RELATÓRIO TÉCNICO

TWO PROBLEMS ON THE GENERATION OF LINEAR EXTENSIONS OF POSETS

Andréa Werneck Richa
NCE/UFRJ

Jayme Luiz Szwarfciter
NCE/UFRJ

NCE — 07/92
a g o s t o

Núcleo de Computação Eletrônica
Universidade Federal do Rio de Janeiro

Tel.: 598-3212 - Fax.: (021) 270-8554
Caixa Postal 2324 - CEP 20001-970
Rio de Janeiro - RJ

Two Problems On the Generation of Linear Extensions of Posets

Andréa Werneck Richa Jayme Luiz Szwarcfiter

Julho de 1992

Resumo

Apresentamos dois resultados sobre a geração de extensões lineares de um poset. Primeiro provamos que as extensões lineares de todo poset podem ser geradas por inserção. A seguir, descrevemos um algoritmo de tempo médio constante para gerar as extensões lineares de um poset série-paralelo em ordem lexicográfica.

Abstract

We describe two results on the generation of linear extensions of a poset. First, we prove that the linear extensions of every poset can be generated by insertion. Next, we describe a constant average time algorithm to generate the linear extensions of a series-parallel poset in lexicographic order.

1 Introduction

Given a set of combinatorial objects, a natural question is whether or not they can be generated following some kind of sistematical criteria. In addition, the interest is to generate them efficiently. For instance, given a poset one could ask whether its linear extensions can be generated by transposition or by insertion. That is, if there is a total ordering of its extensions such that two consecutive extensions differ by a single transposition or by a single insertion of its elements, respectively. It is known [11, 8] that the former kind of generation is not possible in general, and an open question is to characterize the posets that admit it. In contrast, it is shown in Section 2 that the linear extensions of every poset can be generated by insertion.

We say that a generation algorithm runs in *constant average time* if it requires $O(N)$ time, where N is the number of objects being generated. Generating the linear extensions of a poset can be done using the algorithms in [16, 4, 5, 15]. Pruesse & Ruskey [7], however, answered the question in the best possible way, by describing a constant average time algorithm for it. This result is also interesting in view of the fact that the corresponding enumeration problem has been proved to be $\#P$-complete (Brightwell & Winkler [1]). In Section 3, we describe a constant average time algorithm for (optimumly) generating the linear extensions of a series-parallel poset in lexicographic order. Except for the case of irrestricted permutations [9], we do not know other classes of posets whose linear extensions have been generated in lexicographic order within constant average time.

A *partially ordered set* (*poset*) $\mathcal{P}(S, R)$ is a reflexive, transitive and antisymmetric binary relation R on a set S. Denote $n = |S|$. An ordered pair $(a, b) \in R$ is denoted $a \preceq_{\mathcal{P}} b$. or simply $a \preceq b$. By $a \prec b$ we mean $a \preceq b$ and $a \neq b$. An element $a \in S$ is *minimal* (*maximal*) in $\mathcal{P}$ if there is no element $b \in S$ such that $b \prec a$ ($a \prec b$).

If neither $a \preceq b$ nor $b \preceq a$ then a and b are said to be *incomparable*. If no pair of elements of S is incomparable, then $\mathcal{P}$ is a *total ordering* (or *linear ordering* or *chain*). $\mathcal{C}_k$ denotes a chain of length (number of elements) k. An *extension* of $\mathcal{P}(S, R)$ is a poset $\mathcal{Q}(S, R')$ such that $R \subseteq R'$. An extension of $\mathcal{P}$ that is a total ordering $x_1 x_2 \ldots x_n$ is a *linear extension* of $\mathcal{P}$. Let $E(\mathcal{P})$ and $e(\mathcal{P})$ denote the set and the number of linear extensions of $\mathcal{P}$ respectively.

Let $\mathcal{P}(S, R)$ be a poset. A *subposet* of $\mathcal{P}$ is a poset $\mathcal{P}'(S', R')$ such that $S' \subseteq S$ and $R' = R \cap S' \times S'$. We also use $\mathcal{P}_{S'}$ to denote $\mathcal{P}'$. A linear extension of a subposet of $\mathcal{P}$ is a *partial extension* of $\mathcal{P}$. Let $\mathcal{Q}(S', R')$ be a poset. By $\mathcal{P} - \mathcal{Q}$ (or $\mathcal{P} - S'$) we denote the subposet $\mathcal{P}_{S \setminus S'}$. The *dual* of a poset $\mathcal{P}(S, R)$ is the poset $\mathcal{P}'(S, R')$. where $R' = \{(x, y) \mid (y, x) \in R\}$.

A *left* (*right*) *insertion* is an operation that moves an element x from the i-th to the j-th position in a linear extension, $i > j$ ($j > i$). In general, call them a $|j - i|$-insertion, or simply an insertion. A generation of the linear extensions of a poset such that every pair of successive extensions differ by a single insertion is said to be a *generation by insertion*.

2 Generation by Insertion

In this section. we prove that every poset has a generation of its linear extensions by left or by right insertion.

Theorem 1 *The linear extensions of every poset can be generated by left (right) insertion.*

Proof : The proof follows by induction. We first consider the generation by left insertion.

Let $\mathcal{P}(S, R)$ be a poset with n elements. Let $M = \{m_1, \ldots, m_k\}$, $k \geq 1$, denote the set of minimal elements of $\mathcal{P}$. The subposet $\mathcal{P}_M$ has a generation by left insertion : apply the Steinhaus-Johnson-Trotter [12, 3, 13] algorithm to generate non-restricted permutations of M by (left) 1-insertions. If $S = M$ the theorem is proved.

Otherwise, the induction hypothesis states that any poset with $n' < n$ elements has a generation by left insertion. Let x be a maximal element in $S \backslash M$ and $\mathcal{Q}$ be the poset $\mathcal{P} - \{x\}$. Let $\Gamma = \ell_1, \ldots, \ell_{e(\mathcal{Q})}$ be a generation by left insertion of $E(\mathcal{Q})$. We start with $\ell_1 = p_1 p_2 \ldots p_{n-1}$ and add x to this extension. Let p_s be the rightmost element of ℓ_1 such that $p_s \prec_{\mathcal{P}} x$, $1 \leq s < n$. Element p_s necessarily exists, otherwise $S = M$ and the theorem is proved. Generate $n-s$ linear extensions of $\mathcal{P}$ by (left) 1-insertions, successively "sweeping" x one position to the left. That is, x occupies respectively positions $n, n-1, \ldots, s+1$. See Figure 1.

Let ℓ_r, $1 < r \leq e(\mathcal{Q})$, be the next extension to be considered in Γ (all ℓ_i, $1 \leq i < r$, have already been considered).

(a) If the last extension generated from $\ell_{r-1} = p_1 p_2 \ldots p_{n-1}$ was $p_1 \ldots p_{n-1} x$, then the generation from $\ell_r = q_1 q_2 \ldots q_{n-1}$ is as depicted in Figure 1. As ℓ_{r-1} and ℓ_r differed by a left insertion in Γ, so do $p_1 \ldots p_{n-1} x$ and $q_1 \ldots q_{n-1} x$.

(b) If the last extension generated from $\ell_{r-1} = p_1 \ldots p_{n-1}$ was $p_1 \ldots p_s x \ldots p_{n-1}$, $1 \leq s < n-1$, then we can rewrite ℓ_{r-1} as $p_1 \ldots p_j \ldots p_{i-1} p_i p_{i+1} \ldots p_{n-1}$ and ℓ_r as $p_1 \ldots p_i p_j \ldots p_{i-1} p_{i+1} \ldots p_{n-1}$. That is, ℓ_r has been obtained from ℓ_{r-1} by a left $(i-j)$-insertion. Let p_k be the rightmost element $\prec x$ in ℓ_r. Proceed according to the value of s :

- If $s = i$ then $p_1 \ldots p_{i-1} p_i x p_{i+1} \ldots p_{n-1}$, $1 \leq j < i < n-1$, was the last extension generated from ℓ_{r-1}. We can add x to ℓ_r in the following way : begin inserting x between p_{i-1} and p_{i+1}; sweep x to the right using (left) 1-insertions of the element immediately at the right of x; then insert x immediately before p_{i-1}; sweep x to the left using 1-insertions of x until we reach p_k. This procedure is depicted in Figure 2. If $p_{i-1} \prec x$ then $k = i-1$ and we stop at $\dagger$.

- If $s \neq i$ then $k = s$ necessarily. Extension $p_1 \ldots p_j \ldots p_s x \ldots p_i \ldots p_{n-1}$, $1 \leq s < n-1$ and $1 \leq j < i < n$, was the last one obtained from ℓ_{r-1}. Insert p_i immediately before p_j and sweep x to the rightmost end of ℓ_r, using 1-insertions, as in Figure 3 (we assumed that $s \geq j$ without loss of generality).

The generation by right insertion can be obtained from the generation by left insertion of the corresponding dual poset. □

3 Lexicographic Order Generation for Series-Parallel Posets

We will now present a constant average time algorithm for generating the linear extensions of a series-parallel poset in lexicographic order.

3.1 Definitions and Notation

Series-Parallel Posets Series-parallel posets are defined in terms of a minimal series-parallel digraph.

Definition 1

(i) *The digraph with a single vertice and no edges is* minimal series-parallel (MSP);

(ii) *If* $D'(V', E')$ *and* $D''(V'', E'')$ *are two MSP digraphs, so are both digraphs constructed by the following operations :*

(a) Parallel Composition : $D(V' \cup V'', E' \cup E'')$,

(b) Series Composition : $D(V' \cup V'', E' \cup E'' \cup [Max(D') \times Min(D'')])$, *where* $Max(D')$ *is the set of sinks of* D' *and* $Min(D'')$ *the set of sources of* D''.

An MSP digraph may be represented using its *binary decomposition tree* : a strictly binary tree where each leaf is a vertex of the digraph and each internal node represents a series or a parallel composition of the MSP digraphs defined by the subtrees rooted at its children.

Definition 2 *A* poset $\mathcal{P}(S, R)$ *is* series-parallel *if and only if the transitive reduction of* R *on* S *is MSP.*

In Valdes, Tarjan & Lawler [14], we can find a linear time algorithm for recognizing posets of this class which also finds an associated binary decomposition tree, if this is the case. See also [2] for a description of the series-parallel class.

Let S be a set and $\prec'$ a total ordering of S; $S_1 = x_1, x_2, \ldots, x_p$ and $S_2 = y_1, y_2, \ldots, y_q$ two sequences of elements of S. S_1 will be *lexicographically smaller* than S_2 according to $\prec'$ when :

- For some j, $1 \leq j \leq \min\{p, q\}$, $x_j \prec' y_j$ and for all k, $1 \leq k < j$, $x_k = y_k$. Or, alternatively,

- $p < q$ and for all k, $1 \leq k \leq p$, $x_k = y_k$.

The sequences $S_1, \ldots, S_n$ are in *lexicographic order* if and only if $S_i \prec' S_j \Rightarrow i < j$.

Let $T(V, E)$ be an ordered rooted tree and $x, y \in V$, such that x is the father of y in T. *Collapsing* vertices x and y results in the ordered rooted tree $T'(V', E')$ such that $V' = V \backslash \{y\}$, $E' = E \backslash \{(y, z) \in E\} \cup \{(x, z) \mid (y, z) \in E\}$. The ordering of the children of x in T' is the same as in T, except that y is replaced by its children in the ordering they appeared in T.

Let $\{1, \ldots, n-1\}$ be the internal nodes of a binary decomposition tree $T(\mathcal{P})$, $\mathcal{P}$ series-parallel. The *collapsed decomposition tree* $T_C(\mathcal{P})$ is the one obtained from T by succesively collapsing pairs of adjacent series nodes, as long as possible. $T_C(\mathcal{P})$, or simply T_C, is not necessarily binary. However, all parallel nodes in T_C have exactly two children. Denote n_C the number of internal nodes of $T_C(\mathcal{P})$.

Let ${T_C}^i$ denote the subtree of T_C with root i, i a node of T_C. We will write indistinctly subposet with collapsed subtree ${T_C}^k$ or subposet with root k. By traversing a tree in *post-order* we mean : "Visit the leftmost subtree of the root in post-order: Visit the second leftmost subtree of the root in post-order; ...; Visit the rightmost subtree of the root in post-order; Visit the root".

3.2 The Algorithm

The main idea of the following algorithm is to use the collapsed decomposition tree of a series-parallel poset to systematically generate its linear extensions.

The internal nodes of T_C are traversed and numbered in post-order. The analysis of each internal node x is such that x is considered only after all its children have been analysed. During the generation process we will be doing recursive calls at the internal nodes of T_C, where each internal node can only recursively call its post-order successor. Each recursive call k always follows the generation of a new partial extension of the subposet at $k-1$, predecessor of k in post-order. When visiting node k, we look at the partial extension most recently generated at each of the subtrees rooted at its children and combine these extensions according to the nature of k. That is, if k is a series node then we simply concatenate the partial extensions of its children, from left to right. If k is parallel, then we must merge the partial extensions of the left and right children of this node. Denote by l and r the children of k and n_i the number of elements of the subposet ${T_C}^i$. Merging two partial extensions of sizes n_l and n_r at node k corresponds to the generation of the n_l(or n_r)-combinations of n_k. Thus, to perform a parallel composition, use an algorithm to generate combinations in lexicographic order [6, 9]. Series compositions have no inffluence in this ordering. Return from a recursive call when all possible combinations at the corresponding node have been exhausted.

By $ext(k)$ is denoted the most recently generated partial extension relative to ${T_C}^k$, if k is an internal node of T_C. Else k is a leaf of T_C and $ext(k)$ returns the label of k. By "||" is meant the concatenation operation of two sequences of elements.

algorithm GenerateExtSeriesParallel
data Series-parallel poset $\mathcal{P}(S, R)$ with decomposition tree $T(\mathcal{P})$, $n = |S|$
Construct the collapsed decomposition tree $T_C(\mathcal{P})$, collapsing the adjacent series nodes of T.
n_C = number of internal nodes in T_C;
Swap. in T_C, the left subtree with the right subtree of a parallel node, every time the subtree at the left contains more elements (leaves) than the one at the right ;
Number the internal nodes of T_C in post-order and label its leaves $1, \ldots, n$, from left to right.
Compute $n_k = \sum n_i$, i a son of k, $1 \leq k \leq n_C$, the number of elements of $T_C{}^k$;
Generation(1)
end algorithm

procedure *Generation*(k)
  if k is a series node **then**
    Series(k)
  else
    Parallel(k)
end procedure

procedure *Series*(k)
  if $k = n_C$ **then**
    $ext(k)$[= ext(leftmost son of k)||ext(2^{nd} leftmost son of k)||...||ext(rightmost son of k)] is a new linear extension of $\mathcal{P}$
  else
    Generation($k+1$)
end procedure

procedure *Parallel*(k)
  Let l and r be the post-order numbering or the corresponding leaf label, if this is the case. of the left and right children of k respectively;
  $c := \{1, \ldots, n_l\}$ (= first n_l- combination in lexicographical order);
  $ext(k) := ext(l)||ext(r)$
  if $k = n_C$ **then**
    $ext(k)$ is a new linear extension of $\mathcal{P}$
  else
    Generation($k+1$)
  while c is not the last n_l-combination in the lexicographic ordering **do**
    $c :=$ n_l-combination succeding c in lexicographic order;
    $ext(k)$:=merging of $ext(l)$ and $ext(r)$, such that c corresponds to the positions occupied by the elements of $ext(l)$ in the merged sequence;
  if $k = n_C$ **then**
    $ext(k)$ is a new linear extension of $\mathcal{P}$
  else
    Generation($k+1$)
end procedure

The *canonical extension* is the one firstly generated by the algorithm, that is, $12\ldots n$. The lexicographic ordering obtained by the algorithm is relative to the canonical extension.

Let $\mathcal{P}$ be a poset consisting of two disjoint chains $\mathcal{C}_1$ and $\mathcal{C}_2$, of sizes r and $t-r$ respectively. The relation between the r-combinations of t integers and the set $E(\mathcal{P})$ is that there exists an one-to-one function such that every combination $\{a_1,\ldots,a_r\}$, $a_i < a_j$ iff $i < j$, leads to an extension $\in E(\mathcal{P})$ where a_i, $1 \leq i \leq r$, is the position of the i-th element of $\mathcal{C}_1$ in this extension.

Since $n_l \leq n_r$, it follows that $r \leq t/2$. In this case, we need in the average less than 2 steps per r-combination generated [6], as well as for finding its corresponding linear extension. For each new element (not in the previous r-combination) elected to be in an r-combination, one single insertion is needed. All these operations can be performed in constant time.

The implementation employs a doubly linked list with the n elements of $\mathcal{P}$, which keeps track of the most recently generated partial extensions, as in Figure 5. Let $\mathcal{P}_{S_1},\ldots,\mathcal{P}_{S_r}$ be the maximal subposets being analyzed from left to right in T_C at a given time, $\cup S_i = S$. Then for all S_i, S_j, $i < j$, S_i is totally to the left of S_j in the doubly linked list (see Figure 5). If we always restore the pointers between the last element of a maximal partial extension generated and the first one of the following maximal partial extension and if we initialize the doubly linked list with the canonical extension $12\ldots n$, then we do not need to perform any concatenation operation at a series node. This will be crucial to our complexity analysis, as we do not know a priori the number of partial extensions to be concatenated at a series node. In Richa [10], we find a more detailed version of the algorithm.

3.3 Complexity Analysis

The algorithm needs space $O(n^2)$ in the worst case for keeping track of the combinations being generated at every parallel node called recursively.

Recognizing series-parallel posets and constructing the decomposition tree can be done in $O(n+m)$ time, $m = |R|$ [14]. For collapsing the decomposition tree, post-order numbering, building the canonical extension and computing all n_k, we need $O(n)$ time.

The total time spent during the generation of the linear extensions of $\mathcal{P}$ can be expressed as

$$\textit{generation time}\,(\mathcal{P}) \leq c_1\, NRC(T_C(\mathcal{P})) + c_2\, e(\mathcal{P}) \qquad (1)$$

where $NRC(T_C(\mathcal{P}))$ is the number of recursive calls *Generation*(k) performed; c_1, c_2 are constants. The term $c_2\, e(\mathcal{P})$ corresponds to the number of steps consumed for combining the partial extensions of the children of the root node. That is, if the root is a series node, $e(\mathcal{P})$ is equal to the number of calls at the root, and we take constant time at each of them. If the root is parallel, then we would take $\leq 2e(\mathcal{P})$ time to generate all the combinations relative to this node.

The time consumed by the generation of the strictly partial extensions is expressed by $c_1\, NRC(T_C(\mathcal{P}))$, as for every such extension with root at node k there corresponds

exactly one call to *Generation*$(k+1)$, $1 \leq k < n_C$. We take less than 2 steps per partial extension generated.

It remains to prove that $NRC(T_C(\mathcal{P})) = O(e(\mathcal{P}))$, in order to have a constant average time algorithm. The proof is by induction on the number of internal nodes of T_C, considered in reverse post-order (call it *inorder*). The induction hypothesis is as follows :

$$NRC(T_C(\mathcal{P}')) \leq 2\,e(\mathcal{P}') - 1, \tag{2}$$

where $T_C(\mathcal{P}')$ has $n_C' < n_C$ internal nodes. The posets with a single internal node only satisfy equation (2).

Let $T_C(\mathcal{P})$ be a collapsed tree obtained from $T_C(\mathcal{P}')$ by replacing a convenient leaf x by a new internal node α whose children (leaves) are the elements of $\mathcal{P} - \mathcal{P}'$. The leaf x is chosen so that α is the first internal node of $T_C(\mathcal{P})$ in post-order. The choice of x is not unique, as we can see in the example of Figure 6.

The following can occur :

(a) α is a parallel node. Then :

$$e(\mathcal{P}) \geq 2\,e(\mathcal{P}') \tag{3}$$

$$NRC(T_C(\mathcal{P})) = 2\,NRC(T_C(\mathcal{P}')) + 1 \tag{4}$$

It is easy to conclude that, when the children (elements of $\mathcal{P}$) of the parallel node α are composed in series with all other elements of the poset (Figure 7), $e(\mathcal{P}) = 2\,e(\mathcal{P}')$. In this case, for every extension $\beta x\gamma \in E(\mathcal{P}')$, $\beta\gamma \in E(\mathcal{P}' - \{x\})$, there are exactly $\beta yz\gamma$ and $\beta zy\gamma \in E(\mathcal{P})$, where y, z are the elements of $\mathcal{P} - \mathcal{P}'$ added together with α to $\mathcal{P}'$. Equality in (4) is trivially verified. Node α is the first one to be visited in post-order and contributes with one recursive call. For each of the two partial extensions generated at α (yz and zy), we will "complete" them by generating (and combining) all the linear extensions of $\mathcal{P}' - \{x\}$. This will be done calling the internal nodes of $T_C(\mathcal{P}')$ recursivelly, in the same way we did to generate $E(\mathcal{P}')$.

It follows from (2) and (4),

$$NRC(T_C(\mathcal{P})) \leq 2[2e(\mathcal{P}') - 1] + 1 = 4\,e(\mathcal{P}') - 1$$

The proof is completed by applying (3) to the above inequality.

(b) α is a series node. Then :

$$e(\mathcal{P}) \geq \frac{3}{2} e(\mathcal{P}') \tag{5}$$

$$NRC(T_C(\mathcal{P})) = NRC(T_C(\mathcal{P}')) + 1 \tag{6}$$

Equation (6) is similar to (4), but now only one partial extension is generated at α. On the other hand, equation (5) needs a more careful analysis. $e(\mathcal{P})/e(\mathcal{P}')$ is minimum when α and its father β satisfy : *(i)* α has exactly two children; *(ii)* the other child of β is a leaf; and

(iii) β is composed in series with the remaining elements of the poset. (See Figure 8). two children. Then for each pair $\gamma x z \delta, \gamma z x \delta \in E(\mathcal{P}')$, we have $\gamma w y z \delta, \gamma w z y \delta, \gamma z w y \delta \in E(\mathcal{P})$, w, y being the elements added to $\mathcal{P}'$ together with α. Node α was inserted as the right son of β in Figure 8 without loss of generality.

Finally, from (2) and (6),

$$NRC(T_C(\mathcal{P})) \leq 2\, e(\mathcal{P}') \tag{7}$$

Since $e(\mathcal{P}') \geq 1$,

$$e(\mathcal{P}') \leq e(\mathcal{P}) - \frac{1}{2}$$

By replacing in (7), we have proved the validity of the induction hypothesis for $\mathcal{P}$.

4 Conclusion

We have presented a constant average time algorithm for the lexicographic order generation of the linear extensions of a series-parallel poset. Besides it was proved that every poset has a generation by insertion. It would be interesting to verify the following generalizations of the two problems above :

1. a constant average time algorithm for the lexicographic order generation of a *general* poset;

2. a *constant average time algorithm* for the generation by insertion.

References

[1] G. Brightwell and P. Winkler. Counting linear extensions is #P-complete. Technical Report TR 90-49, DIMACS, 1990.

[2] M. Habib and R.H. Mohring. On some complexity properties of n-free posets and posets with bounded decomposition diameter. *Discrete Math.*, 63:157–182, 1987.

[3] S.M. Johnson. Generation of permutations by adjacent transposition. *Math. Comp.*, 17:282–285, 1963.

[4] A.D. Kalvin and Y.L. Varol. On the generation of all topological sortings. *J. Algorithms*, 4:150–162, 1983.

[5] D.E. Knuth and J.L. Szwarcfiter. A structured program to generate all topological sorting arrangements. *Inf. Proc. Letters*, 2:153–157, 1974.

[6] A. Nijenhius and S. Wilf. *Combinatorial Algorithms.* Academic Press. New York, 1975.

[7] G. Pruesse and F. Ruskey. Generating linear extensions fast. to appear. SIAM J. Comput.

[8] G. Pruesse and F. Ruskey. Generating the linear extensions of certain posets by transposition. *SIAM J. Discrete Math.*, 4:413–422, 1991.

[9] E.M. Reignold, J. Nievergelt, and N. Deo. *Combinatorial Algorithms : Theory and Practice.* Prentice-Hall, Englewood Cliffs, 1977.

[10] A.W. Richa. Geração e enumeração de extensões lineares de conjuntos parcialmente ordenados. Master's thesis, UFRJ, Rio de Janeiro, Brazil, 1992.

[11] F. Ruskey. Generating linear extensions of posets by transpositions. *J. Combinatorial Theory (B)*, 1991.

[12] H. Steinhaus. *One Hundred Problems in Elementary Mathmatics.* Basic, 1964.

[13] H.F. Trotter. Algorithm 115 : Perm. *Comm. ACM*, 5:434–435, 1962.

[14] J. Valdes, R.E. Tarjan, and E.L. Lawler. The recognition of series parallel digraphs. *ACM*, 1979.

[15] Y.L. Varol and D. Rotem. An algorithm to generate all topological sorting arrangements. *Comput. J.*, 24:83–84, 1981.

[16] M.B. Wells. *Elements of Combinatorial Computing.* Pergamon Press, Elmsford, 1971.

$$p_1 \cdots p_s \cdots p_{n-2} p_{n-1} x$$
$$p_1 \cdots p_s \cdots p_{n-2} x p_{n-1}$$

$$p_1 \cdots p_s p_{s+1} x \cdots p_{n-2} p_{n-1}$$
$$p_1 \cdots p_s x p_{s+1} \cdots p_{n-2} p_{n-1}$$

Figure 1: Adding the maximal element x to an extension $p_1 \ldots p_{n-1}$ using (left) 1-insertions of x only. p_s is the rightmost element $\prec x$ of this extension.

$$\begin{array}{rl}
p_1 \cdots p_i p_j \cdots p_k \cdots p_{i-1} x p_{i+1} \cdots p_{n-1} & \\
p_1 \cdots p_i p_j \cdots p_k \cdots p_{i-1} p_{i+1} x \cdots p_{n-1} & \\
\vdots & \\
p_1 \cdots p_i p_j \cdots p_k \cdots p_{i-1} p_{i+1} \cdots p_{n-1} x & \dagger \\
p_1 \cdots p_i p_j \cdots p_k \cdots x p_{i-1} p_{i+1} \cdots p_{n-1} & \\
p_1 \cdots p_i p_j \cdots p_k \cdots x p_{i-2} p_{i-1} p_{i+1} \cdots p_{n-1} & \\
\vdots & \\
p_1 \cdots p_i p_j \cdots p_k x \cdots p_{i-1} p_{i+1} \cdots p_{n-1} &
\end{array}$$

Figure 2: $s = i$.

$$\begin{array}{c}
p_1 \cdots p_i p_j \cdots p_s x \cdots p_{n-1} \\
p_1 \cdots p_i p_j \cdots p_s p_{s+1} x \cdots p_{n-1} \\
\vdots \\
p_1 \cdots p_i p_j \cdots p_s \cdots x p_{n-1} \\
p_1 \cdots p_i p_j \cdots p_s \cdots p_{n-1} x
\end{array}$$

Figure 3: $s \neq i$.

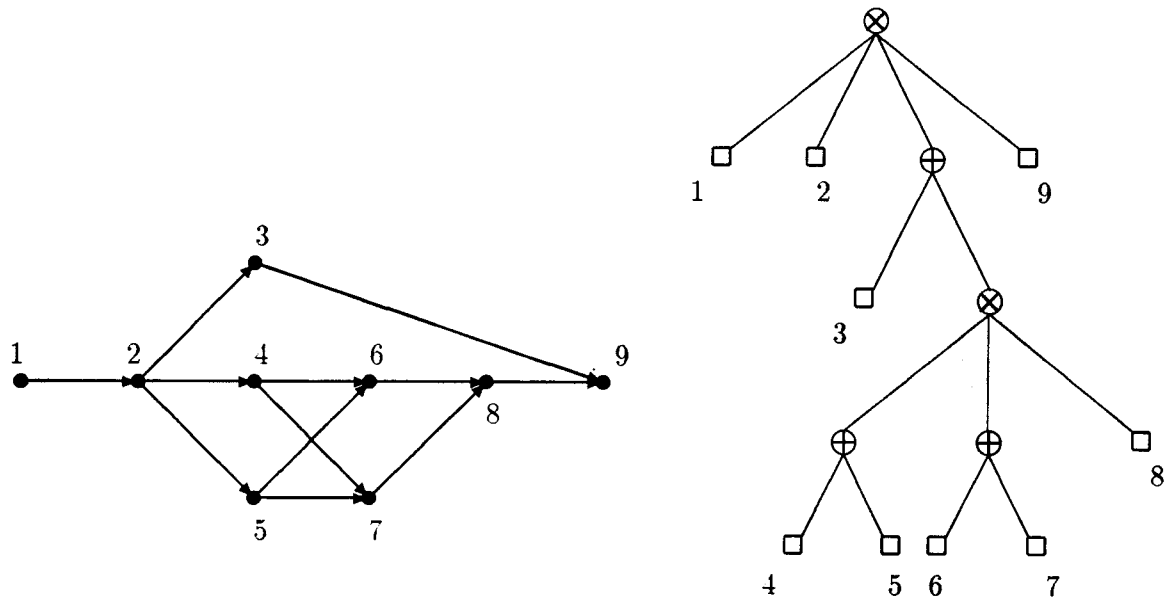


Figure 4: A series-parallel poset and its associated collapsed decomposition tree. Symbol $\otimes$ denotes a series node and symbol $\oplus$ a parallel node.

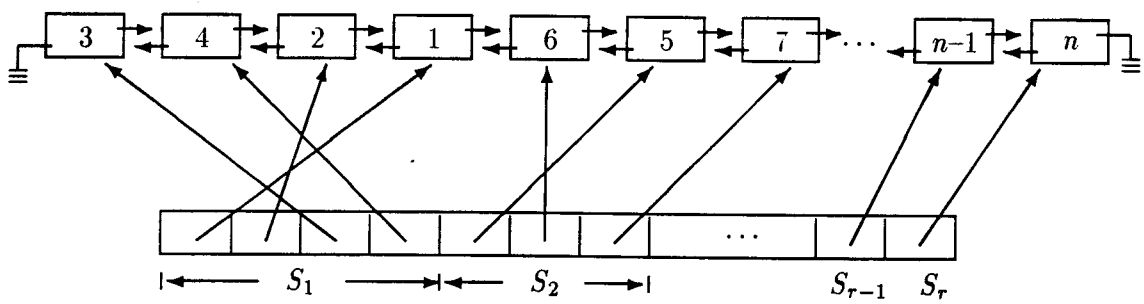


Figure 5: Data structure employed by the algorithm : a vector with n pointers to the n elements in the doubly linked list.

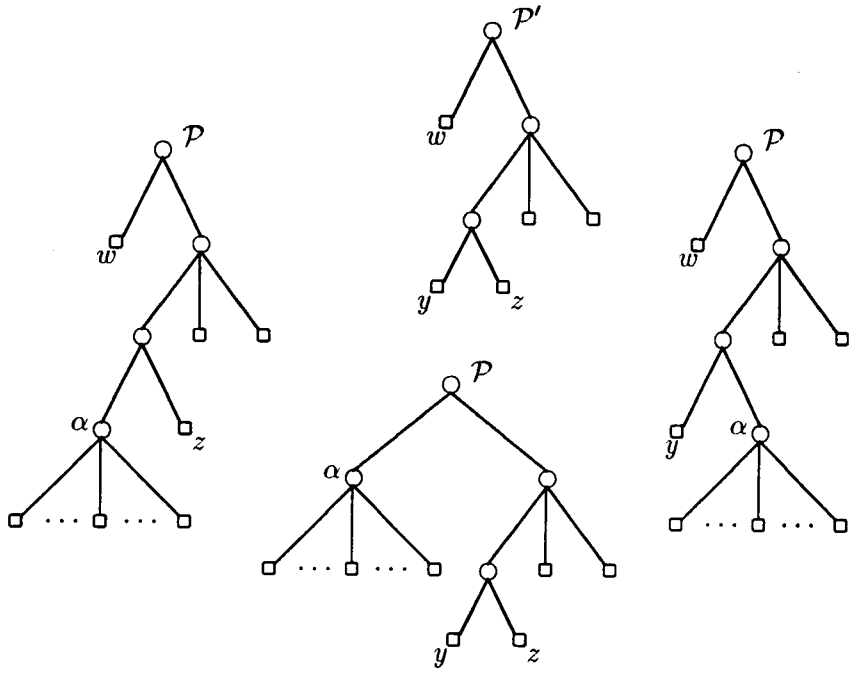


Figure 6: α can be inserted in exactly 3 different positions at $T_C(\mathcal{P}')$.

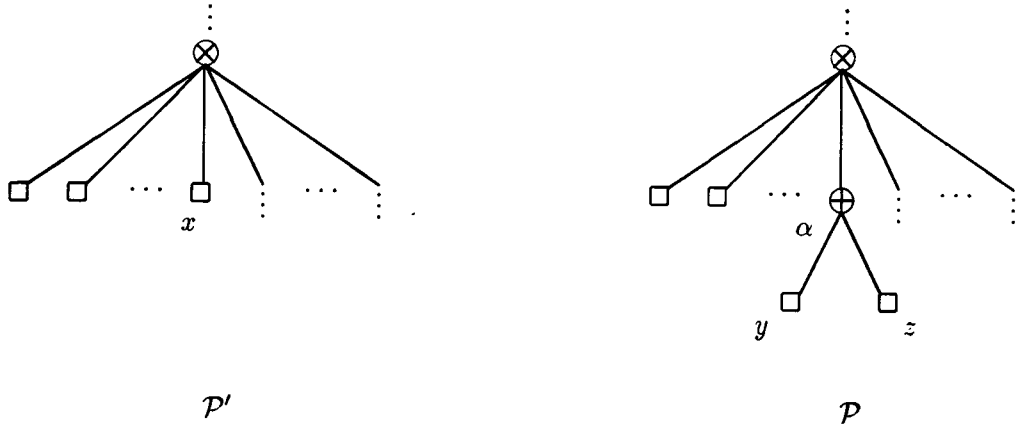
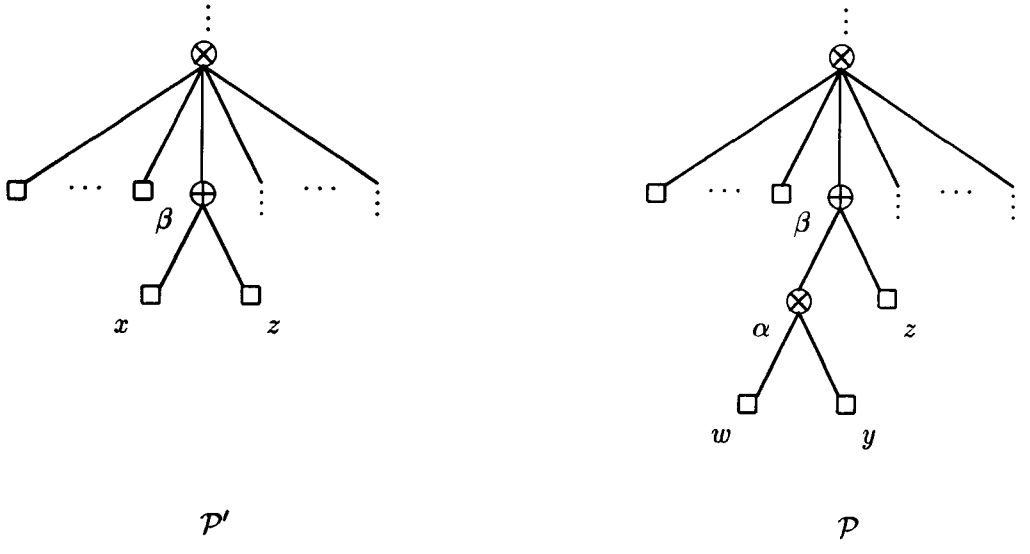


Figure 7: α is a parallel node.



$\mathcal{P}'$ $\mathcal{P}$

Figure 8: α is a series node.