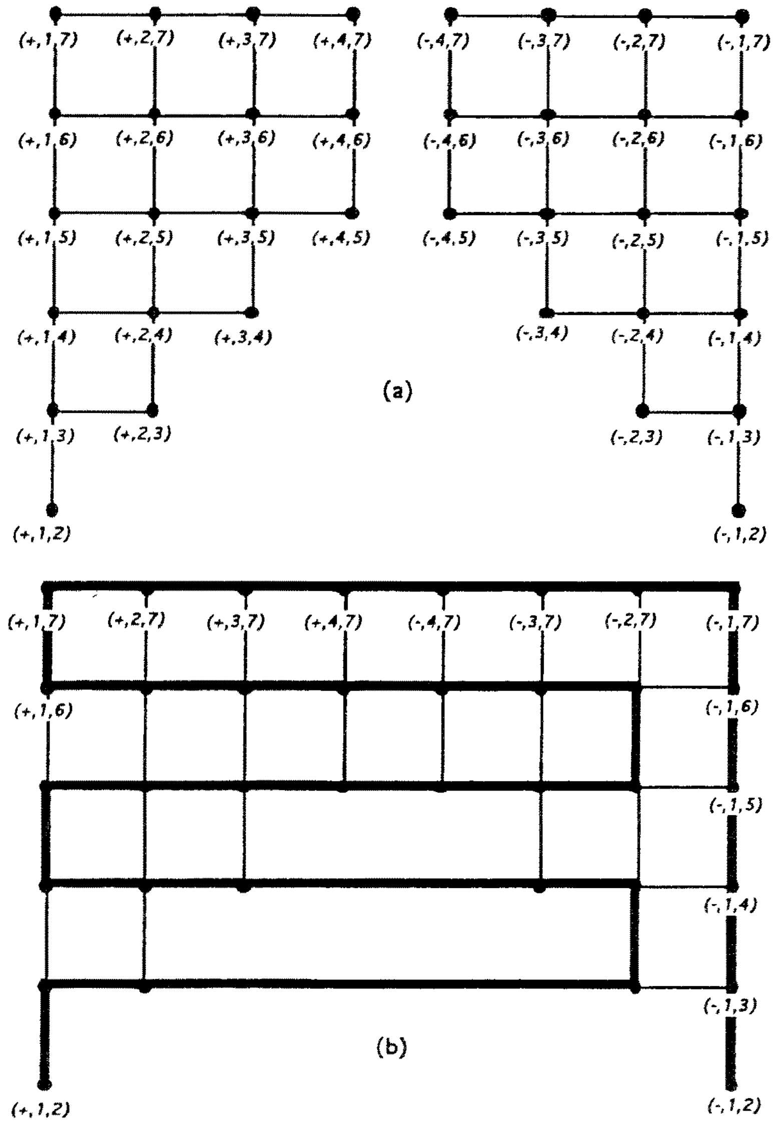

Fig. 2. One case of the canonical path $P^+E_2(L)$.

down during the final stages of the canonical path. Each move down the express lane corresponds to element b changing positions with its left neighbor. When we speak of rightward-moving and leftward-moving rows, it is with reference to Figures 2 and 3. For example, one of the rightward moving rows in Figure 2 consists of the sequence $(+,1,5)$, $(+,2,5)$, $(+,3,5)$, $(+,4,5)$, $(-,4,5)$, $(-,3,5)$, $(-,2,5)$, while an instance of a leftward-moving row is $(-,2,4)$, $(-,3,4)$, $(+,3,4)$, $(+,2,4)$, $(+,1,4)$. All rightward and leftward moves correspond to the element a being switched with one of its neighbors, or a sign change. Upward movement, on the other hand, involves element

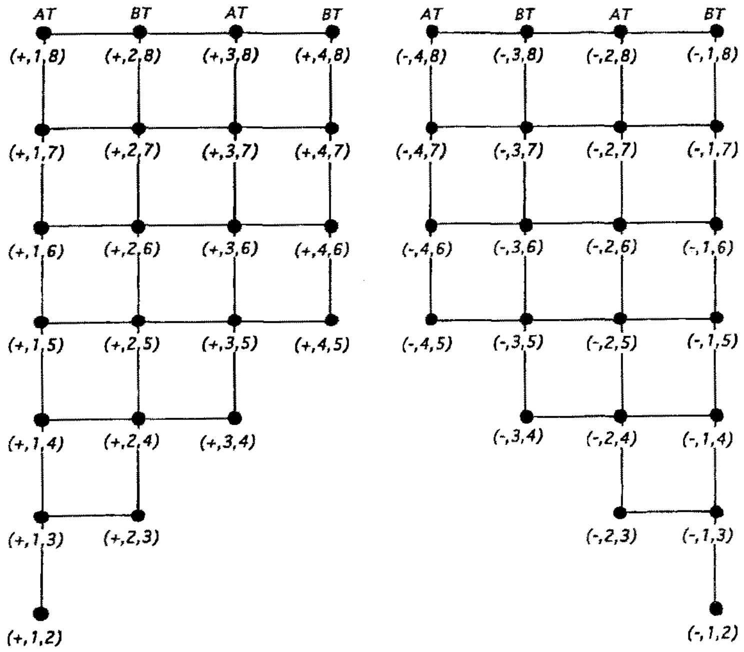

Fig. 3(a). The other case of the canonical path $P^+E_2(L)$.

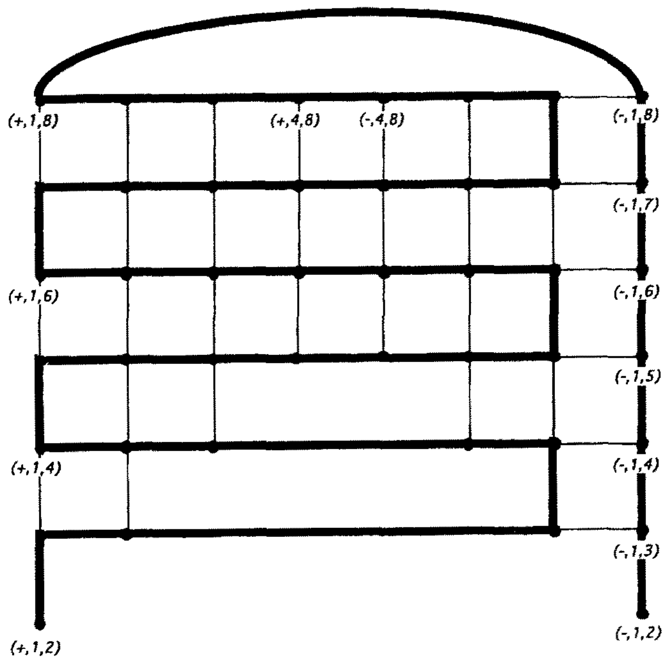

Fig. 3(b). The other case of the canonical path $E_2(L)$.

b switching with its right neighbor. Note that element a can be switched with its right or left neighbor, independently of whether we are on a rightward-moving or leftward-moving row.

THEOREM 3. *Let P be a finite poset, and L be a linear extension of P. Given the following information: a sign S, a linear extension L', and indices I_1, I_2 such that $(S, L') \in E_2(L)$ and $L'(I_1) = L(1)$, $L'(I_2) = L(2)$; then in constant time we may update S, L', I_1, I_2 to represent the next signed linear extension in the canonical path $P^+E_2(L)$.*

Proof. Throughout the proof we use a, b respectively for the elements $L(1), L(2)$. The computation consists of three cases.

Case 1: $I_2 = 2$. In this case a and b are in their leftmost positions, and so we are at either the beginning or the end of the canonical path. If $S = -$, then we are at the end, and no move is possible. Otherwise, $S = +$, and we are at the beginning. We should switch b with its right neighbor, if it is possible. The condition "if it is possible" is tested by comparing element b with its right neighbor according to the partial order. We assume that the condition $x_i \leqslant x_j$ within the given poset P can be tested in fixed time. Testing the possibility of certain moves occurs repeatedly throughout this proof, and we will not explain again that it can be done in constant time. If the switch is not possible, we should make a sign change, setting S to $-$, and bringing us to the end of the canonical path.

Case 2: $I_2 > 2$, $I_1 = 1$, and $S = -$. In this case we are in the express-lane, and element b should be switched with its left neighbor.

Case 3: In all other cases we first determine if we are on a rightward-moving or leftward-moving row, in the geometric sense of Figures 2 and 3. This is decided by seeing if I_2 is odd or even, respectively. Then, by checking S, we decide if element a should attempt a switch with its right or left neighbor. For example, if I_2 is odd and $D = +$, then we are on a rightward-moving row, and element a should consider switching with its right neighbor; if I_2 is odd and $S = -$, then again we are on a rightward-moving row, but now element a should consider switching with its left neighbor. The other two possibilities for (I_2, S), namely $(\text{even}, +)$ and $(\text{even}, -)$, involve leftward-moving rows, and imply a should consider switching with its left and right neighbor, respectively. Having determined in each of the four cases which neighbor a should consider switching with, we test the possibility of such a switch. There are two reasons why it is not enough just to check that the proposed switch preserves a linear extension. In the first place, it is not legal (at this time) for a and b to switch with each other; secondly, on a rightward-moving row, with $S = -$, one must not permit element a to switch with its left neighbor if said switch results in a move into the express lane. If the test reveals that a can make the desired switch then we do so; otherwise, if the desired switch is not possible, then we must choose between making a sign change or trying an upward movement in the sense of Figures 2 and 3.

On a rightward-moving row with $S = +$, an impossible switch of a with its right neighbor can be compensated for by a sign change; this is also true on leftward-moving rows with $S = -$, again when a cannot legally switch with its right neighbor. In the other two cases of impossible switches element a needs to switch with its left neighbor, but element a is already in the leftmost position of the array L'. In any case when a cannot make the desired switch, and it cannot be compensated by a sign change, we consider an upward move. To do so, we check to see if element b can switch with its right neighbor. If b can be switched with its right neighbor, we make the switch; otherwise, we are ready to enter, for the first time, the express lane. This is done either by changing signs (in a leftward moving row), or else by switching element a with its left neighbor (in a rightward moving row). The latter move may have been temporarily rejected during an earlier test. □

4. Realizing the PR Algorithm as a *GLP*

We begin by reviewing the PR algorithm, as given in [11]. In addition to the canonical path $P^+E_2(L)$, we need another path, the *anti-canonical* path $P^-E_2(L)$ which runs from $(-, L)$ to $(+, L)$. The latter proceeds by mimicking the moves of the former, with appropriate reversals of left and right, $+$ and $-$ throughout. Of course, transitions within $P^-E_2(L)$ can be determined in constant time, as done for $P^+E_2(L)$ in Theorem 3. In what follows we shall make use of a function $NextMove$. We have not written out the algorithm for this function explicitly, but a formal specification is implicit in the proof of Theorem 3. The parameters are five in number: $NextMove(\pm, S, L', I, J)$. The first parameter is a sign and indicates whether we are making a move in the canonical path $PE_2^+(L)$ or in the anti-canonical path $PE_2^-(L)$. The next two parameters give our current position (S, L') within the traversal, and the last two play the role of I_1 and I_2 in Theorem 3. For technical reasons, as will be seen later, it is convenient to assume that the last two parameters I and J are not necessarily in order. Hence, to be correct, we say the smaller of I and J plays the role of I_1 in Theorem 3, while the larger plays the role of I_2. The linear order L is only implicitly defined by these parameters; note that in the proof of Theorem 3 no explicit use is ever made of L. The function $NextMove$ returns a result of the form "swap a (or b) with its right (or left) neighbor," or "make a change-sign," or "no move is possible."

Now we are ready to describe a total order on the set of signed linear extensions $E_-^+(P)$; afterwards, we will think about how to move from one signed linear extension to its successor in this total order in constant time. Recall that Min(P) is the set of minimal elements of P. We shall call the total order $TOE_-^+(P)$; its definition is due to Pruesse and Ruskey [11], and is recursive, based on $|P|$. If $|P| = 1$, then the total order is simply $+x$ followed by $-x$, where $P = \{x\}$. Consider now $|P| \geqslant 2$. If P contains a unique minimal element, say Min$(P) = \{c\}$, then $TOE_-^+(P)$ is obtained from $TOE_-^+(P - \{c\})$ by inserting the element c at the beginning of each linear extension of the latter. Otherwise, let a and b be two minimal elements of P,

$\{a, b\} \subseteq \mathrm{Min}(P)$. Recursively determine the order $TOE_{-}^{+}(P - \{a, b\})$. Convert the latter into a sequence of (nonsigned) linear orders $L^1, L^2, \ldots, L^m$ of P by replacing each $(+, L')$ with abL' and each $(-, L')$ by abL'. Observe that each adjacent pair L^i, l^{i+1} so obtained differs by an adjacent transposition. Finally, replace each odd L^i by the canonical path $P^{+}E_2(L^i)$ and each even L^i by the anti-canonical path $P^{-}E_2(L^i)$. We state the theorem that these paths fit together property to produce the desired order.

THEOREM 4 (Pruesse and Ruskey, 1992). *The sequence $TOE_{-}^{+}(P)$ described above includes every signed linear extension of P exactly once; any two consecutive elements of this sequence are related by either sign change or an adjacent transposition, as are the first and last elements of the list.*

Proof. The theorem is proven by induction on $|P|$. The base case, when $|P| = 1$, is clear. The inductive step follows from two observations. If $\mathrm{Min}(P) = \{c\}$, then every linear extension of P consists of c in the first position followed by a linear extension of $P - \{c\}$. If P contains two or more minimal elements, say $\{a, b\} \subseteq \mathrm{Min}(P)$, then every linear extension of P consists of a linear extension of $P - \{a, b\}$, with the elements a, b inserted in such a way as to be compatible with the partial order. □

Imagine we are to generate the order $TOE_{-}^{+}(P)$ according to the above recursive definition. The first step is to find a pair of minimal elements a, b in $\mathrm{Min}(P)$, if there is such, or else the unique minimal element if there is only one. Assume the first case for discussion purpose. Having found $\{a, b\}$, according to the recursive definition, we need the list $TOE_{-}^{+}(P - \{a, b\})$. So, again, we must find either a pair of, or the unique, minimal elements in a certain poset, this time the poset $P - \{a, b\}$. Continuing in this manner, as has been explained by Pruesse and Ruskey [11], before even the first signed linear extension of $TOE_{-}^{+}(P)$ can be written down, we must find an ordered decomposition of P into disjoint subsets,

$$P = S_1 \cup S_2 \cup \cdots \cup S_m, \tag{3.1}$$

in which either

$$|S_i| = 1 \quad \text{and} \quad S_i = \mathrm{Min}(S_i \cup \cdots \cup S_m),$$

or

$$|S_i| = 2 \quad \text{and} \quad S_i \subseteq \mathrm{Min}(S_i \cup \cdots \cup S_m).$$

We are going to represent the decomposition (3.1) by a pair (L^0, J), in which L^0 is a linear extension of P, and $J \subseteq [n]$, $n = |P|$. The relationship between the pair (L^0, J) and the decomposition (3.1) is as follows: L^0 is the linear order obtained by concatenating linear orders on each of the sets S_i, and J consists of those j such that $L^0(j)$ belongs to an S_i of size two as opposed to a singleton S_i. We shall call such a pair (L^0, J) a *preconditioning* for the poset P. Recall that for $L \in E(P)$, L_i is the sequence $L(i), L(i+1), \ldots, L(n)$; using this notation, we may state the conditions that (L^0, J) be a preconditioning for P as follows $(J = \{j_1, j_2, \ldots, j_{2k}\})$:

$$L^0 \in E(P), \tag{3.2}$$

$$j_{2t} = j_{2t-1} + 1, \quad 1 \leqslant t \leqslant k, \tag{3.3}$$

$$L^0(j_t), L^0(j_{t+1}) \in \operatorname{Min}(L^0_{j_t}), \quad t = 1, 3, \ldots, 2k-1, \tag{3.4}$$

$$i \notin J \Longrightarrow \{L^0(i)\} = \operatorname{Min}(L^0_i). \tag{3.5}$$

The reader may verify his understanding of the conditions (3.2)–(3.5) by examining the following algorithm, whose purpose is to find a preconditioning (L^0, J) for P.

Algorithm *PC*

Input: poset P; *Output*: preconditioning (L^0, J).

```
1. Initialize: $J := \emptyset$ and $i := 0$.
2. If $P$ is empty, then stop.
3. Otherwise, let $a$ be a minimal element of $P$.
4. Increment $i$, set $L^0(i)$ to $a$, delete $a$ from $P$.
5. If $a$ was the unique minimal element of $P$, then return to Step 2; otherwise, let $b$ be a second minimal element of $P$.
6. Add $i, i+1$ to the set $J$, increment $i$, set $L^0(i)$ to $b$, delete $b$ from $P$, and return to Step 2. □
```

THEOREM 5. *The algorithm PC determines a preconditioning of P; moreover, the algorithm can be implemented in such a way that the total computation time is $O(n^2)$, n being the size of P.*

Proof. That the algorithm works correctly follows from what has been said already. As for the $O(n^2)$ implementation, this can be done by the usual adjacency list method of carrying out a topological sort [9]. Namely, we compute for each $x \in P$ a count, $C(x)$ of the number of $z < x$. A queue of all x for which $C(x) = 0$ is built. Steps 3, 4, and 5 are executed quickly by taking elements from the queue, if they are available. Deleting an element x from P requires decrementing $C(z)$ for all $z > x$. If such decrementation causes $C(z)$ to become zero, then z is added to the queue. The total amount of incrementing and decrementing to maintain the counts $C(x)$ is exactly, proportional to the number of related pairs in P. □

Now we resume thinking about how the total order $TOE^{\pm}_{-}(P)$ is generated. Given a preconditioning (L^0, J), the first signed linear extension of $TOE^{\pm}_{-}(P)$ is of course $+L^0(1)L^0(2)\ldots L^0(n)$. What is the next? We let (S, L) denote our current position within $TOE^{\pm}_{-}(P)$, so that initially $S = +$ and $L = L^0$. It will be necessary to know at all times the locations within L of the elements $L^0(j)$, $j \in J$, so, for $1 \leqslant t \leqslant 2k$, we define I_t to be the unique index such that $L(I_t) = L^0(j_t)$; or course initially,

$I_t = j_t$ for all t. According to the recursion, we repeatedly make the move computed by $NextMove(+, S, L_{j_1}, I_1, I_2)$, until such time as the returned value is "no move is possible." When no move is possible, the elements $L^0(j_1)$ and $L^0(j_2)$ are in their "home" positions j_1 and j_2, and we compute $NextMove(+, +, L_{j_3}, I_3, I_4)$. After making the latter move, the next move is again an attempt to move $L^0(j_1)$ and $L^0(j_2)$, which incidentally will not result in "no move possible" because we now evoke $NextMove(-, S, L_{j_1}, I_1, I_2)$, the anti-canonical path. Four comments are needed before we state algorithm *NRPR*, the non-recursive PR algorithm. First, it may be necessary to invoke $NextMove(\varepsilon_t, S_t, L_{j_{2t-1}}, I_{2t-1}, I_{2t})$, $t = 1, 2, \ldots$, before finding a pair for which a move is available. (More about ε_t and S_t in a moment.) Second, when the move requested by $NextMove$, is "change sign" and $i = 1$, this means literally to flip the master sign S; but if $i \geqslant 2$, in accordance with the recursively described PR algorithm, a reply of "change sign" means in fact to swap the two elements in positions $I_{2(i-1)-1}$ and $I_{2(i-1)}$. Third, whenever $NextMove(\varepsilon_t, S_t, L_{j_{2t-1}}, I_{2t-1}, I_{2t})$ is invoked, it will be the case that all of the elements $L^0(j_1), L^0(j_2), \ldots L^0(j_{2t-2})$ are in their home positions $j_1, j_2, \ldots, j_{2t-2}$, although in view of the previous comment it may be the case that any individual pair is switched relative to its original order. Fourth and finally, if the number k of incomparable pairs in the decomposition (3.1) is at least 1, then the first time that all k pairs are unable to move we should switch $L(j_{2k-1})$ and $L(j_{2k})$, and terminate the algorithm the second time this happens.

We can now explain all the variables in our algorithm. The current linear extension is L; L^0 is the linear extension; $L^0(j_t)$ and $L^0(j_{t+1})$, $t = 1, 3, \ldots, 2k-1$, play the role of a and b in the proof of Theorem 3; I_t remembers the location of $L^0(j_t)$:

$$L(I_t) = L^0(j_t), \quad 1 \leqslant t \leqslant 2k.$$

The variables S_i, $0 \leqslant i \leqslant k$, store signs: S_0 is the master sign, previously called S, and for $i \geqslant 1$ $S_i = +$ if and only if $L^0(j_{2i-1})$ and $L^0(j_{2i})$ are in the same relative order in which they began the algorithm. The ε_i, $1 \leqslant i \leqslant k$, are also signs, remembering whether a canonical ($\varepsilon_i =$ **true**) or anticanonical ($\varepsilon_i =$ **false**) path is being presently followed by the pair of elements $L^0(j_{2i-1})$ and $L^0(j_{2i})$.

```
Algorithm NRPR
comment [A non-recursive form of the PR algorithm.]
Input: a poset P;
1. Initializations.
1a. Precondition P, using algorithm PC
comment [We assume k, the number of pairs in (3.1), is ⩾ 1.]
1b. $L := L^0$
1c. $I_i := j_i$, $1 \leqslant i \leqslant 2k$
1d. $S_i := +$, $0 \leqslant i \leqslant k$
1e. $\varepsilon_i := +$, $1 \leqslant i \leqslant k$
```

```
2. [Find a move.]
2a. $i := 1$
2b. while $NextMove(\varepsilon_i, S_{i-1}, L_{j_i}, I_{2i-1}, I_{2i})$ = "no move possible" and $i \leqslant k$ do
2c.     flip $\varepsilon_i$
2d.     increment $i$
2e. endwhile
3. [Make the move.]
3a. if $i > k$ then
3b.     if $S_k = -$ then stop
3c.     else exchange $L(I_{2k-1}), L(I_{2k})$, flip $S_k$
3d.     endif
3e. else if $NextMove$ of Step 2 is "change sign" then
3f.             flip $S_{i-1}$, and, if $i > 1$, swap $L(I_{2i-3}), L(I_{2i-2})$,
3g.     else Make the move computed by $NextMove$ in Step 2
3h.     endif
3i. endif
4. Return to Step 2.
```
□

THEOREM 6. *The algorithm NRPR correctly generates the listing $TOE_{\pm}^{+}(P)$ for any poset P.*

Proof. This follows from discussion preceding the statement of the algorithm. □

We need to transform the algorithm one more time, to bring it into the form of a general process *GLP*. The necessary changes are minor. First, we are going to arrange that we never invoke $NextMove$ just to find out if it is possible to move. Rather, we will know in advance when we invoke it that it is possible, and we only carry out the process to find out exactly what is the move, not if the move is possible. We shall maintain an array $v[i]$ of Boolean variables, which tell us for each i, $1 \leqslant i \leqslant k$, whether or not the pair $L(I_{2i-1})$ and $L(I_{2i})$ can be moved. This will be accomplished by the following simple device: each time there is a move of $L(I_{2i-1})$ and $L(I_{2i})$ (other than switching them), we shall check to see if that particular move brought the pair to the end of their circuit. If it did, we reverse ε_i and set $v[i]$ to **false**; if it did not, we leave ε_i unchanged, and we set $v[i]$ to **true**. By introducing an extra Boolean variable $v[k+1]$ we can simplify some of the logic appearing in Step 3 of *NRPR*. This yields the following *GLP* form of the PR algorithm.

```
Algorithm GLPPR
comment A GLP form of the PR algorithm.
Input: a poset $P$
1. [Initializations.]
1a. Precondition $P$, using algorithm $PC$
comment We assume $k$, the number of pairs in (3.1), is $\geqslant 1$.
1b. $L := L^0$
1c. $I_i := j_i$, $1 \leqslant i \leqslant 2k$
```

```
1d. $S_i := +$, $0 \leqslant i \leqslant k$
1e. $\varepsilon_i := +$, $1 \leqslant i \leqslant k$
1f. $v[i] :=$ **true**, $1 \leqslant i \leqslant k+1$
2. **if** $v[i] =$ **false**, $1 \leqslant i \leqslant k+1$, **then** stop
3. **otherwise** $i := \min\{j\colon\ v[j] = \textbf{true}\}$
4. [Make the move.]
4a. **if** $i = k+1$ **then**
4b.     exchange $L(I_{2k-1}), L(I_{2k})$
4c.     $S_k := -$
4d.     $v[k+1] :=$ **false**
4e. **else**
4f.     **case** $NextMove(\varepsilon_i, S_i, L_{j_i}, I_i, I_{i+1})$ **of**
4g.         "change sign": flip $S_{i-1}$, and, if $i > 1$, swap $L(I_{2i-3}), L(I_{2i-4})$
4h.         "swap $a$ (or $b$)...": carry out the indicated swap
4i.     **endcase**
4j. **endif**
5. **if** line 4h above was executed, and the two elements $L(I_{2i-1}), L(I_{2i})$ came to the
   end of their circuit, **then** flip $\varepsilon_i$ and set $v[i]$ to **false**; **otherwise** leave $\varepsilon_i$ unchanged
   and set $v[i]$ to **true**.
6. Set $v[j]$ to **true** for $1 \leqslant j < i$ and return to Step 2.  □
```

This brings us to our final theorem. Recall that for graphs G_1 and G_2 with vertex sets V_1 and V_2 respectively, the *product graph* $G_1 \times G_2$ consists of vertex set $V_1 \times V_2$ and has the following edge relation. Vertex (v_1, w_1) is joined by an edge to (v_2, w_2) if and only if either: v_1 and v_2 are joined in G_1 and $w_1 = w_2$; or, $v_1 = v_2$ and w_1, w_2 are joined in G_2. In this final theorem, we understand the sign set $\{+, -\}$ to have the structure of a single-edge graph, and we understand the set of linear extensions $E(P)$ to have the structure of a graph by joining any two with an edge which differ by an adjacent transposition.

THEOREM 7. *There is a loop-free algorithm for generating all linear extensions $E(P)$ of a given poset P.*

Proof. It is clear from the discussion just before the statement of algorithm $GLPPR$ that the latter both is in the form of a general process, in the sense of Section 2, and generates $TOE^{\pm}_{-}(P)$. By Theorem 3, each execution of Step 4 is carried out in a loop-free manner, and so by Theorem 2, the algorithm $GLPPR$ yields a loop-free traversal of $TOE^{\pm}_{-}(P)$. There remains the question of how to produce each linear extension (as opposed to signed linear extensions) once and only once, still in a loop-free manner. This last issue has been already resolved by Pruesse and Ruskey who point out that the product graph $\{+, -\} \times E(P)$, is bipartite. (Because each factor is bipartite.) Therefore, if we consider the linear extension appearing in every other pair (S_0, L), each extension will be listed once and only once. □

5. Concluding Remarks

We have seen that given a finite partially ordered set P, of size $|P| = n$, in time $O(n^2)$ we can set up data structures which permit us to list efficiently all linear extensions of P. During this listing process, the internal representation of each linear extension, as an ordered array $L(1), L(2), \ldots, L(n)$ of elements of P, will be ordered from its predecessor in constant time, independent of P.

As of this writing, no such loop-free listing algorithm is known for any other fundamental classical combinatorial family whose counting function is in the class $\#P$-complete. It would be of interest to identify one or two other classical combinatorial families, possibly with the aid of Theorem 2, that, like the extensions of a poset, are both in the class $\#P$-complete and admit a loop-free algorithm (one can concoct $\#P$-complete families that admit loop-free listing algorithms by embedding a known $\#P$-complete family in a much larger problem with a trivial loop-free listing and using the many trivial transitions to "buy time" to make the more difficult transitions between the objects of the embedded $\#P$-complete family). It is perhaps not surprising that linear extensions of a poset is the first basic combinatorial family identified with both the $\#P$-complete and loop-free properties, because its counting function has the following notable property. Whereas Valiant's result that the permanent is $\#P$-complete is striking because the corresponding existence problem (finding a matching) is in the class P (by the Ford–Fulkerson algorithm), the theorem of Brightwell and Winkler is all the more so because the corresponding existence problem is trivial: every poset has at least one linear extension.

References

1. Aho, A., Hopcroft, J., and Ullman, J. (1974) *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass.
2. Brightwell, G. and Winkler, P. (1991) Counting linear extensions, *Order* **8**, 225–242.
3. Ehrlich, G. (1973) Loopless algorithms for generating permutations, combinations, and other combinatorial configurations, *J. Assoc. Comput. Mach.* **20**, 500–513.
4. Even, S. (1973) *Algorithmic Combinatorics*, The Macmillan Company, New York.
5. Gray, F. (1953) Pulse code communication, *U.S. Patent 2632058.*
6. Hopcroft, J. E. and Ullman, J. D. (1979) *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, Mass.
7. Joichi, J. T., White, D. E., and Williamson, S. G. (1980) Combinatorial Gray codes, *SIAM J. Computing* **9**, 130–141.
8. Johnson, S. M. (1963) Generation of permutations by adjacent transpositions, *Mathematics of Computation* **17**, 282–285.
9. Knuth, D. (1966) *The Art of Computer Programming*, Vol. II, Addison-Wesley, Palo Alto.
10. Koda, Y. and Ruskey, F. (1993) A Gray code for the ideals of a forest poset, *J. Algorithms* **15**, 324–340.
11. Pruesse, G. and Ruskey, F. (1994) Generating linear extensions fast, *SIAM J. Computing* **23**, 373–386.
12. Trotter, H. (1962) Algorithm 115: Perm., *Communications of the ACM* **5**, 434–435.
13. Valiant, L. (1979) The complexity of computing the permanent, *Theoretical Computer Science* **8**, 189–201.
14. Wilf, H. (1977) A unified setting fjor sequencing, ranking, and selection algorithms for combinatorial objects, *Advances in Math.* **24**, 281–291.

15. Wilf, H. (1989) Combinatorial algorithms: An update, in *CBMS-NSF Regional Conf. Series in Appl. Math.* SIAM, Philadelphia.
16. Williamson, S. G. (1988) *Combinatorics for Computer Science*, Computer Science Press, Rockdale, MD.