A Gray Code for the Ideals of a Forest Poset

Yasunori Koda and Frank Ruskey

Department of Computer Science

University of Victoria

Victoria, B.C. V8W 3P6, Canada

Abstract

We present two algorithms for listing all ideals of a forest poset. These algorithms generate ideals in a Gray Code manner; that is, consecutive ideals differ by exactly one element. Both algorithms use storage $O(n)$, where n is the number of elements in the poset. On each iteration, the first algorithm does a partial traversal of the current ideal being listed and runs in time $O(nN)$, where N is the number of ideals of the poset. The second algorithm mimics the first, but eliminates the traversal and runs in time $O(N)$. This algorithm has the property that the amount of computation between successive ideals is $O(1)$; such algorithms are said to be loopless.

1 Introduction

Gray Codes find application in Electrical Engineering, Computer Science, and Mathematics. The classical Binary Reflected Gray Code is a way of listing all 2^n length n bitstrings so that successive bitstrings differ by a single bit. Here we consider bitstrings as the characteristic vectors of the ideals of a partially ordered set and ask whether all ideals can be listed so that successive ideals differ by a single poset element. We show that such a listing is possible if the Hasse diagram of the poset is a forest. The Binary Reflected Gray code can be generated by a loopless algorithm; we show that a loopless algorithm also exists in the more general case of ideals of a forest poset.

Generating the ideals of a poset has several applications in Operations Research and discrete optimization. Steiner [16] cites the use of procedures for generating ideals as an integral part of dynamic programming algorithms for precedence constrained scheduling problems, assembly line balancing, project scheduling, and reachability in reliability networks. In the context of precedence constrained scheduling problems tree posets are "an important special case" (Coffman and Denning [3]).

The problem of generating the ideals of a tree poset lexicographically was considered by Ruskey [17] and was motivated by a problem in network partitioning. Let

G be an undirected graph with a capacity $c(e)$ associated with each edge e and a weight $w(v)$ associated with each vertex v. Given a constant W and a distinguished vertex v, consider the problem of finding the minimum cut $(X, \overline{X})$ such that $v \in X$ and $\sum_{x \in X} w(x) \leq W$. This problem arose in assigning tasks in a distributed system. Unfortunately, the problem is NP-complete.

In Hu and Ruskey [8] and Rao, Stone, and Hu [15] an enumerative approach to this problem was outlined. Let T be the cut tree of the network and regard T as being rooted at v. These papers showed that the optimum solution to the partitioning problem must be a cut $(X, \overline{X})$ for which the set of vertices in X are connected in T; in other words, X must be an ideal of T. Thus, by generating all ideals of T, the optimum solution may be obtained. In this regard it is clear that a Gray Code listing will have computational benefits, since it is easiest to modify a cut if only one vertex changes.

Aside from the motivations from discrete optimization mentioned above, there is combinatorial interest in the set of ideals of forest posets and in Gray codes for ideals in general. For example, Meir and Moon [12] derive enumerative results for ideals of forest posets and Jamison [9], [10] discusses extremal properties.

We believe that posets provide a natural setting for working with combinatorial Gray Codes. This point of view, for linear extensions of posets, has been explored, for example, in Ruskey [18]. The two most well-known Gray Codes are the "Binary Reflected Gray Code" (BRGC) for generating subsets and the Johnson-Trotter algorithm for generating permutations. The results of [18] provide a natural generalization of the Johnson-Trotter algorithm. The results of the current paper provide a natural generalization of known results about the BRGC.

If the forest poset consists of disjoint chains of m_i elements for $i = 1, 2, \ldots, p$, then the set of ideals is isomorphic to the product space

$$\{0, 1, \ldots, m_1\} \times \{0, 1, \ldots, m_2\} \times \cdots \times \{0, 1, \ldots, m_p\}.$$

In this case, our first algorithm reduces to Algorithm 8 of Stanton and White [20] (see also Exercise 5.37 of [14] and its solution in Fill and Reingold [6]). If all $m_i = k - 1$, then the product space consists of all k-ary strings, and if $m_i = i$, then the product space can be regarded as the set of inversion vectors of all $p!$ permutations. Thus, our results provide a single loopfree algorithm for generating these objects as well as the ideals of arbitrary forests.

In general, our poset terminology is adapted from Stanley [19]. We now introduce our particular notation and terminology.

A partially ordered set (poset) $\mathcal{P}$ is a set of elements $S(\mathcal{P})$ together with an antisymmetric, reflexive, transitive binary relation $R(\mathcal{P})$. If $(x, y) \in R(\mathcal{P})$ then we write $x \preceq y$ and if in addition $x \neq y$ then we write $x \prec y$. If $x, y \in \mathcal{P}$, then we say y *covers* x if $x \prec y$ and no element $z \in P$ satisfies $x \prec z \prec y$. An *ideal*, I, of a poset $\mathcal{P}$ is a subset of $S(\mathcal{P})$ such that $x \in I$ and $y \preceq x$ implies that $y \in I$.

If $x \neq y$ and neither of the relations $x \prec y$ or $y \prec x$ then we write $x \parallel y$ and say that x and y are *incomparable*. An *antichain*, A, is a subset of $S(\mathcal{P})$ in which every two elements are incomparable. Observe that the set of maximal elements in an ideal form an antichain, and that this provides a one-to-one correspondence between the ideals and the antichains of a poset. The poset that is an antichain of n elements is denoted $\mathcal{A}_n$.

A *forest poset*, $\mathcal{F}$, is one in which every element covers at most one other element. A forest poset with a minimum element is called a *tree poset*. The set of minimal elements are the *roots* of the trees in the forest; this set is denoted $Root(\mathcal{F})$. The set of maximal elements are the *leaves* of the forest; this set is denoted $Leaf(\mathcal{F})$. If v covers an element w then v is a *child* of w and w is the *parent* of v. The set of children of v is denoted $Children(v)$ and the parent of v, assuming it has one, is denoted $parent(v)$. If $parent(v) = w$ then $Sibling(v) = Children(w)$; if v has no parent then $Sibling(v) = Root(\mathcal{F})$.

The set of ideals of a poset $\mathcal{P}$, ordered by set inclusion, form a distributive lattice $J(\mathcal{P})$. It is well known that for every finite distributive lattice L there is a unique poset $\mathcal{P}$ for which $J(\mathcal{P})$ is isomorphic to L. The number of elements in $J(\mathcal{P})$ (i.e., the number of ideals of $\mathcal{P}$) is denoted $i(\mathcal{P})$. In a slight abuse of notation, we also use $J(\mathcal{P})$ to denote the Hasse diagram, regarded as an undirected graph, of the lattice of ideals of $\mathcal{P}$. In this context we refer to $J(\mathcal{P})$ as the *ideal graph* of $\mathcal{P}$. The central question that we would like to pose in this paper is:

For which posets $\mathcal{P}$ does $J(\mathcal{P})$ have a Hamiltonian path and how can the ideals along such a path be quickly generated?

In this paper we take some first steps in answering this question.

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

We now introduce two standard ways of combining two posets to get a new poset. We say that two posets $\mathcal{P}$ and $\mathcal{Q}$ are *disjoint posets* if $S(\mathcal{P})$ and $S(\mathcal{Q})$ are disjoint. The *direct sum* of two disjoint posets $\mathcal{P}$ and $\mathcal{Q}$ is the poset $\mathcal{P} + \mathcal{Q}$ on the union $S(\mathcal{P}) \cup S(\mathcal{Q})$ such that $x \preceq y$ in $\mathcal{P} + \mathcal{Q}$ if either (a) $x, y \in \mathcal{P}$ and $x \preceq y$ in $\mathcal{P}$, or (b) $x, y \in \mathcal{Q}$ and $x \preceq y$ in $\mathcal{Q}$. The *ordinal sum* of two disjoint posets $\mathcal{P}$ and $\mathcal{Q}$ is the poset $\mathcal{P} \oplus \mathcal{Q}$ on the union $S(\mathcal{P}) \cup S(\mathcal{Q})$ such that $x \preceq y$ in $\mathcal{P} \oplus \mathcal{Q}$ if either (a) $x \preceq y$ in $\mathcal{P} + \mathcal{Q}$, or (b) $x \in \mathcal{P}$ and $y \in \mathcal{Q}$. A *series-parallel* poset is one which can be recursively built up by applying the operations of direct and ordinal sum, starting with single element posets. Note that a forest poset can be built up by applying the operations of direct sum of two forest posets and the ordinal sum of a forest poset with a single element, again starting with single element posets.

A graph G consists of a vertex set $V(G)$ and an edge set $E(G)$. The *product* of the graphs G and H is the graph $G \times H$ with vertex set $V(G) \times V(H)$, in which $[v, w]$ is adjacent to $[v', w']$ if and only if $v = v'$ and $[w, w'] \in E(G)$ or $w = w'$ and $[v, v'] \in E(H)$. Directly from the definitions we see that

$$J(\mathcal{P}_1 + \mathcal{P}_2) \cong J(\mathcal{P}_1) \times J(\mathcal{P}_2), \tag{1}$$

where $\cong$ denotes graph isomorphism.

If A and B are distinct finite sets of natural numbers, then we say that A is *lexicographically less* than B if the largest element of $(A \cup B) \setminus (A \cap B)$ is in B. For example, $\{1, 3, 5, 6\}$ is lexicographically less than $\{2, 5, 6\}$.

We are interested in efficient algorithms for generating combinatorial objects. Suppose that the objects are represented by sequences of n elements and that the total number of objects is N. A algorithm for generating those objects is said to run in *constant amortized time* if the total amount of computation (excluding output) is $O(N)$. Up to a constant factor, no algorithm can be faster. If the total amount of computation is $O(nN)$, then the algorithm is said to run in *linear amortized time*.

A constant amortized time algorithm is said to be *loopless* (or constant worst-case time) if the amount of computation between successive objects is $O(1)$. Examples of loopless algorithms may be found in Joichi, White, and Williamson [11], Dershowitz [4], and Ehrlich [5].

The algorithm of Ruskey [17] for generating the ideals of a tree poset runs in linear amortized time. A constant amortized time algorithm for generating the ideals of bounded size of a tree poset was described by Beyer and Ruskey [2]. The algorithm of Steiner [16] for generating all ideals of an arbitrary poset runs in linear amortized time. There is also an algorithm of Hikita [7] for generating all ideals of fixed size of a binary tree poset that appears to run in linear amortized time.

In Section 2 we prove the existence of a Gray Code listing of the ideals of a forest poset; that is, where consecutive ideals differ by exactly one element. Then, in Section 3 we present a simple linear amortized time algorithm for producing the Gray Code listing; this algorithm is based on a preorder traversal of the forest. If this algorithm is applied to the antichain $\mathcal{A}_n$ the algorithm reduces to an algorithm for generating the Binary Reflected Gray Code of size n (e.g., Ehrlich [5], Bitner, Ehrlich, and Reingold [1], and Wilf [22]). In Section 4 we eliminate certain redundancy from the algorithm and in Section 5 we show how to attain a loopless algorithm.

The loopless algorithm can be implemented on a restricted version of a *pointer machine*. For a formal definition of pointer machine, see Tarjan [21]. From our point of view, we simply mean that our algorithms can be implemented by Pascal programs which use no arrays and do no arithmetic (but make extensive use of pointers).

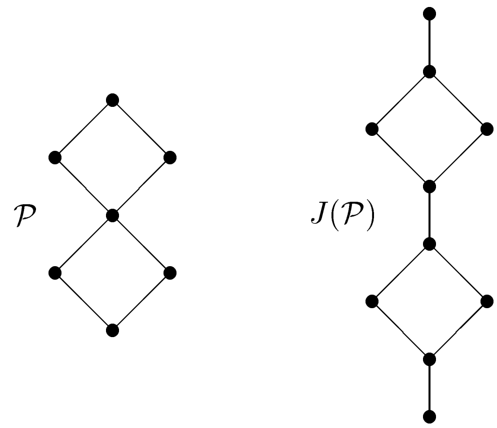

Figure 1: A poset $\mathcal{P}$ for which $d(\mathcal{P}) = 0$ but $J(\mathcal{P})$ has no Hamiltonian path.

2 Existence of a Gray Code

In this section we show that there is a Hamiltonian path in $J(\mathcal{P})$ if $\mathcal{P}$ is a forest poset. This will follow as a consequence of more general results about the ordinal and direct sums of posets. A necessary condition for the existence of a Hamiltonian path in $J(\mathcal{P})$ is that $|d(\mathcal{P})| \leq 1$. However, the condition is not sufficient. Figure 1 shows the series-parallel poset $\mathcal{P} = 1 \oplus (1 + 1) \oplus 1 \oplus (1 + 1) \oplus 1$, for which $d(\mathcal{P}) = 0$ but $J(\mathcal{P})$ has no Hamiltonian path. Let us first determine $d(\mathcal{P})$ if $\mathcal{P}$ is a forest poset.

It is easy to show that $i(\mathcal{P} + \mathcal{Q}) = i(\mathcal{P})i(\mathcal{Q})$ and that $i(\mathcal{P} \oplus \mathcal{Q}) = -1 + i(\mathcal{P}) + i(\mathcal{Q})$. Similarly, we have

$$d(\mathcal{P} + \mathcal{Q}) = d(\mathcal{P})d(\mathcal{Q}), \tag{2}$$

and

$$d(\mathcal{P} \oplus \mathcal{Q}) = \begin{cases} -1 + d(\mathcal{P}) + d(\mathcal{Q}) & \text{if } |\mathcal{P}| \text{ is even} \\ +1 + d(\mathcal{P}) - d(\mathcal{Q}) & \text{if } |\mathcal{P}| \text{ is odd.} \end{cases} \tag{3}$$

These formulas allow us to compute the parity difference of series-parallel posets. For forests we apply (3) with $\mathcal{P} = \mathcal{A}_1$. Since $d(\mathcal{A}_1) = 0$, the parity difference for forests is 0 or 1.

Lemma **2.1** *Let $\mathcal{P}$ and $\mathcal{Q}$ be non-empty posets both of whose ideal graphs have Hamiltonian paths starting at $\emptyset$. Then $J(\mathcal{P} + \mathcal{Q})$ has a Hamiltonian path starting at $\emptyset$. Furthermore, if $|J(\mathcal{P})|$ (or $|J(\mathcal{Q})|$) is even, then $J(\mathcal{P} + \mathcal{Q})$ has a Hamiltonian cycle.*

Proof: Let H be a Hamiltonian path in $J(\mathcal{P})$ and H' be a Hamiltonian path in $J(\mathcal{Q})$. Then $H \times H'$ is a spanning subgraph of $J(\mathcal{P}) \times J(\mathcal{Q})$. Hence, by (1), $H \times H'$ is isomorphic to a spanning subgraph of $J(\mathcal{P} + \mathcal{Q})$. Since H and H' are paths, $H \times H'$ is a $|H|$ by $|H'|$ grid graph. The ideal $\emptyset$ of $J(\mathcal{P} + \mathcal{Q})$ corresponds to the corner vertex $(\emptyset, \emptyset)$ of $H \times H'$. It is easy to construct a Hamiltonian path in a grid graph, starting at a corner. Furthermore, unless both $|H|$ and $|H'|$ are odd, there is a Hamiltonian cycle in $H \times H'$. □

LEMMA **2.2** *Let $\mathcal{P}$ be a poset for which $J(\mathcal{P})$ has a Hamiltonian path starting at $\emptyset$ and ending at $S(\mathcal{P})$, and let $\mathcal{Q}$ be a poset for which $J(\mathcal{Q})$ has a Hamiltonian path starting at $\emptyset$. Then there is a Hamiltonian path in $J(\mathcal{P} \oplus \mathcal{Q})$ starting at $\emptyset$.*

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

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

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

Notice that the condition of Lemma 2.2 that the Hamiltonian path in $J(\mathcal{P})$ ends at $S(\mathcal{P})$ cannot be relaxed since $S(\mathcal{P})$ is a cut vertex of $J(\mathcal{P} \oplus \mathcal{Q})$ (if $\mathcal{P}$ and $\mathcal{Q}$ are both non-empty).

As a consequence of Lemmas 2.1 and 2.2 we have the following theorem, which may be proven by induction on the number of nodes in the forest.

THEOREM **2.1** *If $\mathcal{P}$ is a forest poset, then there is a Hamiltonian path in $J(\mathcal{P})$.*

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

THEOREM **2.2** *Let $\mathcal{F}$ be a forest poset consisting of trees $T_1, T_2, \ldots, T_k$. There is a Hamiltonian cycle in $J(\mathcal{F})$ if and only if $k \geq 2$ and there is at least one tree T_i for which $|J(T_i)|$ is even.*

PROOF: If $k = 1$, then $\emptyset$ is a pendant vertex in $J(\mathcal{F})$. If all $|J(T_i)|$ are odd, then $i(\mathcal{F})$ is odd, and since $J(\mathcal{F})$ is bipartite it cannot have a Hamiltonian cycle. On the other hand, if $k \geq 2$ and there is at least one tree T_i for which $|J(T_i)|$ is even, then by Lemma 2.1 there is a Hamiltonian cycle in $J(\mathcal{F})$. □

3 The First Algorithm

Theorem 2.1 is constructive, but it does not describe a particular Hamiltonian path in the case of direct sum. In this section we will describe an algorithm that finds such a path systematically. In Section 5 we will show how to modify this algorithm so that is is loopless.

Our first algorithm is inspired by the well-known correspondence between counting in binary and the BRGC, as explained, for example, in [14] and [22]. Observe that Algorithm P of Figure 2 can be viewed as counting in binary if "v is changeable" means "bit v is a 0". The bitstring is the characteristic vector of the set St. The BRGC is the sequence of characteristic vectors of F that are output.

The following observation does not depend on the definition of "changeable".

Observation #1: The set St lexicographically increases on each iteration of Algorithm P.

We now modify the definition of changeable so that Algorithm P applies to forest posets.

Throughout this section we deal with a fixed forest poset $\mathcal{F}$ with n elements. Here St is a set of nodes. A node $x \in St$ is said to be *stuck*; a node $x \notin St$ is said to be *free*. It is helpful to visualize membership in St as a status of the node. Thus if $x \in St$ we write $status(x) = stuck$ and if $x \notin St$ then we write $status(x) = free$. Generally speaking, we will use the set St in the algorithms and the status notation in the commentary.

DEFINITION: Let F be an ideal of a forest poset $\mathcal{F}$, and $v \notin St$.

- If $v \notin F$, then v is *augmentable* if v is a root or *parent*$(v) \in F$.
- If $v \in F$, then v is *deletable* if none of the children of v is in F.

In other words, a free node v is augmentable if and only if $v \in \mathit{Root}(\mathcal{F} \setminus F)$, and is deletable if and only if $v \in Leaf(F)$. A node that is augmentable or deletable is said to be *changeable*. Given F and St, if there is a changeable node in $\mathcal{F}$, then $\mathcal{F}$ is *changeable*; otherwise $\mathcal{F}$ is *unchangeable*.

We assume that the forest has been given a *preorder* labeling, and we use this labeling when referring to the nodes of $\mathcal{F}$. A preorder labeling is obtained by recursively labeling a root and then its subtrees; it is the same as a depth-first search labelling. In the figures we assume that the labels increase from left to right. The labels are $1, 2, \ldots, n$.

The algorithm does a preorder traversal of the forest, changing stuck nodes to free, until the first changeable node is encountered; that node is then added to or removed from the tree, according to whether it is augmentable or deletable. It is clear that the running time of Algorithm P is $O(nN)$, where $N = i(\mathcal{F})$. An example of the list of ideals produced by Algorithm P is illustrated in Figure 3. In this figure the ideals have been drawn so that the grid graph structure of $J(\mathcal{P})$ is apparent; ideals are adjacent both vertically and horizontally. The minimum taken at line (P5) is with respect to the preorder labeling. We will refer to the node v selected at line (P5) as the *chosen node.*

The proof of the validity of Algorithm P is a refinement of the proof of the existence of Gray code sequence of ideals of a forest poset. Let $\mathit{Last}(\mathcal{F})$ denote the last ideal output by Algorithm P. Then, by Algorithm $\overline{\mathrm{P}}$ we mean Algorithm P with the statement $F \leftarrow \emptyset$ of line (P1) replaced with $F \leftarrow \mathit{Last}(\mathcal{F})$.

(P1) $F \leftarrow \emptyset$;
(P2) $St \leftarrow \emptyset$;
(P3) Output(F);
(P4) **while** $\mathcal{F}$ is changeable **do begin**
(P5) $v \leftarrow \min\{v \in \mathcal{F} \mid v \text{ is changeable}\}$;
(P6) $St \leftarrow St \cup \{v\} \setminus \{w \in \mathcal{F} \mid w < v\}$;
(P7) **if** $v \in F$ **then** $F \leftarrow F \setminus \{v\}$ **else** $F \leftarrow F \cup \{v\}$;
(P8) Output(F);
(P9) **end**;

Figure 2: Algorithm P

LEMMA **3.1** *Algorithm P produces a Hamiltonian path in $J(\mathcal{F})$ starting at $\emptyset$. Algorithm $\overline{\text{P}}$ produces the same Hamiltonian path, but in opposite order, starting from $Last(\mathcal{F})$.*

PROOF: Our proof is by induction on the number of nodes in $\mathcal{F}$. If $\mathcal{F}$ is empty then both Algorithm P and Algorithm $\overline{\text{P}}$ output $\emptyset$ and halt.

We now assume that $\mathcal{F}$ has at least one node. The forest $\mathcal{F}$ consists of some number of trees, say $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_m$. There are two cases to consider, depending on whether $m = 1$ or $m > 1$.

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

If $m = 1$, then the root is in $Last(\mathcal{F})$ and thus Algorithm $\overline{\text{P}}$ executes exactly as it would on $\mathcal{F} - \{1\}$, except that the root is in all ideals, until F contains only the root. But then, since the root is free, the root is deleted, the emptyset is output, and the algorithm terminates. Note that when $F = \{1\}$, the forest $\mathcal{F} - \{1\}$ is unchangeable.

If $m > 1$ then we regard $\mathcal{F}$ as consisting of a pair of non-empty ideals $\mathcal{F}_L = \mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_p$ and $\mathcal{F}_R = \mathcal{T}_{p+1}, \ldots, \mathcal{T}_m$, where $1 \le p < m$. By the induction hypothesis, Algorithm P first generates all ideals of $\mathcal{F}_L$ ending $Last(\mathcal{F})$, which is unchangeable. The next change is in $\mathcal{F}_R$, and all nodes in $\mathcal{F}_L$ are set free. Thus, by the claim for Algorithm $\overline{\text{P}}$, the ideals of $\mathcal{F}_L$ will now be generated in the opposite order from before. The last ideal of $\mathcal{F}_L$ that is generated is the empty forest, which is unchangeable. Now again, a change is made in $\mathcal{F}_R$ and all the roots in $\mathcal{F}_L$ are set free. The same argument continues until both $\mathcal{F}_L$ and $\mathcal{F}_R$ are unchangeable, terminating the algorithm. Note that $Last(\mathcal{F}) = Last(\mathcal{F}_R)$ if $i(\mathcal{F}_L)$ is even, and $Last(\mathcal{F}) = Last(\mathcal{F}_L) \cup Last(\mathcal{F}_R)$ if $i(\mathcal{F}_L)$ is odd.

A symmetric argument can be given for Algorithm $\overline{\text{P}}$ when $m > 1$. □

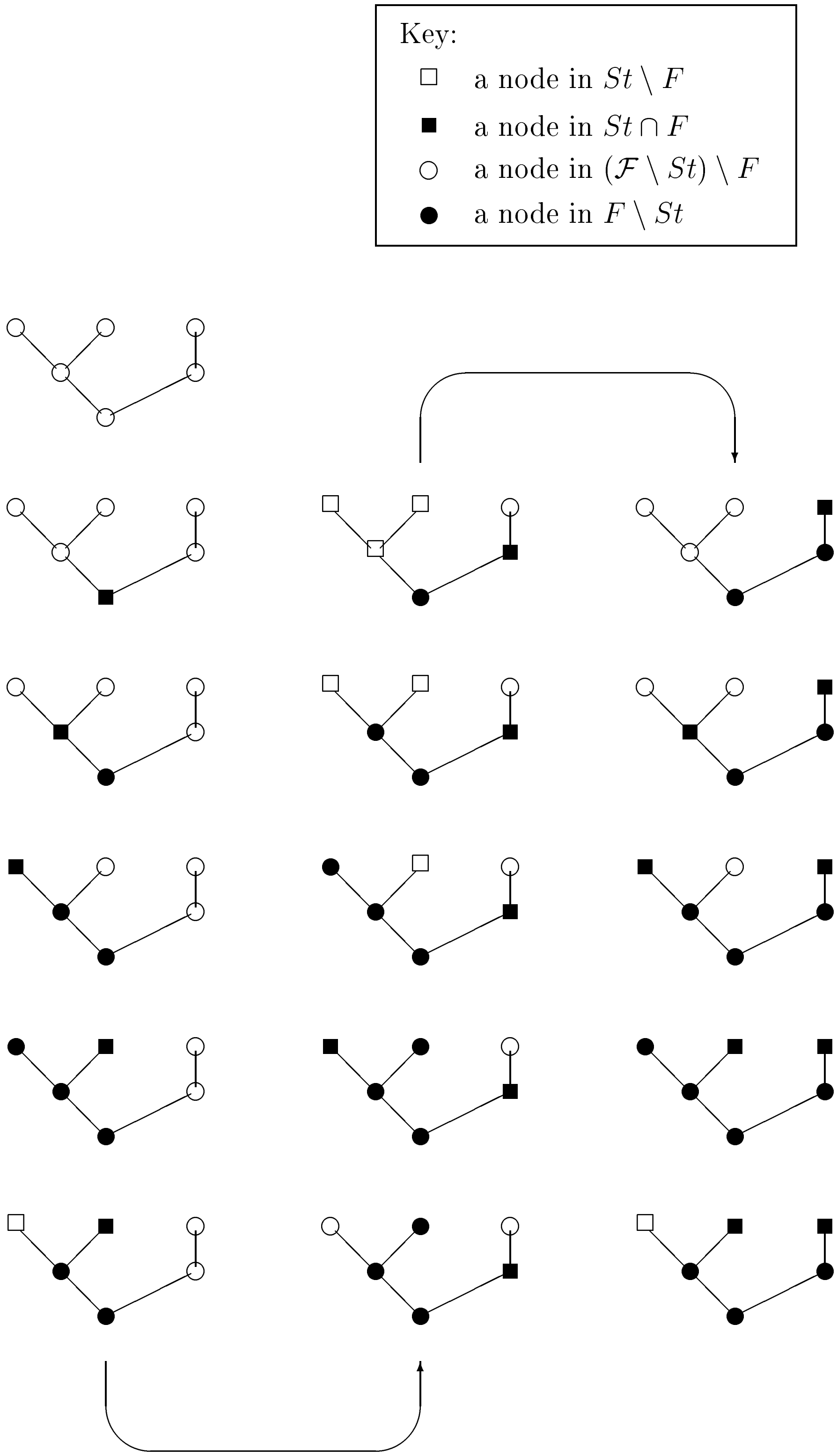

Figure 3: An illustration of the sequence of ideals generated by Algorithm P.

We now make a couple of observations that follow from the preceding proof. Let $\mathcal{T}(v)$ be the subposet of descendants of v (i.e., $S(\mathcal{T}(v)) = \{x \in \mathcal{F} \mid v \prec x\}$). Let $\mathbf{L}$ be the list of ideals produced by Algorithm P on input $\mathcal{T} = \mathcal{T}(v)$, and $\overline{\mathbf{L}}$ the list of ideals produced by Algorithm $\overline{\mathrm{P}}$ on input $\mathcal{T}$. Now suppose that we run Algorithm P on input $\mathcal{F}$ but only output $F \cap \mathcal{T}$ and only when a change to $\mathcal{T}$ has occured. We call this "the list of ideals of $\mathcal{T}$ produced by Algorithm P on input $\mathcal{F}$".

Observation #2: The list of ideals of $\mathcal{T}$ produced by Algorithm P on input $\mathcal{F}$ is $\mathbf{L}, \overline{\mathbf{L}}, \mathbf{L}, \overline{\mathbf{L}}, \ldots, \mathbf{L}'$. The number of $\mathbf{L}$ or $\overline{\mathbf{L}}$ lists is $i(\mathcal{F}')$, where $\mathcal{F}'$ is the subposet of $\mathcal{F}$ induced by those elements $x > v$ such that $x \parallel v$. Thus, $\mathbf{L}' = \mathbf{L}$ if $i(\mathcal{F}')$ is odd and $\mathbf{L}' = \overline{\mathbf{L}}$ if $i(\mathcal{F}')$ is even.

Observation #3: Whenever a node v is deleted, all of its children are stuck.

Observation #4: If $w \in F$ has just been deleted by Algorithm P, where now $F \cap \mathit{Sibling}(w) = \emptyset$, then for all $x \in \mathit{Sibling}(w)$, we have $x \geq w$ and $\mathit{status}(x) = \mathit{stuck}$.

To prove Observations #3 and #4, consider the list $\overline{\mathbf{L}}$ of Observation #2. That list ends $\ldots, \{v, w\}, \{v\}, \emptyset$, if v has at least one child w. From the proof of Lemma 3.1 for Algorithm $\overline{\mathrm{P}}$ when $m = 1$, the subposet $\mathcal{T} - \{v\}$ is unchangeable when v is deleted. Since the ideal is empty in $\mathcal{T} - \{v\}$, the roots of $\mathcal{T} - \{v\}$ must be stuck. Furthermore, w must be the leftmost child of v, since it is the leftmost child that is first added to the ideal $\{v\}$ in the list $\mathbf{L}$.

To close this section we present a formula for the rank of an ideal in the list produced by Algorithm P. Given a forest poset $\mathcal{F}$, the *rank* of an ideal F, denoted $r(F, \mathcal{F})$, is the number of ideals that were generated before it by Algorithm P. Thus, the rank of $\emptyset$ is 0. Denote the trees of $\mathcal{F}$ by $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_m$, and the induced subideals $T_i = F \cap \mathcal{T}_i$. If T is a tree then $\tilde{T}$ denotes the forest obtained by removing the root of T. By $\overline{r}$ denote the rank in the list produced by Algorithm $\overline{\mathrm{P}}$. Given r, the value of $\overline{r}$ can be computed as follows.

$$\overline{r}(F, \mathcal{F}) = -1 + i(\mathcal{F}) - r(F, \mathcal{F})$$

The following formula states a recurrence relation for r. Its proof essentially follows from the proof of Lemma 3.1.

$$r(F, \mathcal{F}) = \begin{cases} 0 & \text{if } F = \emptyset \\ 1 + r(\tilde{F}, \tilde{\mathcal{F}}) & \text{if } m = 1 \\ i(\mathcal{F} \setminus \mathcal{T}_m) r(T_m, \mathcal{T}_m) + r(F \setminus T_m, \mathcal{F} \setminus \mathcal{T}_m) & \text{if } r(T_m, \mathcal{T}_m) \text{ is even} \\ i(\mathcal{F} \setminus \mathcal{T}_m) r(T_m, \mathcal{T}_m) + \overline{r}(F \setminus T_m, \mathcal{F} \setminus \mathcal{T}_m) & \text{if } r(T_m, \mathcal{T}_m) \text{ is odd} \end{cases} \tag{4}$$

The running time of an algorithm based on (4) is $O(|\mathcal{F}|)$ if the quantities $i(\mathcal{F} \setminus \mathcal{T}_m)$ (and their recursive counterparts) have been computed in advance. To be fair, it should be noted that the operations are on integers that can be as large as 2^{n-1}, where $n = |\mathcal{F}|$.

4 Eliminating Redundancy

In order to implement Algorithm P so that it is loopless we first need to understand the operation of Algorithm P a little more deeply. The following two lemmas show that much of the status information of the nodes is redundant. These lemmas express loop invariants of Algorithm P.

LEMMA **4.1** *Let F be an ideal at the beginning of an iteration of Algorithm P.*

(a) If $x \in F \setminus \mathit{Leaf}(F)$, then $\mathit{status}(x) = \mathit{free}$.

(b) If $x \notin F$ and $\mathit{status}(x) = \mathit{free}$, then $\mathit{status}(y) = \mathit{free}$ for all $x \prec y$.

(c) If $x \notin F$ and $\mathit{status}(x) = \mathit{stuck}$, then $\mathit{status}(y) = \mathit{stuck}$ for all $x \prec y$.

LEMMA **4.2** *Let F be an ideal at the beginning of an iteration of Algorithm P, and $x \in \mathit{Leaf}(F)$.*

(a) If $\mathit{status}(x) = \mathit{free}$, then $\mathit{status}(y) = \mathit{stuck}$ for all $y \in \mathit{Children}(x)$.

(b) If $\mathit{status}(x) = \mathit{stuck}$, then $\mathit{status}(y) = \mathit{free}$ for all $y \in \mathit{Children}(x)$.

These lemmas will be proven at the end of this section. The only nodes whose status is not determined by Lemma 4.1 are those in $\mathit{Leaf}(F) \cup \mathit{Root}(\mathcal{F} \setminus F)$. In the case of a node $v \in \mathit{Leaf}(F)$ where v has children, we can use Lemma 4.2 to further limit our attention to which of v or $\mathit{Children}(v)$ is free. We call the resulting set of nodes the *boundary* of F. A formal definition is given below.

DEFINITION **4.1** *Let F be an ideal at the beginning of an iteration of Algorithm P. Then $\mathit{Boundary}(F) = B_1(F) \cup B_2(F) \cup B_3(F) \cup B_4(F)$, where*

$$\begin{aligned}
B_1(F) &= \{x \in \mathit{Leaf}(F) \mid \mathit{status}(x) = \mathit{free}\},\\
B_2(F) &= \{x \in \mathit{Leaf}(F) \mid \mathit{status}(x) = \mathit{stuck} \text{ and } \mathit{Children}(x) = \emptyset\},\\
B_3(F) &= \{x \in \mathit{Root}(\mathcal{F} \setminus F) \mid \mathit{status}(x) = \mathit{free}\},\\
B_4(F) &= \{x \in \mathit{Root}(\mathcal{F} \setminus F) \mid \mathit{status}(x) = \mathit{stuck} \text{ and } \mathit{Sibling}(x) \cap F \neq \emptyset\}.
\end{aligned}$$

Clearly, since the sets are disjoint, B_1, B_2, B_3, B_4 form a partition of $Boundary(F)$. Observe that $Boundary(F)$ is a maximal antichain of $\mathcal{F}$. Knowing the status of the nodes in $Boundary(F)$ allows us to determine the status of all nodes in $\mathcal{F}$. Since the chosen node is in the boundary (either in B_1 or B_3), Algorithm P can be modified to search and update only nodes in the boundary; the modified algorithm is guaranteed to produce exactly the same sequence of ideals as Algorithm P. We need only show how the boundary is to be maintained; that is, which nodes are in the boundary and the status of the nodes in the boundary.

Maintaining the nodes of the boundary is fairly simple since on each iteration the boundary either remains unchanged or changes by deleting the chosen node v and then adding either $parent(v)$ or $Children(v)$.

Maintaining the status of the nodes of the boundary is also quite simple; set $status(v) = stuck$ and for all $x < v$ where x is in the boundary, set $status(x) = free$. We now describe how to do the update in more detail.

We use the set variable B as the boundary. If v is deletable then there are two subcases. If v has a sibling in F, then B is unchanged and now $v \in B_4$. If v has no sibling in F, then v is replaced in B by $x = parent(v)$. Since $x < v$, by line (P6) of Algorithm P, the node x is now in B_1.

If v is augmentable then there are again two subcases. If v has no children, then B is unchanged and now v is in B_2. If v has children, then v is replaced in B by $Children(v)$. By Lemma 4.2, for all $x \in Children(v)$, we set $status(x) = free$, and so $x \in B_3$.

A pseudo-code algorithm using the updates described above is given in Figure 4 as Algorithm L.

We now present proofs of Lemmas 4.1 and 4.2. Since the lemmas state loop invariants we need to check two things. First, that the condition is true before entering the loop; secondly, that if the condition is true on the previous iteration, then it is true on the current iteration. We introduce the notation $NonLeaf(F)$ for $F \setminus Leaf(F)$.

PROOF: [Lemma 4.1] Initially F is empty and all nodes are free. Thus all three conditions (a), (b), and (c) are satisfied before entering the loop. Let F' be the ideal at the beginning of the previous iteration and F the ideal at the beginning of the current iteration. Let v be the chosen node.

(a) The only node to become stuck is v and $v \notin NonLeaf(F)$. If v was deleted then $NonLeaf(F) = NonLeaf(F') \setminus \{v\}$. If v was augmented and v has a parent then $NonLeaf(F) = NonLeaf(F') \cup \{parent(v)\}$; but $parent(v)$ was set free, at line (P6). If v was augmented and v has no parent, then $NonLeaf(F) = NonLeaf(F')$. Thus, in all cases the invariant is maintained.

(b) Here the invariant is maintained because the only node to become stuck is v, nodes $x > v$ are unchanged, and nodes $x < v$ are all set free.

$B \leftarrow Root(\mathcal{F})$;
$F \leftarrow St \leftarrow \emptyset$;
Output(F);
while $\{x \in B \mid x \notin St\} \neq \emptyset$ **do begin**
    $v \leftarrow \min\{x \in B \mid x \notin St\}$;
    $St \leftarrow St \cup \{v\} \setminus \{x \in B \mid x < v\}$;
    if $v \in F$ **then begin** (* delete v *)
        $F \leftarrow F \setminus \{v\}$;
        if $Sibling(v) \cap F \neq \emptyset$ **then** (* do nothing *)
        else begin
            $B \leftarrow B \setminus Sibling(v) \cup \{parent(v)\}$;
            $St \leftarrow St \setminus \{parent(v)\}$;
        end;
    end else begin (* augment v *)
        $F \leftarrow F \cup \{v\}$;
        if $v \in Leaf(\mathcal{F})$ **then** (* do nothing *)
        else begin
            $B \leftarrow B \setminus \{v\} \cup Children(v)$;
            $St \leftarrow St \setminus Children(v)$;
        end;
    end;
    Output(F);
end;

Figure 4: Algorithm L.

(c) If the chosen node v is augmented, then the invariant is clearly maintained because $v \in F$, the status of nodes $x > v$ are unchanged, and the nodes $x < v$ are all free. If the chosen node v is deleted, then its status becomes stuck and we must verify that all of its children are stuck. But this is simply Observation #3. □

Proof: [Lemma 4.2] Let $v \in Leaf(F)$.

(a) If $status(v) = free$, then the next change to a node $x \geq v$ must be to v, which therefore must be deleted. Thus by Observation #3 all of v's children must be stuck.

(b) If v was not stuck on the previous iteration, then v must have just been augmented. Thus in the previous iteration v was free and not in F, and by Lemma 4.1(b), the children of v must have been free. □

5 A Loopless Implementation

In this section we will explain how to modify Algorithm L so that it is loopless. There are three questions that need to be answered in order to attain a loopless algorithm.

Q1: How is the boundary set B to be maintained so that the updates at line (L11) and (L18) can be done in constant time?

Q2: How can the set of stuck nodes St be maintained so that the chosen node can be determined at line (L5) in constant time and so that the updates at lines (L6), (L12) and (L19) can be done in constant time?

Q3: How can the test $Sibling(v) \cap F = \emptyset$ at line (L9) be done in constant time?

We will address each of these questions in turn. The input forest $\mathcal{F}$ is represented as a static multi-linked structure where at each node we have pointers to its parent, left sibling, right sibling, leftmost child, and rightmost child (recall that we think of siblings as being ordered from left to right by the preorder numbering).

[Q1] The set B is maintained as an ordered doubly linked list, where list ordering is by increasing node number. Call the two pointer fields of the doubly linked list *next* and *back*; these fields are part of each node. The two updates are related. At line (L11), to remove a node's siblings we go to its parent in order to access its leftmost and rightmost siblings, which are the leftmost and rightmost children of its parent. At line (L18), to add the children of a node we can just use the leftmost and rightmost children; we just need to make sure that all children are linked together.

[Q2] The set St is maintained by making use of a clever implementation of the BRGC as explained in [14]. However, the implementation of [14] operates on a fixed size array, and we have to operate on a varying size linked list representing B. Let us first see how to implement the BRGC using a linked list and then adapt the idea to our problem.

$$
\begin{array}{lll}
\textbf{repeat} & & \\
\quad q \leftarrow run(head); & & \{i \leftarrow \tau_0\} \\
\quad r \leftarrow next(q); & & \\
\quad bit(q) \leftarrow 1 - bit(q); & & \{g_i \leftarrow \overline{g_i}\} \\
\quad run(head) \leftarrow head; & & \{\tau_0 \leftarrow 1\} \\
\quad run(q) \leftarrow run(r); & & \{\tau_{i-1} \leftarrow \tau_i\} \\
\quad run(r) \leftarrow r; & & \{\tau_i \leftarrow i+1\} \\
\textbf{until } q = tail; & &
\end{array}
$$

Figure 5: Algorithm B

So imagine that we have a fixed size linked list and that we wish to simulate the transformation of St into its lexicographic successor.

We endow each node of the linked list with an additional pointer field, run, which is used to represent the set St. The basic idea is that the run field will be used to record runs of elements in St in a clever manner. A "run" in St is a maximal set of elements of the form $i, i+1, \ldots, i+k-1 \in St$. For the sake of description, suppose that the nodes are labelled $1, 2, \ldots, n$ and that $head = p_1, p_2, \ldots, p_n, p_{n+1} = tail$ are a sequence of pointers to the nodes in the list, where the $(n+1)$st node is a dummy node. By definition, we assign $0, n+1 \notin St$.

Here's how run is defined. If $i \notin St$ then $run(p_i) = p_i$. If $i \in St$ and $i-1 \in St$ then $run(p_i) = p_i$. If, for $j \geq 1$, we have $i, i+1, \ldots, i+j-1 \in St$, $i-1, i+j \notin St$, then $run(p_i) = p_{i+j}$. In other words, the first node of a run points to the element that follows the run and all other nodes point to themselves.

If $run(head) = p_k$ then $St \cup \{k\} \setminus \{1, 2, \ldots, k-1\}$ is the lexicographic successor of St. Thus, we can update run by the three assignment statements $run(head) \leftarrow head$, $run(p_k) \leftarrow run(p_{k+1})$, and $run(p_{k+1}) \leftarrow p_{k+1}$. Note that this update works correctly even if $run(head) = head$.

See Figure 5 for pseudo-code of the above discussion, which we call Algorithm B. Initially, $run(p_i) = p_i$ for all i. The process terminates when $St = \{1, 2, \ldots, n\}$, i.e., when $run(head) = tail$. The resulting algorithm is almost identical to the loopless algorithm for the BRGC as described in [14]; the main difference is that our indexing is shifted by one. In Figure 5 the commented statements are those found on page 178 of [14] for the BRGC. However, our derivation of Algorithm B is completely different from the one given in [14]; their derivation is based on a simulation of the stack arising from the recursive description of the BRGC, whereas ours is based on runs and seems to be much simpler.

Adapting Algorithm B to maintaining St in Algorithm L is fairly straightforward because when nodes are deleted from B they are all stuck and when they are added to B they are all free. In the "do nothing" cases of Algorithm L, we do a standard iteration of Algorithm B. In the other cases the assignment $run(head) \leftarrow head$ is done and *head* may have to be updated if it is removed from B.

The only other updating of *run* occurs when $Sibling(v) \cap F = \emptyset$ and v is not an only child. In that case let $r = next(q)$, where q points to the chosen node v, and let $prf = next(rchild(parent(q)))$. Recall that by Observation #4, v must be a leftmost sibling and all of its siblings are stuck. If $run(r) \neq prf$ (i.e., the node pointed to by *prf* is stuck), then we set $run(prf) \leftarrow run(r)$. Finally, we must set $run(r) \leftarrow r$ as an itialization for future iterations.

[Q3] To detect whether $Sibling(v) \cap F = \emptyset$ we could maintain a counter $count(x)$ at each node x, indicating the number of it's children that are currently in F; i.e., $count(x) = |Children(x) \cap F|$. Then the condition $Sibling(v) \cap F = \emptyset$ is equivalent to the condition $count(parent(v)) = 0$.

Since $0 \leq count(x) \leq |Children(x)|$, the counter can be replaced by a pointer $ptr(x)$ to a child of x. The pointer moves right when a child is deleted and left when a child is augmented. If $count(x) = 0$ then $ptr(x)$ is nil. This pointer is what is used in the Pascal implementation that may be obtained from the second author.

6 Concluding Remarks

We have developed a loopless algorithm for generating the ideals of a forest poset. This algorithm is a generalization of existing algorithms for generating the Binary Reflected Gray Code.

There are many interesting remaining questions. For example, if Algorithm P is restricted to output only ideals of some fixed size, say k, then the list of ideals produced has the property that successive ideals differ by the the addition of one element and the deletion of another. We hope that this paper will initiate further study of Hamiltonicity properties of ideal graphs.

If $\mathcal{P}$ is a chain of n elements, then Algorithm P takes time $\Omega(nN)$, where $N = i(\mathcal{P}) = n + 1$. Thus, in the worst case, it is not a constant amortized time algorithm. However, on randomly selected rooted trees (e.g., as generated On randomly selected rooted trees (e.g., as generated by the algorithm of Nijenhuis and Wilf [13]), Algorithm P appears to run in constant amortized time, and we conjecture that this is true in general.

References

[1] J. Bitner, G. Ehrlich, and E. Reingold, "Efficient Generation of the Binary Reflected Gray Code and its Applications," *Communications of the ACM*, 19 (1976) 517-521.

[2] T. Beyer and F. Ruskey, "Constant Average Time Generation of Subtrees of Bounded Size," unpublished manuscript, May, 1989.

[3] E.G. Coffman and P.J. Denning, *Operating Systems Theory*, Prentice-Hall, 1973 (pp. 94).

[4] N. Dershowitz, "A Simplified Loop-free Algorithm for Generating Permutations," *BIT*, 15 (1975) 158-164.

[5] G. Ehrlich, "Loopless Algorithms for Generating Permutations, Combinations, and Other Combinatorial Configurations," *JACM*, 20 (1973) 500-513.

[6] J. Fill and E.M. Reingold, *Solutions Manual for Combinatorial Algorithms: Theory and Practice*, Prentice-Hall, 1977.

[7] T. Hikita, "Listing and Counting Subtrees or Equal Size of a Binary Tree", Information Processing Letters, 17 (1983) 225-229.

[8] T.C. Hu and F. Ruskey, "Circular cuts in a network," *Mathematics of Operations Research*, 5 (1980) 422-434.

[9] R.E. Jamison, "On the Average Number of Nodes in a Subtree of a Tree", J. Combinatorial Theory, Series B, 35 (1983) 207-223.

[10] R.E. Jamison, "Monotonicity of the Mean Order of Subtrees", J. Combinatorial Theory, Series B, 37 (1984) 70-78.

[11] J.T. Joichi, D.E. White, and S.G. Willamson, "Combinatorial Gray Codes," *SIAM J. Computing*, 9 (1980) 130-141.

[12] A. Meir and J.W. Moon, "On Subtrees of Certain Families of Rooted Trees", *Ars Combinatoria*, 16-B (1983) 305-318.

[13] A. Nijenhuis and H. S. Wilf, *Combinatorial Algorithms*, Second Edition, Academic Press, 1978.

[14] Reingold, Nievergelt, and Deo, *Combinatorial Algorithms: Theory and Practice*, Prentice Hall, 1977.

[15] G.S. Rao, H.S. Stone and T.C. Hu, "Scheduling in a Distributed Processor System with Limited Memory," *IEEE Trans. Computers* C-28, pp 291-299.

[16] G. Steiner, "An Algorithm to Generate the Ideals of a Partial Order," *Operations Research Letters*, 5 (1986) 317-320.

[17] F. Ruskey, "Listing and Counting Subtrees of a Tree," *SIAM J. Computing*, 10 (1981) 141-150.

[18] F. Ruskey, "Generating Linear Extensions of Posets," *J. Combinatorial Theory (B)*, 54 (1992) 77-101.

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

[20] D. Stanton and D. White, *Constructive Combinatorics*, Springer, 1986.

[21] R.E. Tarjan, "A class of algorithms which require nonlinear time to maintain disjoint sets," *Journal of Computer and Systems Sciences*, 18 (1979) 110-127.

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

7 Appendix

This appendix is not intended for publication.

In this appendix we present a loopless Pascal implementation of Algorithm L. Input consists of n the number of nodes in the forest, followed by the parent array, with a 0 entry for a root node. For example, the tree of Figure 3 is input on two lines, with 6 on the first line and 0 1 2 2 1 5 on the second line.

```
program Gray_Forest ( input, output );

const MAX = 50;

type
  node_ptr = ^node;
  node = record
           (* static fields *)
           parent : node_ptr;
           rsib, lsib : node_ptr;
           rchild, lchild : node_ptr;
           (* dynamic fields *)
           next, back, run, ptr : node_ptr;
           inF : boolean;
         end;

var q,r   : node_ptr;
    head  : node_ptr; {head of boundary list}
    root  : node_ptr; {static}
    count : integer;  {optional}

(*********************** Input *********************)
procedure Input;
var
  i, p, n : integer;
  nd : array [0..MAX] of node_ptr;
begin
  readln( n );
  new( nd[0] );
  with nd[0]^ do begin
      lchild := nil;  rchild := nil;
```

```
  end;
  for i := 1 to n do begin
      read( p );  new( nd[i] );
      with nd[i]^ do begin
          parent := nd[p];
          if parent^.lchild = nil then parent^.lchild := nd[i]
          else begin
              parent^.rchild^.rsib  := nd[i];
              parent^.rchild^.next:= nd[i];
          end;
          lsib := parent^.rchild;
          parent^.rchild := nd[i];
          rsib := nil;
          lchild := nil;  rchild := nil;
          next := rsib;  back := lsib;
          inF := false;  ptr := nil;
          run := nd[i];
      end;
  end;
  root := nd[0];  head := nd[0]^.lchild;
  {Initialize dummy node}
  nd[0]^.rchild^.next := nd[0];
  nd[0]^.run := nd[0];  nd[0]^.next := nil;  nd[0]^.ptr := nil;
end {of Input};

(********************** PrintIt ********************)
procedure PrintIt;
{details left to reader}
end {of PrintIt};

(*********************** Join *********************)
procedure join( x,y : node_ptr );
begin
  x^.next := y;  y^.back := x;
end {of join};

(********************** DELETE ********************)
procedure Delete( q,r : node_ptr );
var par, prf : node_ptr;
begin
  par := q^.parent;
  if par^.ptr = par^.lchild then begin
      par^.ptr := nil;
      prf := par^.rchild^.next;
      if head = q then head := par
```

```
      else join( q^.back, par );
      join( par, prf );
      if par^.rchild <> par^.lchild then begin
          if r^.run <> prf then prf^.run := r^.run;
          r^.run := r;
      end;
  end else {a sibling is still in F} begin
      q^.run := r^.run;  r^.run := r;  {standard update}
      par^.ptr := par^.ptr^.lsib;  {decrement ptr}
  end;
end {of Delete};

(********************** AUGMENT **********************)
procedure Augment( q,r : node_ptr );
var par : node_ptr;
begin
  par := q^.parent;
  if par^.ptr = nil  {increment ptr}
      then par^.ptr := par^.lchild
      else par^.ptr := par^.ptr^.rsib;
  if q^.lchild = nil {no children} then begin
      q^.run := r^.run;  r^.run := r;  {standard update}
  end else {at least one child} begin
      if head = q then head := q^.lchild
      else join( q^.back, q^.lchild );
      join( q^.rchild, r );
  end;
end {of Augment};

(*********************** main ***********************)
begin
  Input;

  PrintIt;
  q := head^.run;  r := q^.next;
  repeat
     head^.run := head;
     if q^.inF then Delete( q,r ) else Augment( q,r );
     q^.inF := not q^.inF;
     PrintIt;
     q := head^.run;  r := q^.next;
  until q = root;

end.
```