Computing a Poset from its Realizer *

Enrico Nardelli[1,2] **Vincenzo Mastrobuoni**[1]
Alesiano Santomo[1]

1. Dipartimento di Matematica Pura ed Applicata, Univ. of L'Aquila, Via Vetoio, Coppito, I-67010 L'Aquila, Italia. E-mail: nardelli@univaq.it
2. Istituto di Analisi dei Sistemi ed Informatica, Consiglio Nazionale delle Ricerche, Viale Manzoni 30, I-00185 Roma, Italia.

Printed on: July 7, 1997

Abstract

In this paper we provide an efficient algorithm for computing the graph representing a d-dimensional poset which is given by means of the d linear extensions realizing it.

Keywords: partially ordered sets, algorithms, data structures.

1 Introduction

Let $P = (X, \leq_P)$ be a partially ordered set (*poset*) defined on the ground set X by means of the partial order relation $\leq_P$. Let $\mathcal{L} = (X, \leq_{\mathcal{L}})$ be a poset defined on the same ground set but such that $\leq_{\mathcal{L}}$ is a total order relation and $x \leq_P y \Rightarrow x \leq_{\mathcal{L}} y$. Poset $\mathcal{L}$ is said to be a *linear extension* of P.

A set $\{\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_k\}$ of linear extensions of a poset P is said to be a *realizer* of P if $x \leq_P y \Rightarrow x \leq_{\mathcal{L}_i} y, \forall i = 1, 2, \ldots, k$ and $x||y$[1] $\Rightarrow \exists i, j$ with $i \neq j$ such that $x \leq_{\mathcal{L}_i} y$ and $y \leq_{\mathcal{L}_j} x$.

The minimum k such that a set $\{\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_k\}$ of linear extensions of a poset P is a realizer of P is said to be the *linear dimension* of P.

Let poset $P = (X, \leq_P)$ of linear dimension d be given by means of d linear extensions realizing it. An algorithmic problem posed by Spinrad [9] is how to

*Research partially supported by the "Algoritmi, Modelli di Calcolo e Strutture Informative" 40%-Project of the italian Ministry for University and Scientific & Technological Research (MURST)

[1] $x||y$ means that neither $x \leq_P y$ nor $y \leq_P x$

efficiently build the adjacency lists of the directed graph $G_P = (X, E)$ representing P, i.e. such that $(x, y) \in E \Leftrightarrow x \leq_P y$. Sometimes G_P is also called the *transitive closure representation* of P.

When $d = 2$ McConnell and Spinrad [4], as part of a recognition algorithm for 2-dimensional partial orders, give an optimal algorithm which runs in worst-case $O(m + n)$ time, using worst-case $O(n)$ space, where $n = |X|$ and $m = |E|$. From now on, when speaking of 'time' we always mean 'worst-case time' and with 'space' we mean 'worst-case space'.

For the special cases $d = 3$ and $d = 4$, using ad-hoc techniques developed for the rectangle enclosure problem [2], the transitive closure representation of the poset can be built, respectively, in $O(m + n \log n)$ time using $O(n)$ space and in $O(m + n \log^2 n)$ time[2] using $O(n)$ space.

When $d \geq 3$ Spinrad, using the *bridged range tree* of Willard [10] and Lueker [3], provides in [9] an algorithm which runs in $O(m + n \log^{d-1} n)$ time using $O(n \log^{d-1} n)$ space.

Spinrad's approach can be improved using the *range trees with slack parameter*, introduced by Mehlhorn [5] and analyzed in detail by Smid [8]. This variation of the range tree, when used with the slack parameter $f(n) = \frac{\epsilon \log \log n}{d-1}$, allows to insert and delete d-dimensional points with a query time of $O(\frac{\log^{d+\epsilon} n}{\log \log^{d-1} n} + k)$, where k is the number of reported answers, an amortized update time of $O(\frac{\log^d n}{\log \log^{d-1} n})$, using $O(n \frac{\log^{d-1} n}{\log \log^{d-1} n})$*space*. This allows to compute the transitive closure representation of a d-dimensional poset in $O(m + n \frac{\log^{d-1+\epsilon} n}{\log \log^{d-2}})$ time using $O(n \frac{\log^{d-2} n}{\log \log^{d-2} n})$ space.

Also, note that Overmars ([7]:theorems 6.1 and 7.1) proves that a range query on a set of n points in the d-dimensional $n \times n$ grid can be answered in $O(k + \log^{d-2} n \log \log n)$ time using $O(n \log^{d-2} n)$ space, where k is the number of reported answers. This technique can be used to build the transitive closure graph of a d-dimensional poset from its realizer but only provides an $O(m + n \log^{d-2} n \log \log n)$ time algorithm and requires $O(n \log^{d-2} n)$ space.

In this paper we provide a better general algorithm using $O(m + n \log^{d-2} n)$ time and $O(n \log^{d-3} n)$ space, which is almost always better than previous solutions. Indeed, only for the special case $d = 4$ our general algorithm has a slightly worse space complexity. But for all other values of d our solution is equivalent (when $d = 3$) or better (when $d > 4$) than the others known in the literature.

We first give the solution for $d = 3$ and then discuss its generalization.

The paper is organized as it follows. In section 2 we give the basic definitions and show how data structures we use are initialized. In section 3 we show how to

[2] We use $\log^k n$ for $(\log n)^k$ and $\log \log^k n$ for $(\log \log n)^k$

produce the adjacency lists of the graph representing the given poset. Section 4 contains correctness proofs, the generalization technique and some final remarks.

2 Preliminaries

2.1 Basic definitions

Let $\mathcal{L}_1$, $\mathcal{L}_2$, and $\mathcal{L}_3$ be three linear extensions which are a realizer of poset $P = (X, \leq_P)$. Let $|X| = n$. We want to efficiently build the adjacency lists of the graph $G_P = (X, E)$ which is the transitive closure representation of P. We assume, for the sake of simplicity in algorithm description, $n = 2^h$ for some h. The extension to all values of n is straightforward and left to the reader.

Each linear extension $\mathcal{L}_i$, $i = 1, 2, 3$, is represented by an array $L_i[1..n]$, where $L_i[k]$, $k = 1, \ldots, n$, is the name of the element which is in position k in the i-th extension.

We make use of the following additional data structures:

- Two arrays $I_1[1..n]$ and $I_2[1..n]$ such that $I_1[k] - 1$ (resp. $I_2[k] - 1$) gives the number of elements following the element of name k in L_1 (resp. L_2);
- A tree T, similar to the x-*fast trie* of Willard [11], that is suitably initialized with information from $\mathcal{L}_1$ and $\mathcal{L}_2$;
- An array $N[1..n]$ used in building T;
- Two lists P_T and P_R, used during the phase of construction of adjacency lists of G_P to contain additional information related to the position in T of the current element.

The algorithm receives in input the three arrays L_1, L_2 and L_3 and produces in output the array $A[1..n]$ of pointers, such that $A[k]$ points to the list of elements adjacent to element of name k in the graph representing the given poset.

The algorithm has three phases. In the first phase the additional data structures are initialized. In the second phase tree T is adjusted. In the third phase the array A of adjacency lists for $G_P = (X, E)$ is produced.

Description is given by means of pseudo-code: for more details see [6].

2.2 Initialization

Arrays I_1 and I_2 are initialized as above described using, respectively, arrays L_1 and L_2.

Nodes in tree T are records with an elem field (representing the name of the element) and two pointers to the left son and to the right son.

A record is created for each element in $L_1[k]$. Array N is then initialized by assigning in position $n-k+1$ a pointer to the node representing the element of name $L_1[k]$.

Tree T is finally built by creating a balanced structure where all the internal nodes have the value zero in their elem field, and the leaves are the records pointed by array N. This is done so that the left-to-right order of the leaves corresponds to the order of elements in N, that is to the reverse order of the elements in $\mathcal{L}_1$. The height of T is $O(\log n)$, due to the balanced way it is built.

Array A of adjacency lists is initialized by inserting the loop edge corresponding to $x \leq_P x$ relations.

The time complexity of the first phase is $O(n)$ for initializing arrays I_1, I_2, N, and A plus the time for building T. This can be easily built in time $O(n)$, for example using a recursive approach that builds the tree from the leaves upwards.

Hence the first phase runs in linear time. The space used is $\Theta(n)$.

3 Computing the poset

3.1 Tree adjustment

In this phase elements in T are moved upwards so that the more an element is in front in $\mathcal{L}_2$ the highest it is moved in T. Moreover all useless (i.e. with the elem field equal to zero) nodes are pruned from T.

Hence after the adjustment phase T has the heap property with respect to $\mathcal{L}_2$, and the search tree property of T with respect to $\mathcal{L}_1$ has been weakened in a controlled way (specified by invariant 3.2 below).

Tree adjustment is done by means of a recursive visit of T, until a node with at least one son having a non-zero value is reached. At this point the son of the current node more in front in $\mathcal{L}_2$ is exchanged with the current node, that is sifted down in the tree.

This process is specified in more detail by the following high level description:

```
if the current element is not a leaf
    then
        ADJUST the left subtree; ADJUST the right subtree;
        EXCHANGE the current node with the son more in front in L2;
        if the exchanged son is a leaf
            then prune it from the tree
            else ADJUST the subtree rooted at the exchanged node;
```

The running time for the adjustment phase is given by the recurrence $P(k) = 2P(\frac{k}{2}) + O(\log n)$, since the only effect of the further ADJUST after the EXCHANGE is to make the zero value in the elem field of the current node sift down in the tree until to the leaf level. Since it is, for any constant $\epsilon < 1$, $\log n = O(n^{1-\epsilon})$ the above recurrence has solution $P(k) = O(n)$.

Hence the second phase runs in $O(n)$ time and uses $\Theta(n)$ space.

At the end of the adjustment phase tree T satisfies the following two invariants. Let $\pi(k)$ be the path (i.e. sequence of nodes) in T from the root to element k. Let $\pi'(k)$ be $\pi(k)$ minus the node representing element k.

Invariant 3.1 *Along $\pi(k)$ elements are encountered according to their order relation in $\mathcal{L}_2$.*

Invariant 3.2 *Each element j belonging to a subtree rooted at a node $x \notin \pi(k)$ such that x is a left (resp. right) son of a node in $\pi'(k)$ satisfies $j \geq_{\mathcal{L}_1} k$ (resp. $j \leq_{\mathcal{L}_1} k$).*

3.2 Building adjacency lists

This phase is accomplished one element at a time, starting from the last element in L_3. The process is done in three steps, according to the following high level algorithm.

algorithm ADJACENCES;

for k:= n **down to** 1 **do begin**
STEP-1: search in T element of name $L_3[k]$ using value $I_1[L_3[k]]$ and storing in P_T pointers to nodes in $\pi(L_3[k])$ and in P_R pointers to those right sons of nodes in $\pi(L_3[k])$ which are not in $\pi(L_3[k])$ themselves;
STEP-2: delete $L_3[k]$ from T;
STEP-3: check nodes in P_T and P_R for dominance with respect to k;

In STEP-1 the search for elements in T uses as search key the values in I_1. Given the way T has been initialized, the value $I_1[k]$ uniquely determines the search path in T for the element of name k. This digital search tree [1] property is maintained during the adjustment phase. Hence, element k will be found along such a path at a height which depends on the value $I_2[k]$: the larger is such a value the closer k is found to the root of T.

We denote with $\rho(T)$ the pointer to the root node of T. Searching for the element of name k is done by starting at $\rho(T)$ and following, like in digital search trees, the path given by the binary value of $I_1[k]$. During the search, lists P_T and P_R are created as above described.

Clearly, the search for each element is executed in $O(\log n)$ time. Lists P_T and P_R are emptied before each of such a call. When the current search terminates lists P_T and P_R have a $O(\log n)$ length.

Once the current search has returned the pointer to the desired node, such a node has to be deleted from T and the tree has to be adjusted. Deletion is done in STEP-2 by substituting the value in the elem field of the found element with the value of one of its sons, suitably chosen to maintain the tree invariants. This process is similar to that described in the adjustment phase. Clearly, deletion does not increase the height of T. Each deletion step runs in $O(\log n)$ time.

Elements which dominate k are now found in STEP-3 with the help of lists P_T and P_R, according to the following high level description (note that dominance testing for elements in P_T can be done using only $\mathcal{L}_1$):

for each element j in P_T
  if j dominates the current element k
    then adjacency with j is inserted in $A[k]$;

for each element j in P_R
  if j dominates the current element k
    then
      adjacency with j is inserted in $A[k]$;
      the dominance test is recursively repeated on both sons of j

The processing of nodes in P_T has a $O(\log n)$ running time. Let m_k denote the number of edges in the adjacency list of node k. Concerning the processing of nodes in P_R, it is easy to check that it has a $O(\max(m_k, \log n))$ running time since each time recursion is invoked exactly one edge is added to the adjacency list of node k. The space used in the third step is $\Theta(n)$.

The running time for the third phase is thus $O(n \log n + \sum_{k=1}^{n} \max(m_k, \log n))$, that is $O(m + n \log n)$. The space used is $\Theta(n)$.

Since both the first two phases run in linear time and space, the total running time for the poset computation is $O(m + n \log n)$ using $\Theta(n)$ space.

4 Correctness and generalization

4.1 Correctness

Correctness of the presented algorithms derives from the following theorems.

Theorem 4.1 *Each directed edge (x, y) in G_P corresponds to a couple (x, y) such that $x \leq_P y$.*

Proof. To prove the statement we prove that when the directed edge (x, y) is added to the adjacency list of x it is $(x \leq_{\mathcal{L}_1} y) \wedge (x \leq_{\mathcal{L}_2} y) \wedge (x \leq_{\mathcal{L}_3} y)$. But $x \leq_{\mathcal{L}_1} y$ derives from the test on L_1 and $x \leq_{\mathcal{L}_2} y$ derives for nodes in P_T from invariant 3.1 and for nodes in P_R from the test on L_2. Concerning the third linear extension, since nodes in L_3 are processed in reverse order and they are deleted from T after their processing, whichever node x is still in T when y is being processed clearly satisfies $x \leq_{\mathcal{L}_3} y$. □

To discuss the second theorem we need some notation. Let F_y be the set $\{x | x \leq_{\mathcal{L}_1} y\}$ and let S_y be the set $\{x \mid x \leq_{\mathcal{L}_2} y\}$. Let $\tau(P_R)$ denote the set of nodes in those subtrees whose roots are pointed by pointers in P_R. Similarly let $\tau(y)$ denote the set of nodes in the subtree whose root is element y.

We give a preliminary lemma.

Lemma 4.2 *When the search process in* STEP-1 *has found node y it is $F_y \cap S_y \subseteq P_T \cup \tau(P_R)$.*

Proof. Assume, by way of contradiction, that $x \in F_y \cap S_y$ but $x \notin P_T \cup \tau(P_R)$. The latter might happen in two ways: (i) $x \in \tau(y)$ or (ii) $x \in \tau(w)$, where w is the left son of a node in $\pi'(y)$ and $w \notin \pi(y)$. But (i) is absurd since, by invariant 3.1, if $x \in \tau(y)$ then y precedes x in the first linear extension. Also (ii) is absurd since, by invariant 3.2, if $x \in \tau(w)$ then y precedes x in the second linear extension. □

Theorem 4.3 *For each couple (x, y) such that $x \leq_P y$ a directed edge (x, y) is present in G_P.*

Proof. From the previous lemma and the way algorithms above work, each time it is $x \leq_P y$ element x is found during the processing of element y either in P_T or in $\tau(P_R)$ and y is added to list $A[x]$. □

From the above discussion we have therefore proved the following theorem:

Theorem 4.4 *Given a 3-dimensional poset $P = (X, \leq_P)$ on n elements by means of three linear extensions realizing it, the graph $G_P = (X, E)$ which is the transitive closure representation of P can be produced in $O(m + n \log n)$ worst-case running time, where $m = |E|$, using $\Theta(n)$ worst-case space.*

□

4.2 Generalization

We now discuss how to extend the result of theorem 4.4 to a dimension $d \geq 4$. Following Overmars [7] we use a method of Willard and Lueker [12]. Let T_{d-1}

be the data structure used to solve the problem for dimension d. For $d \geq 4$ the last linear extension is set aside and the remaining $d-1$ are considered to build structure T_{d-1} by the following recursive process.

Let T_k, $k > 2$, be the structure to be built with the remaining k linear extension. A balanced binary tree (*primary structure*) is built on the n elements according to their order in the last of the remaining k linear extensions, and the elements are stored in the leaves. Each internal node x contains in a $(k-1)$-dimensional version of the structure (*secondary structure*) all the elements in the leaves which are descendants of node x, taking into account the remaining $(k-1)$ linear extensions. If $k-1=2$ then T_2 is the tree T discussed in the previous sections and the recursion terminates.

Let $R(d,n)$ denote the time needed to initialize the structure T_{d-1} for n elements. From the previous discussion and the analysis for $d=3$ follows $R(4,n) = O(n) + 2R(4,\frac{n}{2})$ whose solution is $R(4,n) = O(n \log n)$. The general case is given by the solution to the recurrence $R(d,n) = R(d-1,n) + 2R(d,\frac{n}{2})$, that is $R(d,n) = O(n \log^{d-3} n)$.

We now consider elements in the inverse order as they appear in the d-th linear extension. Assume the element currently considered is y.

To find elements x such that $x \leq_P y$ a search is done in the primary structure for element y. Let $\pi(y)$ be the path, in the primary structure, from the root to the leaf containing y. Let $\sigma(y)$ denote the set of nodes which are left sons of a node in $\pi(y)$ but do not belong to $\pi(y)$. Nodes in $\sigma(y)$ identify $O(\log n)$ secondary structures which are (recursively) queried with y.

Remember that element y is deleted from trees of type T_2 after the query has been answered. Hence, each element x returned from (recursive) queries to secondary structures cleary satisfies $x \leq_P y$.

Let $P(d)$ and $S(d)$ denote, respectively, the worst-case running time and the worst-case space used by structure T_{d-1} to build k edges of the graph representing a d-dimensional poset on n elements. It is easy to check that $P(d)$ and $S(d)$ are expressed by the solution to the following recurrences:

$$\begin{cases} P(d) &= O(k) + Q(d) \\ Q(d) &= O(\log n) Q(d-1) \\ Q(4) &= O(\log n) O(n \log n) \end{cases}$$

and

$$\begin{cases} S(d) &= O(\log n) S(d-1) \\ S(4) &= O(\log n) \Theta(n) \end{cases}$$

We can therefore conclude with the following theorem:

Theorem 4.5 *Given a d-dimensional poset $P = (X, \leq_P)$ on n elements by means of d linear extensions realizing it, the graph $G_P = (X, E)$ which is the*

transitive closure representation of P can be produced in $O(m + n \log^{d-2} n)$ worst-case running time, where $m = |E|$, using $O(n \log^{d-3} n)$ worst-case space.

□

Acknowledgments. We thank both referees for their valuable comments that greatly helped in improving the presentation. Thanks also to one of the referees for suggesting the use of range trees with slack parameter to improve Spinrad's approach.

References

[1] Daniel E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, Reading, MA, 1975.

[2] D.T. Lee and Franco P. Preparata. An improved algorithm for the rectangle enclosure problem. *Journal of Algorithms*, 3(3):218–224, 1982.

[3] George S. Lueker. A data structure for orthogonal range queries. In *19th Annual Symposium on Foundations of Computer Science (FOCS'78)*, pages 28–34, Ann Arbor, Mich., October 1978. IEEE.

[4] R. McConnell and Jeremy Spinrad. Linear-time modular decomposition and efficient transitive orientation of comparability graphs. In *5th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'94)*, pages 536–545, 1994.

[5] Kurt Mehlhorn. *Data Structures and Algorithms 3: Multi-Dimensional Searching and Computational Geometry*, volume 3 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, 1984.

[6] Enrico Nardelli, Vincenzo Mastrobuoni, and Alesiano Santomo. Implementing the computation of a poset from its realizer. Technical Report n.128, Dipartimento di Matematica Pura ed Applicata, Università di L'Aquila, L'Aquila, January 1997.

[7] Mark H. Overmars. Efficient data structures for range searching on a grid. *Journal of Algorithms*, 9:254–275, 1988.

[8] Michiel H. Smid. Range trees with slack parameter. *ALCOM: Algorithms Review*, 2:77–87, 1991.

[9] Jeremy Spinrad. Dimension and algorithms. In Vincent Bouchitté and Michel Morvan, editors, *International Workshop on Orders, Algorithms, and Applications (ORDAL'94)*, pages 33–52, Lyon, France, July 1994. Lecture Notes in Computer Science n.831, Springer-Verlag.

[10] Dan E. Willard. Predicate-oriented database search algorithms. Ph. D. Dissertation, Harvard University, Cambridge, Mass., September 1978. Available as Tech.Rep TR-20-78, Center for Research in Computing Technology.

[11] Dan E. Willard. Log-logarithmic worst-case range queries are possible in space θ(N). *Information Processing Letters*, 17:81–84, 1983.

[12] Dan E. Willard and George S. Lueker. Adding range restriction capability to dynamic data structures. *Journal of the Association for Computing Machinery*, 32(3):597–617, July 1985.