On Building the Transitive Reduction of a Two-Dimensional Poset *

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: May 5, 1997

Abstract

In this paper we provide an efficient algorithm for computing the graph representing the transitive reduction of a two dimensional poset which is given by means of the two 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.

Given a poset $P = (X, \leq_P)$, a directed graph $H_P = (X, E)$ representing P can be defined, such that $(x, y) \in E \Leftrightarrow (x \leq_P y) \wedge \neg[(w \in X) \wedge (w \neq x) \wedge (w \neq$

*Research partially supported by the Italian MURST 40% project "Algoritmi, Modelli di Calcolo e Strutture Informative".

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

$y) \wedge (x \leq_P w) \wedge (w \leq_P y)]$. Sometimes H_P is also called the *transitive reduction representation* 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 [6] is how to efficiently build the adjacency lists of the directed graph $H_P = (X, E)$ which is the transitive reduction representation of P.

Ma and Spinrad [4] give for two-dimensional posets an algorithm which runs in worst-case $O(n^2)$ time and worst-case $O(n)$ space, where $n = |X|$. This is optimal only for posets whose transitive reduction representation is such that $m = |E| = \Omega(n^2)$. Whenever the poset is such that $m = o(n^2)$ there is the possibility of an improvement.

Güting, Nurmi, and Ottmann [1] use computational geometry techniques to provide an $O(m + n \log n)$ worst-case time and $O(n)$ worst-case space algorithm to compute all the m direct (i.e. non-redundant) dominances in a set of two-dimensional points of the plane. Their approach therefore directly provides a better algorithm than Ma and Spinrad's for $m = o(n^2)$. But for sparser posets such that $m = o(n \log n)$ there is still some space for improvement.

In this paper we provide a better algorithm for the case $m = o(\frac{n \log n}{\log \log n})$ at the price of a slightly higher space usage. Our algorithm, in fact, runs in $O(m \log \log n + n \frac{\log n}{\log \log n})$ worst-case time using $O(n \frac{\log n}{\log \log n})$ worst-case space. From now on, when speaking of 'time' we always mean 'worst-case time' and with 'space' we mean 'worst-case space'.

The paper is organized as follows. In section 2 we give the basic definitions and introduce the used techniques. In section 3 the data structure allowing us to obtain the cited bounds is described. Finally, section 4 shows how to produce the adjacency lists of the transitive reduction graph of the given poset.

2 Preliminaries

Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be two 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 $H_P = (X, E)$ which is the transitive reduction representation of P.

Each linear extension $\mathcal{L}_i$, $i = 1, 2$, 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 also use an array $I[1..n]$ such that $I[k]$ gives the position where element of name k occurs in L_1.

Let us consider the poset elements as a set of 2-dimensional points in the plane, with coordinates given from their position in the two linear extensions.

Let us define a *direct dominance* relation between two points $P = (P_x, P_y)$ and $Q = (Q_x, Q_y)$ as follows. P directly dominates Q, written $Q \prec P$, if $(Q_x < P_x) \wedge (Q_y < P_y)$ and no other point $R \neq Q$ exists such that $(Q \prec R) \wedge (R \prec P)$. Clearly, the direct dominance relation is exactly the set of edges of the transitive reduction representation of the poset.

Following Güting, Nurmi and Ottmann [1], we note that the zone of the plane directly dominated by a point P is the whole rectangle having as corners the origin and P, less the union of those rectangles having as corners the origin and every point Q which is directly dominated by P.

Our approach to compute direct dominances is based on the *line-sweep* technique from computational geometry [5]. We move a horizontal line starting from the highest point in the vertical direction towards the origin of the coordinate axes. Each time a point is encountered some direct dominances between the current point and points previously considered are possibly produced. Hence the basic approach is to cycle through the set of elements in descending order of their second linear extension.

The critical point is the data structure used to find and report direct dominances. In a traditional approach to direct dominance the line-sweep technique can be combined with a data structure called *interval tree* [5] to obtain all direct dominances in $O((m+n)\log n)$ time using $O(n)$ space.

Güting, Nurmi, and Ottmann [1] combine the line-sweep technique with a *divide and conquer* approach and the use of a stack instead of an interval tree to improve the time bound to $O(m + n\log n)$, while maintaining a $O(n)$ space usage.

We resort to the more traditional approach but couple it with our variation of a data structure defined ad-hoc for finite domains, namely the *interval trie* of Karlsson and Overmars [3]. This is a dynamic data structure used to solve the *1-dimensional stabbing problem* [5] in a finite domain. That is, given a set of n 1-dimensional intervals whose endpoints are from the domain $U = [0..u-1]$, the interval trie supports insertion and deletions of intervals and efficiently answers stabbing queries, i.e. queries asking for all intervals containing the query point $p \in U$.

In [3] there is no space analysis for the interval trie, but an implementation directly derived from the description in the paper produces a worst-case space usage of $O(u^2)$ and an initialization time of $O(u \log\log u)$.

In fact, an interval trie is made up of $O(\frac{\log u}{\log\log u})$ substructures. The i-th substructure stores only the intervals of length between $(\log^\epsilon u)^{i-1}$ and $(\log^\epsilon u)^i$, for some constant $\epsilon > 0$. The i-th substructure uses one global Johnson tree [2] on the domain $U = [0..u-1]$ and has $O(\frac{u}{(\log^\epsilon u)^{i-1}})$ parts. For each part there are two Johnson trees on the domain U, containing respectively those left and right endpoints of intervals falling into the part, and one list containing the name of intervals wholly covering that part. Each interval wholly covers $O(\log^\epsilon u)$

parts of the substructure it belongs to. For more details on the interval trie the reader is referred to [3]. A Johnson tree on the domain U uses $O(u)$ space and is initialized in $O(\log\log u)$ time. This gives for the whole interval trie a space usage of (big-oh is omitted in summation upper bounds to improve readability):

$$\sum_{i=1}^{\frac{\log u}{\log\log u}} \sum_{j=1}^{\frac{u}{(\log^\epsilon u)^{i-1}}} O(u) = \sum_{i=1}^{\frac{\log u}{\log\log u}} O(\frac{u}{(\log^\epsilon u)^{i-1}} u) =$$

$$= O(u^2) \sum_{i=1}^{\frac{\log u}{\log\log u}} O(\frac{1}{(\log^\epsilon u)^{i-1}}) = O(u^2),$$

and, analogously, an initialization time of:

$$\sum_{i=1}^{\frac{\log u}{\log\log u}} \sum_{j=1}^{\frac{u}{(\log^\epsilon u)^{i-1}}} O(\log\log u) = O(u \log\log u).$$

The following lemma is therefore a directly derivable corollary of results discussed in [3].

Lemma 2.1 *Given a set of n 1-dimensional intervals whose endpoints are from the domain $[1..n]$ an interval trie exists solving the stabbing problem with a query time of $O(k + \frac{\log n}{\log\log n})$ (where k is the number of reported answers), insertion and deletion times of $O(\log\log n)$, and an initialization time of $O(n \log\log n)$ using $O(n^2)$ space.*

3 The re-normalized interval trie

We improve on the space usage by defining a *re-normalized interval trie* that introduces two changes in the Karlsson and Overmars' structure.

The first modification derives from the observation that each part of the i-th substructure of an interval trie only contains $O((\log^\epsilon u)^{i-1})$ different points of the overall domain U. Hence in each part we can re-normalize the overall domain of size $O(u)$ to the $O((\log^\epsilon u)^{i-1})$ size of the domain considered in that part.

From this observation it follows that the two Johnson trees used for each part of the substructure i only have a $O((\log^\epsilon u)^{i-1})$ size and are initialized in $O(\log(i-1) + \log\log\log u)$ time.

A second change comes from observing that we can discard the global Johnson tree used in each of the $O(\frac{\log u}{\log\log u})$ substructures. The global Johnson tree is used during insertion and deletion of intervals to access and update in $O(\log\log u)$ the list of all those parts that are covered by the considered interval. But we can, during the insertion of an interval, thread together all

the occurences of the interval that are inserted in these lists and connect the occurence in the leftmost list to the left endpoint of the interval in the corresponding Johnson tree. If we implement these lists as bi-directionally linked lists, during the deletion of an interval we can start from the left endpoint of the interval and delete all the inserted occurences within the same time bound.

With the modifications above described we maintain for all the operations the same time bounds of the standard interval trie, but now the overall space used by the *re-normalized interval trie* is

$$\sum_{i=1}^{\frac{\log u}{\log\log u}} \sum_{j=1}^{\frac{u}{(\log^\epsilon u)^{i-1}}} O((\log^\epsilon u)^{i-1}) =$$

$$= \sum_{i=1}^{\frac{\log u}{\log\log u}} O(\frac{u}{(\log^\epsilon u)^{i-1}}(\log^\epsilon u)^{i-1}) = O(u) \sum_{i=1}^{\frac{\log u}{\log\log u}} 1 = O(u\frac{\log u}{\log\log u}),$$

and, analogously, the initialization time is:

$$\sum_{i=1}^{\frac{\log u}{\log\log u}} \sum_{j=1}^{\frac{u}{(\log^\epsilon u)^{i-1}}} O(\log(i-1) + \log\log\log u) =$$

$$= u\log\log\log u \sum_{i=1}^{\frac{\log u}{\log\log u}} O(\frac{1}{(\log^\epsilon u)^{i-1}}) + u \sum_{i=1}^{\frac{\log u}{\log\log u}} O(\frac{\log(i-1)}{(\log^\epsilon u)^{i-1}}) =$$

$$= O(u\log\log\log u).$$

From the above discussion and the results in [3] the following theorem derives.

Theorem 3.1 *Given a set of n 1-dimensional intervals whose endpoints are from the domain $[1..n]$ a re-normalized interval trie exists solving the stabbing problem with a query time of $O(k + \frac{\log n}{\log\log n})$ (where k is the number of reported answers), insertion and deletion times of $O(\log\log n)$, and an initialization time of $O(n\log\log\log n)$ using $O(n\frac{\log n}{\log\log n})$ space.*

4 Building adjacency lists

Algorithms are described in a Pascal-like form, omitting **begin-end** whenever deducible from typografical layout.

Arrays I is initialized by the following procedure:

procedure POSITION {initialize array I depending on array L_1}
var k: **integer**;

for k:=1 **to** n **do** $I[L_1[\mathsf{k}]]$:=k;

Building of adjacency lists is done through the following algorithm:

algorithm ADJACENCES;

Let S be an empty and initialized re-normalized interval trie;
Let l be an empty list of integers;
Insert interval with name n and endpoints $[0, I[L_2[n]]]$ into S;
for k:=$n-1$ **down to** 1 **do**
    p:=$I[L_2[k]]$;
    l:= the list of names of those intervals that contain p in S;
    add to the adjacency list of element k all the elements in l;
    insert interval with name k and endpoints $[0, I[L_2[k]]]$ into S;
    for each interval with name i in l **do**
        delete i from S;
        insert interval with name i and endpoints $[p, I[L_2[i]]]$ into S;

The algorithm above correctly reports all direct dominances, since each time a new interval enters in the re-normalized interval trie the existing intervals which directly dominate it are updated by subtracting from their extension the span covered by the new interval.

The time complexity of the algorithm is given by the initialization time plus the time needed to answer the $n-1$ stabbing queries plus the time needed to update intervals. Let m be the total number of reported answers to stabbing queries. Then time complexity for all the stabbing queries is $O(m + n\frac{\log n}{\log\log n})$ and for all updates is $O(m \log\log n)$. Initialization time is a lower order term, hence from lemma 2.1 and theorem 3.1 we have the claimed result.

References

[1] Ralf-Hartmut Güting, Otto Nurmi, and Thomas Ottmann. Fast algorithms for direct enclosures and direct dominances. *Journal of Algorithms*, 10:170–186, 1989.

[2] Donald B. Johnson. A priority queue in which initialization and queue operations take $O(\log \log D)$ time. *Mathematical Systems Theory*, 15:295–309, 1982.

[3] Rolf G. Karlsson and Mark H. Overmars. Scanline algorithms on a grid. *BIT*, 28:227–241, 1988.

[4] T.-H. Ma and Jeremy Spinrad. Transitive closure for restricted classes of partial orders. *Order*, 8:175–183, 1991.

[5] Franco P. Preparata and Michael I. Shamos. *Computational Geometry*. Text and Monographs in Computer Science. Springer-Verlag, 1985.

[6] 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.