GENERATING LINEAR EXTENSIONS FAST

GARA PRUESSE* AND FRANK RUSKEY†

Abstract. One of the most important sets associated with a poset $\mathcal{P}$ is its set of linear extensions, $E(\mathcal{P})$. In this paper, we present an algorithm to generate all of the linear extensions of a poset in constant amortized time; that is, in time $O(e(\mathcal{P}))$, where $e(\mathcal{P}) = |E(\mathcal{P})|$. The fastest previously known algorithm for generating the linear extensions of a poset runs in time $O(n \cdot e(\mathcal{P}))$, where n is the number of elements of the poset. Our algorithm is the first constant amortized time algorithm for generating a "naturally defined" class of combinatorial objects for which the corresponding counting problem is #P-complete. Furthermore, we show that linear extensions can be generated in constant amortized time where each extension differs from its predecessor by one or two adjacent transpositions. The algorithm is practical and can be modified to efficiently count linear extensions, and to compute $P(x < y)$, for all pairs x, y, in time $O(n^2 + e(\mathcal{P}))$.

Key words. Poset, linear extension, transposition, combinatorial Gray code

AMS subject classifications. 05C45, 06A05, 06A06, 68Q25

1. Introduction.

One definition of the adverb "fast" is "in quick succession" (Webster's Collegiate Dictionary [1]). The purpose of this paper is to show that the linear extensions of a partially ordered set (poset) can be generated fast; so fast, in fact, that no algorithm can be faster, up to constant factors. Furthermore, the constants involved are very small and our algorithms extend the practical range of posets for which extensions can be generated and counted.

Linear extensions are of great interest to computer scientists because of their relation to sorting and scheduling problems. For example, there are many NP-complete one-processor scheduling problems with precedence constraints ([13]), and one obvious way of solving such problems is by generating all linear extensions of the precedence constraints and picking the best extension. Linear extensions are also of interest to combinatorists because of their relation to counting problems ([2],[21]). Our results can be used to efficiently generate standard Young Tableau of a given shape, alternating permutations, and any of the many other combinatorial objects that can be viewed as linear extensions of particular posets.

Given a poset $\mathcal{P}$, two questions naturally arise. The *generation question* asks whether the linear extensions, $E(\mathcal{P})$, of $\mathcal{P}$ be efficiently generated. The *counting question* asks whether $e(\mathcal{P})$, the size of the set $E(\mathcal{P})$, can be efficiently determined. The recent result of Brightwell and Winkler [4] that the counting question is #P-complete indicates that the counting question may be no easier than the generation question. We give the best possible answer to the generation question in the sense that our algorithm generates $E(\mathcal{P})$ in time complexity $O(e(\mathcal{P}))$ (aside from a small amount of preprocessing).

We say that a generation algorithm runs in *constant amortized time* if it runs in time $O(N)$, where N is the number of objects generated. In the case of linear extensions we assume that a unit cost oracle is available that takes two poset elements a and b and returns whether $a \prec b$, or not. We also assume that the poset elements

[*] Department of Computer Science, University of Toronto, Toronto, Ontario, M5S 1A4, Canada, gara@cs.utoronto.ca. Research supported in part by NSERC. Research in this paper took place while visiting Simon Fraser University, Burnaby, Canada.

[†] Department of Computer Science, University of Victoria, Victoria, B.C., V8W 2Y2, Canada, fruskey@csr.uvic.ca. Research supported in part by the Natural Sciences and Engineering Research Council of Canada under Grant A3379.

have been labeled in a particular manner, to be described later. This labeling can be carried out in time $O(n^2)$ on an n element poset. Aside from the space used for the poset, the amount of space required by our algorithm is $O(n)$. No constant amortized time generation algorithm was previously known for a class of combinatorial objects for which the corresponding counting problem is #P-complete.

The problem of generating the linear extensions of a poset has been considered by Knuth and Szwarcfiter [12], Varol and Rotem [24], and Kalvin and Varol [11]. In these papers the term "topological sorting" is used instead of "linear extension". The most efficient of these algorithms appears to be that of Varol and Rotem [24], whose time complexity is given as $O(n \cdot e(\mathcal{P}))$ in [11], where n is the number of elements in the poset. It is worth noting that the Varol and Rotem algorithm is very simple and elegant, and is quite efficient in practice. The only algorithm that we are aware of for counting linear extensions of arbitrary posets is that of Wells [25], but it appears to be difficult to analyze. For particular classes of posets, such as series-parallel or bounded width, efficient algorithms for counting are known (see, for example Bouchitte and Habib [3]).

2. Strategy and Definitions. A popular strategy for efficiently generating some set of combinatorial objects is to insist that successive objects in the listing differ by some small and prescribed way. Listings of combinatorial objects that have this property are called (generalized or combinatorial) Gray codes. For example, the binary reflected Gray codes yield a method for generating all the n-bit strings such that each bit string differs from its predecessor by the flipping of one bit. Gray codes have been found for several classes of combinatorial objects; many of these are described in Wilf [26].

We will regard linear extensions as permutations of the elements of the poset. When generating various classes of permutations, the most common "closeness" criteria is that successive permutations differ by a transposition of two of their elements; sometimes this is further restricted to a transposition of adjacent elements only. The well-known algorithm of Steinhaus [22], Johnson [10], and Trotter [23] provides a Gray code listing of all the $n!$ permutations of n elements where each permutation differs from its predecessor by a transposition of two adjacent elements. Thus we say that the $n!$ permutations can be *generated by (adjacent) transpositions.* The permutations of an n-set correspond to the linear extensions of the poset that is an n element antichain.

In general, it is not always possible to generate the linear extensions of a poset by transpositions, adjacent or not; for example, the linear extensions of the poset consisting of two non-trivial chains and only if n and m are both odd ([5], [6], and [16]). Thus, the linear extensions of the poset in Figure 1 (two 2-element chains) cannot be generated by transpositions. The linear extensions of some classes of posets have been shown to be generable by transpositions (see [18],[15], [20]). It is an open problem to characterize the posets which have this property. Even when the linear extensions of a family of posets can be generated by transpositions, a fast algorithm to perform the generation may not exist.

The basic strategy of our initial algorithm is to generate each linear extension twice, where each extension is flagged, plus or minus. The algorithm keeps track of the signs of the extensions and only "outputs" the plus extensions. Thus, in a sense, this algorithm falls into the class of generation algorithms that generate more objects than those that are actually output.

We now introduce our terminology and notation.

A poset (or partially ordered set) $\mathcal{P}$ is a reflexive, transitive, and antisymmetric relation $R(\mathcal{P})$ on a set $S(\mathcal{P})$. An ordered pair $(a,b) \in R(\mathcal{P})$ is denoted $a \preceq_{\mathcal{P}} b$, or, when it will not lead to confusion, simply $a \preceq b$. By $a \prec b$ we mean $a \preceq b$ and $a \neq b$. An element a is *minimal* in $\mathcal{P}$ if there is no element b such that $b \prec a$. Let $\mathit{Min}(\mathcal{P})$ denote the set of minimal elements of $\mathcal{P}$. If $a \prec b$ and there does not exist a c in $S(\mathcal{P})$ such that $a \prec c \prec b$, then we say that b *covers* a. Let $\mathit{Cover}(a)$ denote the set of elements that cover a. Elements a and b are said to be *incomparable* if $a \not\preceq b$ and $b \not\preceq a$. We write $a \| b$ to indicate that a and b are incomparable. If no pair of elements of $S(\mathcal{P})$ are incomparable, then $\mathcal{P}$ is a *total ordering.* If $\mathcal{P}$ is a total ordering on $S(\mathcal{P}) = \{x_1, x_2, \ldots, x_n\}$ such that $x_i \prec x_j$ if and only if $i < j$, then we sometimes use $x_1 x_2 \cdots x_n$ to denote $\mathcal{P}$. An *extension* of $\mathcal{P}$ is a poset $\mathcal{Q}$ such that $S(\mathcal{P}) = S(\mathcal{Q})$, and $R(\mathcal{P}) \subseteq R(\mathcal{Q})$. An extension of $\mathcal{P}$ which is a total ordering is called a *linear extension* of $\mathcal{P}$. Let $E(\mathcal{P})$ denote the set of linear extensions of $\mathcal{P}$, and let $e(\mathcal{P})$ denote $|E(\mathcal{P})|$. We let $\pm E(\mathcal{P})$ denote $\{+l, -l \mid l \in E(\mathcal{P})\}$.

The *height*, $h(x)$, of an element x is the average position that it occupies in a linear extension. Thus, a minimum element has height 1, a maximum element has height $|S(\mathcal{P})|$, and if $\mathcal{P}$ is an antichain, then all elements have height $(|S(\mathcal{P})|+1)/2$. The probability that x precedes y is denoted $P(x < y)$; it is the number of extensions in which x precedes y divided by the total number of extensions. In connection with sorting algorithms it is desirable to find pairs of elements x and y where $P(x < y)$ is close to $1/2$.

For $T \subseteq S(\mathcal{P})$, we let $\mathcal{P} \backslash T$ denote the poset on the set $S(\mathcal{P}) \backslash T$ with the relations set $R(\mathcal{P}) \cap (S(\mathcal{P}) \backslash T)^2$. Suppose a and b are incomparable elements of $S(\mathcal{P})$ such that a has the same relationship to all other elements of $S(\mathcal{P})$ as b; more precisely, suppose that, for all $c \in S(\mathcal{P})$, $c \prec a$ if and only if $c \prec b$, and $a \prec c$ if and only if $b \prec c$. Then a and b are called *siblings.* For posets $\mathcal{P}$ and $\mathcal{Q}$, if $R(\mathcal{P}) \cup R(\mathcal{Q})$ is antisymmetric, then we let $\mathcal{P} + \mathcal{Q}$ denote the poset on the set $S(\mathcal{P}) \cup S(\mathcal{Q})$ with the relation set which is the transitive closure of $R(\mathcal{P}) \cup R(\mathcal{Q})$. For example, $\mathcal{P} + abc$ is the poset on the set $S(\mathcal{P}) \cup \{a, b, c\}$ with the relation set which is the transitive closure of $R(\mathcal{P}) \cup \{(a,b), (b,c)\}$. If $\mathcal{P} + \mathcal{Q} = \mathcal{P}$, then we say $\mathcal{P}$ *induces* $\mathcal{Q}$. For example, if $\mathcal{P} + abc = \mathcal{P}$, then $\{(a,b), (b,c)\} \subseteq R(\mathcal{P})$, and every linear extension of $\mathcal{P}$ has $a \prec b \prec c$; therefore, we say $\mathcal{P}$ induces abc. For element disjoint total orders $\alpha, \beta, \gamma, \delta$, we let $\alpha(\beta + \gamma)\delta$ denote the poset $\alpha\beta\delta + \alpha\gamma\delta$.

Consider the graph which has $E(\mathcal{P})$ as its vertex set, such that two vertices are adjacent in the graph whenever the corresponding linear extensions differ by a single transposition. This graph is called the *transposition graph* of the poset $\mathcal{P}$ and is denoted $G(\mathcal{P})$. The subgraph of $G(\mathcal{P})$ on the same vertex set but containing only the edges which correspond to adjacent transpositions is called the *adjacent transposition graph* and is denoted $G'(\mathcal{P})$. Generating the linear extensions of $\mathcal{P}$ by (adjacent) transpositions is equivalent to finding a Hamiltonian path in the graph $G(\mathcal{P})$ $(G'(\mathcal{P}))$. Figure 1 shows a poset and its transposition graph. If α and β are linear extensions of $\mathcal{P}$, then by $D(\alpha, \beta)$ we denote the distance in $G(\mathcal{P})$ from α to β, and by $D'(\alpha, \beta)$ we denote the corresponding distance in $G'(\mathcal{P})$.

Transposition graphs are bipartite and connected. If the partite sets of $G(\mathcal{P})$ are not the same size, then there is no Hamiltonian cycle through the graph; if the difference in the size of the partite sets is more than one, there is no Hamiltonian path through the graph and thus, the linear extensions of $\mathcal{P}$ cannot be generated by transpositions. Ruskey [17] conjectures that this necessary condition for the existence of a Hamiltonian path is also sufficient, suggesting a possible characterization of the

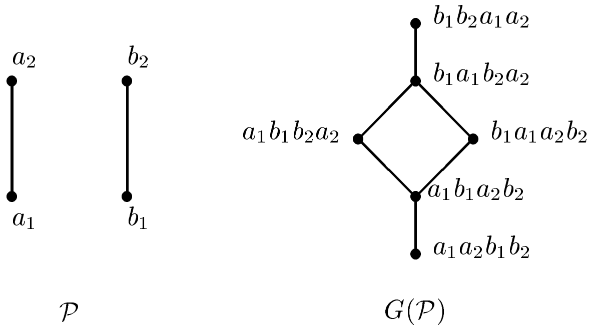

FIG. 1. *A poset and its transposition graph.*

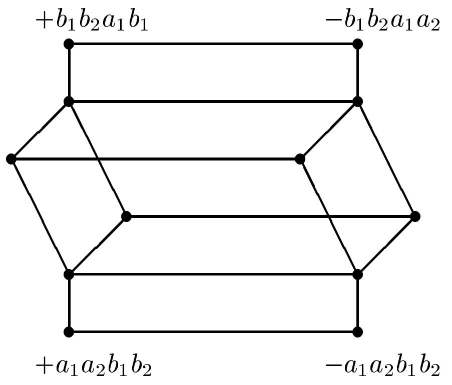

FIG. 2. *The graph $G(\mathcal{P}) \times K_2$.*

posets whose linear extensions can be generated by transpositions. In Figure 1, the partite sets have a size difference of two, so the linear extensions of that poset cannot be generated by transpositions.

If G is a graph, then let $G \times K_2$ be the graph which results from taking two copies of G, and adding the edges which correspond to an isomorphism between the two copies. To differentiate between the copies of G, we will prefix the vertices of one copy of G with "+" and the other with "−". For example, Figure 2 shows $G(\mathcal{P}) \times K_2$, where $\mathcal{P}$ is the poset shown in Figure 1.

A lemma which will be useful in later sections can be stated as follows:

LEMMA 2.1. *If a and b are siblings in $\mathcal{P}$, then $G(\mathcal{P}) \cong G(\mathcal{P} + ab) \times K_2$.*

Proof. Observe that $E(\mathcal{P}) = E(\mathcal{P} + ab) \cup E(\mathcal{P} + ba)$. For any linear extension l of $\mathcal{P}$, transposing a and b in l yields another linear extension of $\mathcal{P}$. Therefore, the operation which transposes a and b in a linear extension provides an isomorphism between $G(\mathcal{P} + ab)$ and $G(\mathcal{P} + ba)$. ◻

If $e(\mathcal{P}) = 1$ (i.e., if $\mathcal{P}$ is a total order), $G(\mathcal{P}) \times K_2$ is an edge. For the purpose of inductively showing the existence of Hamiltonian cycles, we consider this graph to have a Hamiltonian cycle, since it has a Hamiltonian path such that the endpoints are adjacent.

3. The Graph $G'(\mathcal{P}) \times K_2$ is Hamiltonian.

The proof that $G'(\mathcal{P}) \times K_2$ is Hamiltonian forms the basis of the efficient algorithm to be presented in the next section. That this is true for a certain kind of poset, called a B-poset, was shown by

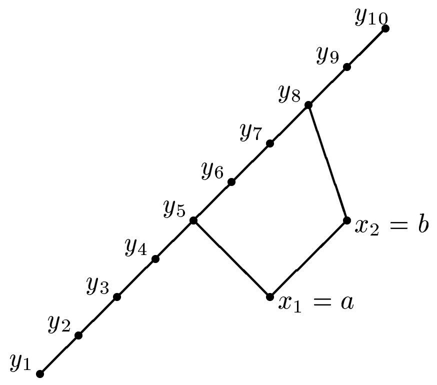

FIG. 3. *A B-poset.*

Pruesse and Ruskey [15], and this result will be used in the proof of the general case.

DEFINITION 3.1. *A* B-poset *is a poset $\mathcal{P}$ whose elements can be partitioned into two disjoint chains, $x_1 \prec x_2 \prec \cdots \prec x_n$ and $y_1 \prec y_2 \prec \cdots \prec y_m$, such that $y_j \not\preceq x_i$ for all i and j, where $1 \leq i \leq n$ and $1 \leq j \leq m$.*

An example of a B-poset is shown in Figure 3. Note that $\kappa = x_1 x_2 \ldots x_n y_1 y_2 \ldots y_m$ is a linear extension of any B-poset. The extension κ is called the *canonical linear extension* of a B-poset. Define $mr(x_i)$ to be the largest index j such that $x_i \parallel y_j$; if $x_i \prec y_1$, then $mr(x_i) = 0$. For the B-poset of Figure 3, $mr(a) = 4$ and $mr(b) = 7$.

The following lemma was proved in [15].

LEMMA 3.2. *Let $\mathcal{P}$ be a B-poset. Then there exists a Hamiltonian cycle in $G'(\mathcal{P}) \times K_2$ which uses the edge $[+\kappa, -\kappa]$.*

Figure 4 shows the graph $G'(\mathcal{P}) \times K_2$, where $\mathcal{P}$ is the B-poset shown in Figure 3. The edges corresponding to the isomorphism between the two copies of $G'(\mathcal{P})$ have been omitted for the sake of clarity. One can think of traveling up a vertical edge as transposing $b = x_2$ with its neighbor on the right, and traveling along a horizontal edge as transposing $a = x_1$ with one of its neighbors. A Hamiltonian path between $+\kappa$ and $-\kappa$ is shown in Figure 5. All B-posets used in the remainder of the paper have $n = 2$; we call these *2B-posets.*

A graph similar to that shown in Figure 4 arises whenever $a \parallel y_1$; we call this the *typical case.* If $a \prec y_1$ then $G'(\mathcal{P})$ is a path; we call this the *atypical case.* In other words, the typical case occurs when $mr(a) > 0$ and the atypical case occurs when $mr(a) = 0$.

THEOREM 3.3. *For every poset $\mathcal{P}$, the graph $G'(\mathcal{P}) \times K_2$ is Hamiltonian.*

Proof. The proof of the theorem is by induction on $|S(\mathcal{P})|$. For the base cases of the induction, $\mathcal{P}$ is the poset on zero or one elements; in both of these cases $G'(\mathcal{P}) \times K_2$ is an edge.

Suppose $|S(\mathcal{P})| > 1$. If $\mathcal{P}$ has a unique minimum a, then $G'(\mathcal{P}) \cong G'(\mathcal{P} \backslash \{a\})$, and by the inductive hypothesis $G'(\mathcal{P} \backslash \{a\}) \times K_2$ is Hamiltonian.

Otherwise, let $\mathcal{P}$ have two minimal elements a and b. By the inductive hypothesis, the graph $G'(\mathcal{P} \backslash \{a, b\}) \times K_2$ has a Hamiltonian cycle H'. Replace each signed linear extension $+\alpha_i$ on H' with $ab\alpha_i$; replace each linear extension $-\alpha_i$ with $ba\alpha_i$. The result is a cycle $\beta_1, \beta_2, \ldots, \beta_M$, where $M = 2 \cdot e(\mathcal{P} \backslash \{a, b\})$, in $G'(\mathcal{P}) \times K_2$, which visits exactly those linear extensions in which a and b precede all other elements of $S(\mathcal{P})$.

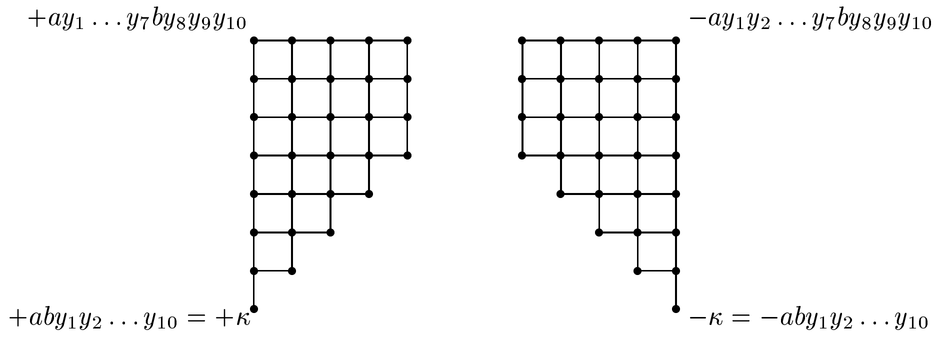
$+ay_1\dots y_7by_8y_9y_{10}$
$-ay_1y_2\dots y_7by_8y_9y_{10}$
$+aby_1y_2\dots y_{10} = +\kappa$
$-\kappa = -aby_1y_2\dots y_{10}$

FIG. 4. *The graph $G'(\mathcal{P}) \times K_2$.*

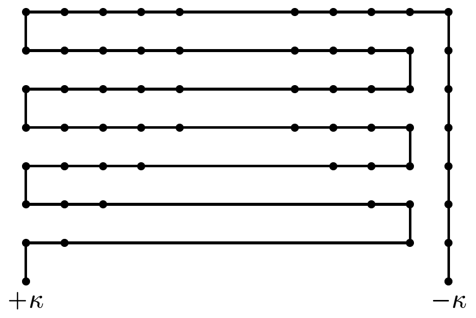

FIG. 5. *A Hamiltonian cycle through $G'(\mathcal{P}) \times K_2$.*

That is, this cycle visits all the linear extensions of

$$\mathcal{P}' = \mathcal{P} + \sum_{x \in S(\mathcal{P})\backslash\{a,b\}} ax + bx.$$

The poset $\mathcal{P}'$ is $\mathcal{P}$ extended so that a and b are covered by every other element of $Min(\mathcal{P})$.

For each $\beta_i = x_i y_i \zeta_i$, where $x_i = a, y_i = b$ or $x_i = b, y_i = a$, the poset $\mathcal{P} + x_i y_i + \zeta_i$ is a B-poset. By Lemma 3.2, there is a Hamiltonian path through $G'(\mathcal{P}+x_iy_i+\zeta_i)\times K_2$ from $+\beta_i$ to $-\beta_i$. We substitute the occurrence of β_i with this path in H', for each odd i. For each even i, we substitute the reverse of this path for the occurrence of β_i. Call the resulting walk H.

To prove that H is a Hamiltonian cycle through $G'(\mathcal{P}) \times K_2$, it is necessary to show that every vertex on the cycle H is a linear extension of $\mathcal{P}$; this is true, since $E(\mathcal{P}') \subseteq E(\mathcal{P})$, and hence each B-poset generated is an extension of $\mathcal{P}$. It is also necessary to show that for each linear extension l of $\mathcal{P}$, $+l$ and $-l$ both occur exactly once on H. Suppose l induces the order xy on $\{a, b\}$ and the order ζ on $S(\mathcal{P})\backslash\{a, b\}$. Then $xy\zeta$ is a linear extension of $\mathcal{P}'$, and l is a linear extension of the B-poset $\mathcal{P} + xy + \zeta$; also, every other B-poset generated either does not induce the order xy or does not induce the order ζ. Therefore, $+l$ and $-l$ are generated only during the generation of $E(\mathcal{P} + xy + \zeta)$; i.e., each $+l$ and $-l$ are generated exactly once. ☐

Observe that the reference to Lemma 3.2 in the preceding proof was not strictly necessary because the B-posets that occur are all 2B-posets. In the "typical" case, the cycle of Figure 5 could be used; in the "atypical" case the cycle is obvious (move b

to the right as far as possible, change signs, and then move the b back to the left). If $mr(b)$ is even, the cycle of Figure 5 is slightly different and uses the edge $[+a\gamma b, -a\gamma b]$, where $\gamma = y_1 y_2 \cdots y_m$. These cycles are used in the algorithm of the following section.

COROLLARY 3.4. *If $\mathcal{P}$ is a poset with a pair of siblings, then $G(\mathcal{P})$ is Hamiltonian.*

Proof. Suppose $\mathcal{P}$ has a pair of siblings a, b. By Theorem 3.3, $G'(\mathcal{P} + ab) \times K_2$ is Hamiltonian; therefore $G(\mathcal{P} + ab) \times K_2$ is Hamiltonian. Hence by Lemma 2.1, $G(\mathcal{P})$ is Hamiltonian. □

It is an open problem to determine whether $G'(\mathcal{P})$ is Hamiltonian, where $\mathcal{P}$ is a poset with a pair of siblings.

4. The Algorithm.

The proof of Theorem 3.3 is constructive. In this section, we present the recursive algorithm implicit in the inductive proof. The algorithm runs in *constant amortized time*; i.e., generating all the linear extensions of a poset $\mathcal{P}$ takes time $O(e(\mathcal{P}))$.

We first give an overview of the algorithm and use a small example to give a general understanding of how it works. We then give the details of the algorithm and a proof of its correctness.

The algorithm is an *in-place* algorithm; it maintains an array `le` in which contains the current linear extension, and maintains a variable `IsPlus` which keeps track of the sign ("+" or "−"). We go from one linear extension to the next by making changes to the array or reversing the sign.

The main procedure used by the algorithm, which we call `GenLE`, is recursive and basically follows the path indicated in Figure 5. Every level of the recursion has an associated pair of minimal elements of the current subposet. For example, in the poset shown in Figure 1, a_1, b_1 are a pair of minimal elements of $\mathcal{P}_1 = \mathcal{P}$, and a_2, b_2 are a pair of minimal elements of $\mathcal{P}_2 = \mathcal{P}_1 \backslash \{a_1, b_1\}$. These pairs are determined by some preprocessing which will be described later.

The procedures `Move` and `Switch` are used to change the current linear extension. They operate in $O(1)$ time as follows:

`Switch(i):` If $i = 0$, then the sign is changed; that is, `IsPlus` is changed. If $i > 0$, then a_i and b_i are transposed.

`Move(x,left):` This call transposes `x` with the element on its left.

`Move(x,right):` This call transposes `x` with the element on its right.

Each time a new linear extension l of $\mathcal{P}_i$ is generated by the call `GenLE(i)` (i.e., each time `Move` or `Switch` is called), `GenLE(i-1)` is called; the call `GenLE(i-1)` moves $a_1, b_1, \ldots, a_{i-1}, b_{i-1}$ in all possible ways through l, while maintaining the order $a_{i-1} \prec b_{i-1}$ (or $b_{i-1} \prec a_{i-1}$, depending on their order at the point of calling `GenLE(i-1)`). If $i = 1$, then `GenLE(i-1)` does nothing.

For example, starting with $+a_1 b_1 a_2 b_2$ and executing the calling sequence `GenLE(2)`, `Switch(2)`, `GenLE(2)` on the poset shown in Figure 1 leads to the trace of calls shown in Figure 6.

We now follow with the details of our implementation. The reader should refer to the Pascal procedure `GenLE` of Figure 8.

The implementation of the algorithm maintains four global arrays: array `le` is the linear extension; array `li` is its inverse. Arrays `a` and `b` store the elements a_i and b_i. In our discussion of the algorithm, a_i and b_i are considered to be fixed at the outset and unchanging throughout the run of the algorithm. The arrays will be maintained so that `a[i]` always contains the value of the leftmost of the ith pair, and

Procedure Call			Linear Extension
			$+a_1b_1a_2b_2$
`GenLE(2)`			
	`GenLE(1)`		
		`Move(`$b_1, \rightarrow$`)`	$+a_1a_2b_1b_2$
		`Switch(0)`	$-a_1a_2b_1b_2$
		`Move(`$b_1, \leftarrow$`)`	$-a_1b_1a_2b_2$
	`Switch(1)`		$-b_1a_1a_2b_2$
	`GenLE(1)`		
		`Switch(0)`	$+b_1a_1a_2b_2$
`Switch(2)`			$+b_1a_1b_2a_2$
`GenLE(2)`			
	`GenLE(1)`		
		`Move(`$a_1, \rightarrow$`)`	$+b_1b_2a_1a_2$
		`Switch(0)`	$-b_1b_2a_1a_2$
		`Move(`$a_1, \leftarrow$`)`	$-b_1a_1b_2a_2$
	`Switch(1)`		$-a_1b_1b_2a_2$
	`GenLE(1)`		
		`Switch(0)`	$+a_1b_1b_2a_2$

FIG. 6. *The trace of the calling sequence for the poset of Figure 1.*

`b[i]` contains the rightmost. Also, the current sign, either plus (+) or minus (−), is maintained.

The boolean function `Right(x)` is used to determine whether the element `x` can move to the right. It operates in $O(1)$ time as follows:

`Right(b[i])`: Returns true only if `b[i]` is incomparable with the element to its right in the array `le`.

`Right(a[i])`: Returns true only if `a[i]` is incomparable with the element to its right in the array `le` and the element to the right is not `b[i]`.

We now describe our preprocessing. We successively strip off pairs a_i, b_i of minimal elements for $i = 1, 2, \ldots$ until there are no elements left. If a unique minimum element is encountered then it is simply deleted and does not become part of a pair. Let `MaxPair` be the index of last pair of minimal elements we strip from $\mathcal{P}$, the remainder of $\mathcal{P}$ being a total ordering, or empty. This preprocessing is detailed in Figure 7. Note that `MaxPair` is not uniquely determined by the poset, but that it depends on the order in which the elements are stripped from $\mathcal{P}$.

We say the linear extension l is in *proper order up to* i if for all $1 \leq j \leq i$, the elements a_j and b_j are adjacent in l, and l induces the orders $a_1a_2\ldots a_ia_h$ and $a_1a_2\ldots a_ib_h$ for all h, where $i < h \leq$ `MaxPair`. The initial linear extension of the listing must be properly ordered up to `MaxPair`; the preprocessing of Figure 7 does this.

Assuming that `Right( b[MaxPair+1] )` is false, the initial call is simply `GenLE( MaxPair + 1 )`; this is the same as the following procedure calls, which we call the *calling sequence.*

```
GenLE( MaxPair ); Switch( MaxPair ); GenLE( MaxPair );
```

The algorithm consists of executing the preprocessing, setting `IsPlus` to plus (+),

```
$i \leftarrow j \leftarrow 0$;
$\mathcal{Q} \leftarrow \mathcal{P}$;
while $S(\mathcal{Q}) \neq \emptyset$ do
    if $\mathcal{Q}$ has exactly one minimal element $x$ then begin
        $j \leftarrow j + 1$;
        $le[j] \leftarrow x$;
        $\mathcal{Q} \leftarrow \mathcal{Q} \backslash \{x\}$;
    end else begin
        let $a', b'$ be any two minimal elements of $\mathcal{Q}$;
        $i \leftarrow i + 1$;
        $j \leftarrow j + 2$;
        $a[i] \leftarrow a'$;
        $b[i] \leftarrow b'$;
        $le[j-1] \leftarrow a'$;
        $le[j] \leftarrow b'$;
        $\mathcal{Q} \leftarrow \mathcal{Q} \backslash \{a', b'\}$;
    end {else}
end {while}
MaxPair $\leftarrow i$;
```

FIG. 7. *The preprocessing routine.*

and then executing the calling sequence.

A Pascal procedure implementing `GenLE` is given in Figure 8. We now prove the following theorem.

THEOREM 4.1. *The algorithm* `GenLE` *generates the linear extensions along a Hamiltonian path in $G'(\mathcal{P}) \times K_2$.*

Proof. In order to prove the theorem, we first prove the following proposition.

PROPOSITION 4.2. *Let the linear extension in array* `le` *be $\xi = \delta a_i b_i \gamma$, and let ξ be properly ordered up to i. Then for each linear extension $l \in E(\mathcal{P} + a_i b_i + \gamma)$,* `GenLE(i)` *generates each of $+l$, $-l$ exactly once. Furthermore, if $i = 1$, then the last extension generated is $-\xi$, and if $i > 1$ then the last extension generated is $+\xi'$, where ξ' differs from ξ by a transposition of a_{i-1} and b_{i-1}.*

Proof. The proof proceeds by induction on i. If $i = 1$ the recursive calls `GenLE(0)` do nothing, `Switch(0)` just changes the sign, and δa_1 is induced by $\mathcal{P}$.

It is easy to confirm that the algorithm in Figure 8, when stripped of its recursive calls, and in which `Switch` just changes the sign, simply follows the path indicated in Figure 2. In this case, `GenLE(1)` just finds a Hamiltonian path from $+\xi$ to $-\xi$ through $G'(\mathcal{Q}) \times K_2$, where $\mathcal{Q}$ is the 2B-poset $\mathcal{P} + a_1 b_1 + \gamma$.

If $i > 1$, then assume without loss of generality that the sign in storage when `GenLE` is invoked is "+". There are α, β, γ such that $+\xi = +\alpha a_{i-1} b_{i-1} \beta a_i b_i \gamma$. Because of the way the preprocessing selects the pairs a_j, b_j, we are assured that $\mathcal{P}$ induces the order $\beta(a_i + b_i)$ (of course, β could be empty).

As mentioned before, the basic structure of the algorithm, stripped of recursive calls, follows the Hamiltonian path in a 2B-poset as indicated in Figure 2, where `Switch` now transposes a_{i-1} and b_{i-1}; therefore it generates $+E' = +E(\mathcal{P} + \alpha(a_{i-1} + b_{i-1})\beta(a_i b_i + \gamma))$. Each linear extension in $+E'$ is properly ordered up to $i - 1$.

As each such linear extension $+l = +\alpha a_{i-1} b_{i-1} \beta \zeta$ (or $+l' = +\alpha b_{i-1} a_{i-1} \beta \zeta$), where $\zeta \in E(a_i b_i + \gamma)$, is generated, `GenLE(i-1)` is called on $+l$. By the inductive

```
procedure GenLE ( i : integer );
var mrb,mra,mla,x : integer;  typical : boolean;
begin
if i > 0 then begin
  GenLE( i-1 );
  mrb := 0;   typical := false;
  while Right( b[i] ) do begin
     mrb := mrb + 1;
     Move( b[i], riht );  GenLE( i-1 );
     mra := 0;
     if Right( a[i] ) then begin
        typical := true;
        repeat
           mra := mra + 1;
           Move( a[i], riht );  GenLE( i-1 );
        until not Right( a[i] );
     end {if};
     if typical then begin
        Switch( i-1 );  GenLE( i-1 );
        if odd( mrb ) then mla := mra - 1 else mla := mra + 1;
        for x := 1 to mla do begin
           Move( a[i], left );  GenLE( i-1 );
        end;
     end {if};
  end {while};
  if typical and odd( mrb )
     then Move( a[i], left )
     else Switch( i-1 );
  GenLE( i-1 );
  for x := 1 to mrb do begin
     Move( b[i], left );  GenLE( i-1 );
  end;
end {if};
end {GenLE};
```

FIG. 8. *Pascal procedure* GenLE.

hypothesis, that call generates $\pm E(\mathcal{P} + a_{i-1}b_{i-1} + \beta\zeta)$ (or $\pm E(\mathcal{P} + b_{i-1}a_{i-1} + \beta\zeta)$, respectively), starting at $+l$ and ending at $+l'$ if $i > 2$, and ending at $-l$ if $i = 2$. Since there are an even number of vertices in the product of the graph with a edge, there are an even number of calls to `GenLE(i-1)`. Thus if $i > 2$ then the sign of the final permutation is unchanged, and if $i = 2$, then the relative ordering of a_{i-1} and b_{i-1} is unchanged. The union over all such ζ is $\pm E(\mathcal{P} + a_{i-1}b_{i-1} + a_ib_i + \gamma) \cup \pm E(\mathcal{P} + b_{i-1}a_{i-1} + a_ib_i + \gamma) = \pm E(\mathcal{P} + a_ib_i + \gamma)$. □

Let a, b be `a[MaxPair]`, `b[MaxPair]` respectively, and suppose the calling sequence is executed on the preprocessed poset $\mathcal{P}$. By the proposition, the first call to GenLE generates $\pm E(\mathcal{P} + ab)$; then a and b are transposed, and then $\pm E(\mathcal{P} + ba)$ is generated. Therefore, $E(\mathcal{P})$ is generated, and the Theorem is proved. □

In analyzing the time complexity of the algorithm, we assume that `Right`, `Switch`, and `Move` can be implemented in constant time. This is easily accomplished as long as the inverse `li` of `le` is maintained. Each call to `Move` and `Switch` generates one more linear extension. Observe that the call `GenLE(i)` generates at least two calls to `GenLE(i-1)`. Each iteration of a while-loop or for-loop in the algorithm executes a `Move`, thereby generating a linear extension. The only occasion in which GenLE can be recursively called and no linear extension generated is when $i = 0$, and this happens at most once per linear extension generated. Therefore, the algorithm runs in constant time per linear extension, when generating $\pm E(\mathcal{P})$. By suppressing the linear extensions which are prefixed with "$-$", we generate $E(\mathcal{P})$ in constant amortized time. Another way to think of the preceding argument is to consider the underlying computation tree, where each internal node is a recursive call and each leaf is a linear extension. The total amount of computation can be divided up so that each node is assigned a constant amount of computation. Since each internal node has at least two children, the number of leaves is greater than the number of internal nodes and therefore the total amount of computation is proportional to the number of leaves.

Observe that the generation of the minus ("$-$") vertices only occurs when $i = 1$ in Algorithm `GenLE`. This suggests that $i = 1$ be treated as a special case and that minus ("$-$") vertices be omitted entirely by simply skipping to the next plus ("+") vertex. If this is done, then it saves some computation but the same list of extensions is produced as before, and successive extensions can differ by a large number of transpositions.

If one only wants to compute the number of extensions, then some computation can be saved by only computing the number of vertices at the $i = 1$ level of the recursion, and not generating the extensions explicitly; i.e., never moving a_1 and b_1. The number of vertices (extensions) can be determined from $mr(a) =$ `mra` and $mr(b) =$ `mrb`; the number is $(mr(a) + 1)[mr(b) + 1 - mr(a)/2]$. Furthermore, $mr(a)$ and $mr(b)$ change by at most unity from one extension to the next, since only adjacent transpositions are used. This leads to an algorithm whose running time is $O(e(\mathcal{P} \setminus \{a_1, b_1\}))$. In general, we have

$$2 \cdot e(\mathcal{P} \setminus \{a_1, b_1\}) \;\leq\; e(\mathcal{P}) \;\leq\; n(n-1) \cdot e(\mathcal{P} \setminus \{a_1, b_1\}).$$

The lower bound is attained when $a_1 \prec c$ and $b_1 \prec c$ for all elements c of $\mathcal{P} \setminus \{a_1, b_1\}$. The upper bound is attained when a_1 and b_1 are maximal, as well as minimal.

5. Gray Codes for Linear Extensions.

We now show that linear extensions can be listed so that successive extensions differ by at most a few adjacent transpositions. We first show the existence of such listings, and then how to modify the results of the previous sections to show that the set of linear extensions of a poset can be listed so that successive extensions differ by only one or two adjacent transpositions. Let us say that an ordering $\alpha_1, \alpha_2, \ldots, \alpha_{e(\mathcal{P})}$ of the extensions of $\mathcal{P}$ has *delay C* if $D'(\alpha_i, \alpha_{i+1}) \leq C$ for all $0 \leq i < e(\mathcal{P})$, where $\alpha_0 = \alpha_{e(\mathcal{P})}$. Thus we are going to show the existence of a delay 2 ordering of $E(\mathcal{P})$. Furthermore, such a listing can be done in constant amortized time. The existence of a delay 3 ordering is not difficult to show.

If G is a graph then by G^k we denote the graph with the same vertex set as G but which has an edge between every pair of vertices that are connected by a path of length at most k in G. In other words, if M is the incidence matrix of G, then M^k is the incidence matrix of G^k, where arithmetic is done mod 2. The *cube* of G is G^3 and the *square* of G is G^2. A poset $\mathcal{P}$ has a delay k ordering if and only if $G'(\mathcal{P})^k$ is Hamiltonian. A result of Sekanina [19] is that the cube of every connected graph is

Hamiltonian. Since $G'(\mathcal{P})$ is always connected, $G'(\mathcal{P})^3$ is Hamiltonian and a delay 3 ordering exists.

The graph $G'(\mathcal{P})$ is not always 2-connected; otherwise the existence of a delay 2 ordering would by implied by a result of Fleischner [7] which states that the square of every 2-connected graph is Hamiltonian.

Even though $G'(\mathcal{P})$ is not in general 2-connected, the posets with 2-connected transposition graphs are easy to characterize. First, consider the question of which transposition graphs have pendant vertices. If $\mathcal{P}$ consists of two disjoint chains, then $G'(\mathcal{P})$ has two pendant vertices; if $\mathcal{P}$ is a B-poset and is not the disjoint union of two chains, then $G'(\mathcal{P})$ has one pendant vertex; otherwise $G'(\mathcal{P})$ has no pendant vertices.

LEMMA 5.1. *For every poset $\mathcal{P}$, the graph $H(\mathcal{P})$ is 2-connected, where $H(\mathcal{P})$ is $G'(\mathcal{P})$ minus any pendant vertices.*

This may be proven by showing that every pair of incident edges of $H(\mathcal{P})$ is on a 4, 6, or 8-cycle. This lemma does not help us in finding an efficient algorithm for listing a delay 2 ordering of $E(\mathcal{P})$. Instead, we prove it by applying Theorem 3.3 and the following lemma.

LEMMA 5.2. *If G is bipartite and $G \times K_2$ is Hamiltonian, then G^2 is Hamiltonian.*

Proof. Let G be a bipartite graph on n vertices, and let

$$(v_1, x_1), (v_2, x_2), \cdots, (v_{2n}, x_{2n})$$

be a Hamiltonian cycle through $G \times K_2$, where $v_i \in V(G)$ and $x_i \in V(K_2) = \{1, 2\}$, for all $i, 1 \leq i \leq 2n$. Consider the sequence of vertices $S = v_2, v_4, \ldots, v_{2n}$.

Since G is bipartite, so is $G \times K_2$; thus the vertices of S are all the vertices of one partite set of $G \times K_2$. Also, for a vertex u of G, $(u, 1)$ and $(u, 2)$ are adjacent are are therefore in different partite sets of $G \times K_2$. Therefore each vertex of G appears exactly once in S. For each i, the vertices v_i and v_{i+2} are either of distance one in G (if $x_i \neq x_{i+2}$) or of distance two in G (if $x_i = x_{i+2}$). Therefore S is a Hamiltonian cycle in G^2. □

We conjecture that Lemma 5.2 may be extended to graphs which are not bipartite.

CONJECTURE 5.3. *If $G \times K_2$ is Hamiltonian, then G^2 is Hamiltonian.*

A graph is in class $\mathcal{H}(s, t)$ if it has a closed walk that visits every vertex at least s times and at most t times. (See [8], [14].) Thus $\mathcal{H}(1, 1)$ is the class of Hamiltonian graphs. Observe that if $G \times K_2$ is Hamiltonian, then $G \in \mathcal{H}(1, 2)$ (just consider the walk that results when the two copies of G are identified). The converse is not true; if G is the triangle with a pendant edge added to each vertex, then $G \in \mathcal{H}(1, 2)$ but $G \times K_2$ is not Hamiltonian. Theorem 3.3 shows that for every poset $\mathcal{P}$, the graph $G'(\mathcal{P})$ is in $\mathcal{H}(1, 2)$. In general, these graphs are not Hamiltonian. We make a conjecture.

CONJECTURE 5.4. *If $G \in \mathcal{H}(1, 2)$, then G^2 is Hamiltonian.*

The example of $K_{2,6}$ shows that the converse of the conjecture is false.

The proof of Lemma 5.2 is constructive; applying that construction to $G'(\mathcal{P})$ yeilds the following result.

THEOREM 5.5. *The linear extensions of any poset can be generated with delay 2 in constant amortized time.*

Proof. We run the Algorithm `Genle` given in Figure 8, but instead of suppressing the linear extensions with a negative sign, we suppress every other linear extension; i.e., if we generate the list $l_1, l_2, l_3, l_4, l_5, \ldots$, then we output the list $l_1, l_3, l_5, \ldots$. By the proof of Lemma 5.2, this is a delay 2 listing of the linear extensions. It has the same running time as `Genle`, i.e., constant amortized time. □

In the remainder of this section we discuss how to use the algorithm to compute $P(x < y)$ and $h(x)$. We use the version of `GenLE` that generates each extension exactly twice, where each successive extension differs by an adjacent transposition from its predecessor. We first discuss how to compute $P(x < y)$.

Let us define an xy-run to be a maximal sequence of successive extensions where x precedes y. We maintain two arrays of integers, call them S and T. The value of $S[x, y]$ is the sum of the lengths of the previous xy-runs. The value of $T[x, y]$ is the iteration at which the current xy-run started. At each iteration, exactly one adjacent pair, say xy, is transposed. If this occurs at the t-th iteration, then $S[x, y]$ is incremented by $t - T[x, y]$ and $T[y, x]$ is set to t. At the termination of the algorithm, $P(x < y)$ is $S[x, y]$ divided by $2e(\mathcal{P})$. Since only a constant amount of update is done at each iteration the total computation is $O(n^2 + e(\mathcal{P}))$.

To compute $h(x)$ we proceed in a similar fashion. An x-run is a maximal sequence of extensions in which x occupies the same position. Here the value of $S[x]$ is the weighted sum of the lengths of the previous x-runs and $T[x]$ is the iteration at which the current x-run started. At each iteration exactly one adjacent pair, say xy, is transposed. If this occurs at the t-th iteration, then for $z = x, y$, $S[z]$ is incremented by $p[z] * (t - T[z])$ and $T[z]$ is set to t, where $p[z]$ is the position that z occupied in the extension. At termination the value of $h(x)$ is $S[x]$ divided by $2e(\mathcal{P})$.

6. Concluding Remarks.

The algorithm given in section 6 generates the linear extensions of a poset in constant amortized time, an improvement over the $O(n)$ amortized time algorithm of [11], which is the fastest previously known algorithm. A further refinement to the work presented here would be to generate the linear extensions by a "loopfree" algorithm (i.e., constant computation in the worst case in producing a new extension from the current one).

We have fully implemented the counting and generating algorithms in C and found them to be quite efficient in practice. On a Sun SPARCstation SLC, the program generated the 2,702,765 extensions of a 12 element fence poset in 4.2 seconds. These extensions are counted by the Euler numbers; counting took 1.7 seconds. The 199,360,981 extensions of a 14 element fence were counted in 91 seconds and generated in 281 seconds. The 2,674,440 extensions of the 2 by 14 grid were generated in 8.3 seconds. These extensions are counted by the Catalan numbers.

Enumeration of the linear extensions of a poset has recently been shown to be #P-complete. The algorithm in section 3 constitutes the first constant amortized time algorithm for generating a naturally defined class of combinatorial objects where the associated counting problem is #P-complete. This leads to some interesting questions about the complexity of generating other combinatorial objects for which counting is #P-complete.

Do all #P-complete problems admit constant amortized time generation algorithms? In asking this question we assume that an initial object has been supplied as part of the input. For example, for Hamiltonian cycles the input would consist of a graph G and a Hamiltonian cycle in G; similarly, in this paper we have assumed that $12 \cdots n$ is a linear extension of the input poset.

Intuitively, if the existence question is difficult (NP-complete), then the generation question will be difficult as well, but we have not answered, or even formalized, this intuition. Even if the existence question is easy (in P), there are many problems for which the complexity of generating is unknown. For example, can the ideals of a given input poset be generated in constant amortized time? Counting ideals is #P-complete, but finding an ideal is trivial. What about generating the spanning trees of a graph in constant amortized time? Finding a spanning tree and counting the number of such trees are both in P, but no one has discovered a constant amortized time algorithm for generating the spanning trees. Is there, in fact, any interesting relationship between constant amortized time generation algorithms and the complexity of existence and/or counting? Some related questions, for various polynomial time complexity measures (instead of constant amortized time), were considered by Johnson, Yannakakis, and Papadimitriou [9].

Acknowledgements. We would like to thank Derek Corneil, Mike Fellows, and Carla Savage for helpful discussions, Malcolm Smith for work on the figures, and especially Ken Wong for implementing the algorithms and carefully reading the manuscript.

Note added in proof. As was pointed out by L. Babai (private communication), it is easy to contrive a #P-complete object that can be generated quickly by starting with a #P-complete object, and pumping up the number of instances by taking the union with an easily-counted, easily-generated, but more numerous object. For example, given a poset $\mathcal{P}$ on n vertices, consider the set $\{x : x \text{ is an ideal of } \mathcal{P}\} \cup \{x : x \subseteq [n^2]\}$. This set is #P-complete to count and, given an appropriate representation, is easy to generate in constant amortized time.

A counterexample on twenty-four vertices to conjectures 5.3 and 5.4 has recently been found by J. van den Heuvel (private communication).

REFERENCES

[1] *Webster's New Collegiate Dictionary*, Merriam, 1980.
[2] M. Aigner, *Combinatorial Theory*, Springer-Verlag, 1979.
[3] V. Bouchitte and M. Habib, *The calculation of invariants for ordered sets*, Algorithms and Order (ed. I. Rival), (1989), pp. 231–279.
[4] G. Brightwell and P. Winkler, *Counting linear extensions is #P-complete*, Order, 8 (1992), pp. 225–242.
[5] M. Buck and D. Wiedemann, *Gray codes with restricted density*, Discrete Math., 48 (1984), pp. 163–171.
[6] P. Eades, M. Hickey, and R. Read, *Some Hamilton paths and a minimal change algorithm*, JACM, 31 (1984), pp. 19–29.
[7] H. Fleischner, *The square of every two-connected graph is Hamiltonian*, J. Combin. Theory (B), 16 (1974), pp. 29–34.
[8] B. Jackson and N. C. Wormald, *k-walks of graphs*, Australasian Journal of Combinatorics, 2 (1990), pp. 135–146.
[9] D. Johnson, M. Yannakakis, and C. Papadimitriou, *On generating all maximal independent sets*, Information Processing Letters, 27 (1988), pp. 119–123.
[10] S. Johnson, *Generation of permutations by adjacent transpositions*, Math. Comp., 17 (1963), pp. 282–285.
[11] A. Kalvin and Y. Varol, *On the generation of all topological sortings*, J. Algorithms, 4 (1983), pp. 150–162.
[12] D. Knuth and J. Szwarcfiter, *A structured program to generate all topological sorting arrangements*, Information Processing Letters, 2 (1974), pp. 153–157.
[13] J. Lenstra, A. Rinnooy Kan, and P. Brucker, *Complexity of machine scheduling problems*, Annals of Discrete Math., 1 (1977), pp. 343–362.
[14] G. Pruesse, *A generalization of hamiltonicity*, Tech. Report 236/90, U. Toronto, 1990.
[15] G. Pruesse and F. Ruskey, *Generating the linear extensions of certain posets by transpositions*, SIAM J. Discrete Math., 4 (1991), pp. 413–422.
[16] F. Ruskey, *Adjacent interchange generation of combinations*, J. Algorithms, 9 (1988), pp. 162–180.
[17] ———, *Research problem 90*, Discrete Math., 70 (1988), pp. 111–112.

[18] ———, *Generating linear extensions of posets by transpositions*, J. Combinatorial Theory (B), 54 (1992), pp. 77–101.

[19] M. Sekanina, *On an ordering of the set of vertices of a connected graph*, Publ. Fac. Sc. Brno., 412 (1960), pp. 137–141.

[20] G. Stachowiak, *Hamilton paths in graphs of linear extensions for unions of posets*, SIAM J. Discrete Math., 5 (1992), pp. 199–206.

[21] R. Stanley, *Enumerative Combinatorics, Vol. I*, Wadsworth, 1986.

[22] H. Steinhaus, *One Hundred Problems in Elementary Mathematics*, Basic, 1964.

[23] H. Trotter, *Algorithm 115: Perm*, Comm. ACM, 5 (1962), pp. 434–435.

[24] Y. Varol and D. Rotem, *An algorithm to generate all topological sorting arrangements*, Computer J., 24 (1981), pp. 83–84.

[25] M. Wells, *The elements of combinatorial computing*, Pergammon Press, 1971.

[26] H. Wilf, *Combinatorial Algorithms, An Update*, SIAM, 1989.